# Guia da Linguagem de Programação do VisuAlg*

* Este guia foi integralmente copiado da opção "A Linguagem do VisuAlg" a partir do menu Ajuda. Pode-se obter ajuda específica de um determinado comando escolhendo a opção "Referência da linguagem do VisuAlg" no mesmo menu.

**Introdução**

A linguagem que o VisuAlg interpreta é bem simples: é uma versão portuguesa dos pseudocódigos largamente utilizados nos livros de introdução à programação, conhecida como "Portugol". Tomei a liberdade de acrescentar-lhe alguns comandos novos, com o intuito de criar facilidades específicas para o ensino de técnicas de elaboração de algoritmos. Inicialmente, pensava em criar uma sintaxe muito simples e "liberal", para que o usuário se preocupasse apenas com a lógica da resolução dos problemas e não com as palavras-chave, pontos e vírgulas, etc. No entanto, cheguei depois à conclusão de que alguma formalidade seria não só necessária como útil, para criar um sentido de disciplina na elaboração do "código-fonte".

A linguagem do VisuAlg permite apenas **um comando por linha**: desse modo, não há necessidade de *tokens* separadores de estruturas, como o ponto e vírgula em Pascal. Também não existe o conceito de blocos de comandos (que correspondem ao *begin* e *end* do Pascal e ao { e } do C), nem comandos de desvio incondicional como o *goto*. Na versão atual do VisuAlg, com exceção das rotinas de entrada e saída, **não** há nenhum subprograma embutido, tal como *Inc()*, *Sqr()*, *Ord()*, *Chr()*, *Pos()*, *Copy()* ou outro.

**Importante:** para facilitar a digitação e evitar confusões, todas as palavras-chave do VisuAlg foram implementadas sem acentos, cedilha, etc. Portanto, o tipo de dados *lógico* é definido como `logico`, o comando *se..então..senão* é definido como `se..entao..senao`, e assim por diante. O VisuAlg também não distingue maiúsculas e minúsculas no reconhecimento de palavras-chave e nomes de variáveis.

**Formato Básico do Pseudocódigo e Inclusão de Comentários**

O formato básico do nosso pseudocódigo é o seguinte:

```
algoritmo "semnome"
// Função :
// Autor :
// Data :
// Seção de Declarações
inicio
// Seção de Comandos
fimalgoritmo
```

A primeira linha é composta pela palavra-chave `algoritmo` seguida do seu nome delimitado por aspas duplas. Este nome será usado como título nas janelas de leitura de dados (nas futuras versões do VisuAlg, talvez utilizemos este dado de outras formas). A seção que se segue é a de declaração de variáveis, que termina com a linha que contém a palavra-chave `inicio`. Deste ponto em diante está a seção de comandos, que continua até a linha em que se encontre a palavra-chave `fimalgoritmo`. Esta última linha marca o final do pseudocódigo: todo texto existente a partir dela é ignorado pelo interpretador.

O VisuAlg permite a inclusão de comentários: qualquer texto precedido de "//" é ignorado, até se atingir o final da sua linha. Por este motivo, os comentários não se estendem por mais de uma linha: quando se deseja escrever comentários mais longos, que ocupem várias linhas, cada uma delas deverá começar por "//".

**Tipos de Dados**

O VisuAlg prevê quatro tipos de dados: **inteiro**, **real**, **cadeia de caracteres** e **lógico** (ou *booleano*). As palavras-chave que os definem são as seguintes (observe que elas não têm acentuação):

- `inteiro`: define variáveis numéricas do tipo inteiro, ou seja, sem casas decimais.
- `real`: define variáveis numéricas do tipo real, ou seja, com casas decimais.

- `caractere`: define variáveis do tipo *string*, ou seja, cadeia de caracteres.
- `logico`: define variáveis do tipo *booleano*, ou seja, com valor VERDADEIRO ou FALSO.

O VisuAlg permite também a declaração de variáveis estruturadas através da palavra-chave `vetor`, como será explicado a seguir.

**Nomes de Variáveis e sua Declaração**

Os nomes das variáveis devem começar por uma letra e depois conter letras, números ou *underline*, até um limite de 30 caracteres. As variáveis podem ser simples ou estruturadas (na versão atual, os vetores podem ser de uma ou duas dimensões). Não pode haver duas variáveis com o mesmo nome, com a natural exceção dos elementos de um mesmo vetor.

A seção de declaração de variáveis começa com a palavra-chave `var`, e continua com as seguintes sintaxes:

```
<lista-de-variáveis> : <tipo-de-dado>
<lista-de-variáveis> : vetor "["<lista-de-intervalos>"]" de <tipo-de-dado>
```

Na *<lista-de-variáveis>*, os nomes das variáveis estão separados por vírgulas. Na *<lista-de-intervalos>*, os *<intervalo>* são separados por vírgulas, e têm a seguinte sintaxe:

```
<intervalo>: <valor-inicial> .. <valor-final>
```

Na versão atual do VisuAlg, tanto *<valor-inicial>* como *<valor-final>* devem ser inteiros. Além disso, exige-se evidentemente que *<valor-final>* seja maior do que *<valor-inicial>*.

Exemplos:

```
var a: inteiro
    Valor1, Valor2: real
    vet: vetor [1..10] de real
    matriz: vetor [0..4,8..10] de inteiro
    nome_do_aluno: caractere
    sinalizador: logico
```

Note que não há a necessidade de ponto e vírgula após cada declaração: basta pular linha. A declaração de vetores é análoga à linguagem Pascal: a variável `vet` acima tem 10 elementos, com os índices de [1] a [10], enquanto `matriz` corresponde a 15 elementos com índices [0,8], [0,9], [0,10], [1,8], [1,9], [1,10], ... até [4,10]. O número total de variáveis suportado pelo VisuAlg é 500 (cada elemento de um vetor é contado individualmente).

**Constantes e Comando de Atribuição**

O VisuAlg tem três tipos de constantes:

- *Numéricos*: são valores numéricos escritos na forma usual das linguagens de programação. Podem ser inteiros ou reais. Neste último caso, o separador de decimais é o ponto e não a vírgula, independente da configuração regional do computador onde o VisuAlg está sendo executado. O VisuAlg também não suporta separadores de milhares.
- *Caracteres*: qualquer cadeia de caracteres delimitada por aspas duplas (").
- *Lógicos*: admite os valores VERDADEIRO ou FALSO.

A atribuição de valores a variáveis é feita com o operador `<-`. Do seu lado esquerdo fica a variável à qual está sendo atribuído o valor, e à sua direita pode-se colocar qualquer expressão (constantes, variáveis, expressões numéricas), desde que seu resultado tenha tipo igual ao da variável.

Alguns exemplos de atribuições, usando as variáveis declaradas acima:

```
a <- 3
Valor1 <- 1.5
Valor2 <- Valor1 + a
vet[1] <- vet[1] + (a * 3)
matriz[3,9] <- a/4 - 5
nome_do_aluno <- "José da Silva"
sinalizador <- FALSO
```

**Operadores Aritméticos**

| | |
|---|---|
| +,- | Operadores unários, isto é, são aplicados a um único operando. São os operadores aritméticos de maior precedência. Exemplos: -3, +x. Enquanto o operador unário - inverte o sinal do seu operando, o operador + não altera o valor em nada o seu valor. |
| \ | Operador de divisão inteira. Por exemplo, 5 \ 2 = 2. Tem a mesma precedência do operador de divisão tradicional. |
| +,-,*,/ | Operadores aritméticos tradicionais de adição, subtração, multiplicação e divisão. Por convenção, * e / têm precedência sobre + e -. Para modificar a ordem de avaliação das operações, é necessário usar parênteses como em qualquer expressão aritmética. |
| MOD ou % | Operador de módulo (isto é, resto da divisão inteira). Por exemplo, 8 MOD 3 = 2. Tem a mesma precedência do operador de divisão tradicional. |
| ^ | Operador de potenciação. Por exemplo, 5 ^ 2 = 25. Tem a maior precedência entre os operadores aritméticos binários (aqueles que têm dois operandos). |

**Operadores de Caracteres**

| | |
|---|---|
| + | Operador de concatenação de *strings* (isto é, cadeias de caracteres), quando usado com dois valores (variáveis ou constantes) do tipo "caractere". Por exemplo: "Rio " + " de Janeiro" = "Rio de Janeiro". |

**Operadores Relacionais**

| | |
|---|---|
| =, <, >, <=, >=, <> | Respectivamente: igual, menor que, maior que, menor ou igual a, maior ou igual a, diferente de. São utilizados em expressões lógicas para se testar a relação entre dois valores do mesmo tipo. Exemplos: 3 = 3 ( 3 é igual a 3?) resulta em VERDADEIRO ; "A" > "B" ("A" está depois de "B" na ordem alfabética?) resulta em FALSO. |

**Importante:** No VisuAlg, as comparações entre *strings* **não diferenciam** as letras maiúsculas das minúsculas. Assim, "ABC" é igual a "abc". Valores lógicos obedecem à seguinte ordem: FALSO < VERDADEIRO.

**Operadores Lógicos**

| | |
|---|---|
| nao | Operador unário de negação. nao VERDADEIRO = FALSO, e nao FALSO = VERDADEIRO. Tem a maior precedência entre os operadores lógicos. Equivale ao NOT do Pascal. |
| ou | Operador que resulta VERDADEIRO quando um dos seus operandos lógicos for verdadeiro. Equivale ao OR do Pascal. |
| e | Operador que resulta VERDADEIRO somente se seus dois operandos lógicos forem verdadeiros. Equivale ao AND do Pascal. |
| xou | Operador que resulta VERDADEIRO se seus dois operandos lógicos forem diferentes, e FALSO se forem iguais. Equivale ao XOR do Pascal. |

**Comandos de Saída de Dados**

```
escreva (<lista-de-expressões>)
```

Escreve no dispositivo de saída padrão (isto é, na área à direita da metade inferior da tela do VisuAlg) o conteúdo de cada uma das expressões que compõem *<lista-de-expressões>*. As expressões dentro desta lista devem estar separadas por vírgulas; depois de serem avaliadas, seus resultados são impressos na ordem indicada. É equivalente ao comando *write* do Pascal.

De modo semelhante a Pascal, é possível especificar o número de espaços no qual se deseja escrever um determinado valor. Por exemplo, o comando `escreva(x:5)` escreve o valor da variável `x` em 5 espaços, alinhado-o à direita. Para variáveis reais, pode-se também especificar o número de casas fracionárias que serão exibidas. Por exemplo, considerando `y` como uma variável real, o comando `escreva(y:6:2)`escreve seu valor em 6 espaços colocando 2 casas decimais.

```
escreval (<lista-de-expressões>).
```

Idem ao anterior, com a única diferença que pula uma linha em seguida. É equivalente ao *writeln* do Pascal.

Exemplos:

```
algoritmo "exemplo"
var x: real
    y: inteiro
    a: caractere
    l: logico
inicio
x <- 2.5
y <- 6
a <- "teste"
l <- VERDADEIRO
escreval ("x", x:4:1, y+3:4) // Escreve: x 2.5    9
escreval (a, "ok")           // Escreve: testeok (e depois pula linha)
escreval (a, " ok")          // Escreve: teste ok (e depois pula linha)
escreval (a + " ok")         // Escreve: teste ok (e depois pula linha)
escreva (l)                  // Escreve: VERDADEIRO
fimalgoritmo
```

Note que o VisuAlg separa expressões do tipo numérico e lógico com um espaço à esquerda, mas não as expressões do tipo caractere, para que assim possa haver a concatenação. Quando se deseja separar expressões do tipo caractere, é necessário acrescentar espaços nos locais adequados.

**Comando de Entrada de Dados**

```
leia (<lista-de-variáveis>)
```

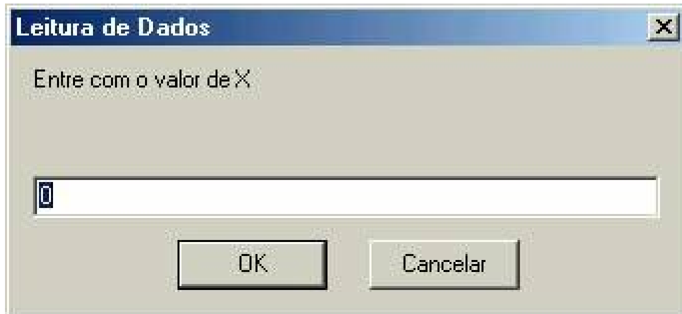
Recebe valores digitados pelos usuário, atribuindo-os às variáveis cujos nomes estão em *<lista-de-variáveis>* (é respeitada a ordem especificada nesta lista). É análogo ao comando *read* do Pascal.

Veja no exemplo abaixo o resultado:

```
algoritmo "exemplo 1"                                                      var
x: inteiro;
inicio
leia (x)
escreva (x)
fimalgoritmo
```

O comando de leitura acima irá exibir uma janela como a que se vê ao lado, com a mensagem padrão:

```
"Entre com o valor de <nome-de-variável>"
```




Se você clicar em *Cancelar* ou teclar *Esc* durante a leitura de dados, o programa será imediatamente interrompido.

**Comando de Desvio Condicional**

```
se <expressão-lógica> entao
    <seqüência-de-comandos>
fimse
```

Ao encontrar este comando, o VisuAlg analisa a `<expressão-lógica>`. Se o seu resultado for `VERDADEIRO`, todos os comandos da `<seqüência-de-comandos>` (entre esta linha e a linha com `fimse`) são executados. Se o resultado for `FALSO`, estes comandos são desprezados e a execução do algoritmo continua a partir da primeira linha depois do `fimse`.

```
se <expressão-lógica> entao
    <seqüência-de-comandos-1>
senao
    <seqüência-de-comandos-2>
fimse
```

Nesta outra forma do comando, se o resultado da avaliação de `<expressão-lógica>` for `VERDADEIRO`, todos os comandos da `<seqüência-de-comandos-1>` (entre esta linha e a linha com `senao`) são executados, e a execução continua depois a partir da primeira linha depois do `fimse`. Se o resultado for `FALSO`, estes comandos são desprezados e o algoritmo continua a ser executado a partir da primeira linha depois do `senao`, executando todos os comandos da `<seqüência-de-comandos-2>` (até a linha com `fimse`).

Estes comandos equivalem ao *if...then* e *if...then...else* do Pascal. Note que não há necessidade de delimitadores de bloco (como *begin* e *end*), pois as seqüências de comandos já estão delimitadas pelas palavras-chave `senao` e `fimse`. O VisuAlg permite o aninhamento desses comandos de desvio condicional.

**Comando de Seleção Múltipla**

O VisuAlg implementa (com certas variações) o comando *case* do Pascal. A sintaxe é a seguinte:

```
escolha <expressão-de-seleção>
caso <exp11>, <exp12>, ..., <exp1n>
   <seqüência-de-comandos-1>
caso <exp21>, <exp22>, ..., <exp2n>
   <seqüência-de-comandos-2>
...
outrocaso
   <seqüência-de-comandos-extra>
fimescolha
```

Veja o exemplo a seguir, que ilustra bem o que faz este comando:

```
algoritmo "Times"
var time: caractere
inicio
escreva ("Entre com o nome de um time de futebol: ")
leia (time)
escolha time
caso "Flamengo", "Fluminense", "Vasco", "Botafogo"
   escreval ("É um time carioca.")
caso "São Paulo", "Palmeiras", "Santos", "Corínthians"
   escreval ("É um time paulista.")
outrocaso
   escreval ("É de outro estado.")
fimescolha
fimalgoritmo
```

**Comandos de Repetição**

O VisuAlg implementa as três estruturas de repetição usuais nas linguagens de programação: o laço contado `para...ate...faca` (similar ao *for...to...do* do Pascal), e os laços condicionados `enquanto...faca` (similar ao *while...do*) e `repita...ate` (similar ao *repeat...until*). A sintaxe destes comandos é explicada a seguir.

**Para ... faça**

Esta estrutura repete uma seqüência de comandos um determinado número de vezes.

```
para <variável> de <valor-inicial> ate <valor-limite> [passo <incremento>] faca
   <seqüência-de-comandos>
fimpara
```

| | |
|---|---|
| *<variável >* | É a variável contadora que controla o número de repetições do laço. Na versão atual, deve ser necessariamente uma variável do tipo `inteiro`, como todas as expressões deste comando. |
| *<valor-inicial>* | É uma expressão que especifica o valor de inicialização da variável contadora antes da primeira repetição do laço. |
| *<valor-limite >* | É uma expressão que especifica o valor máximo que a variável contadora pode alcançar. |
| *<incremento >* | É opcional. Quando presente, precedida pela palavra `passo`, é uma expressão que especifica o incremento que será acrescentado à variável contadora em cada repetição do laço. Quando esta opção não é utilizada, o valor padrão de *<incremento>* é 1. Vale a pena ter em conta que também é possível especificar valores negativos para *<incremento>*. Por outro lado, se a avaliação da expressão *<incremento >* resultar em valor nulo, a execução do algoritmo será interrompida, com a impressão de uma mensagem de erro. |
| `fimpara` | Indica o fim da seqüência de comandos a serem repetidos. Cada vez que o programa chega neste ponto, é acrescentado à variável contadora o valor de *<incremento >*, e comparado a *<valor-limite >*. Se for menor ou igual (ou maior ou igual, quando *<incremento >* for negativo), a seqüência de comandos será executada mais uma vez; caso contrário, a execução prosseguirá a partir do primeiro comando que esteja após o `fimpara`. |

*<valor-inicial >, <valor-limite > e <incremento >* são avaliados uma **única vez** antes da execução da primeira repetição, e **não se alteram durante a execução do laço**, mesmo que variáveis eventualmente presentes nessas expressões tenham seus valores alterados.

No exemplo a seguir, os números de 1 a 10 são exibidos em ordem crescente.

```
algoritmo "Números de 1 a 10"
var j: inteiro
inicio
para j de 1 ate 10 faca
   escreva (j:3)
fimpara
fimalgoritmo
```

**Importante:** Se, logo no início da primeira repetição, *<valor-inicial >* for maior que *<valor-limite >* (ou menor, quando *<incremento>* for negativo), o laço não será executado nenhuma vez. O exemplo a seguir não imprime nada.

```
algoritmo "Numeros de 10 a 1 (não funciona)"
var j: inteiro
inicio
para j de 10 ate 1 faca
   escreva (j:3)
fimpara
fimalgoritmo
```

Este outro exempo, no entanto, funcionará por causa do **passo -1**:

```
algoritmo "Numeros de 10 a 1 (este funciona)"
var j: inteiro
inicio
para j de 10 ate 1 passo -1 faca
   escreva (j:3)
fimpara
fimalgoritmo
```

**Enquanto ... faça**

Esta estrutura repete uma seqüência de comandos enquanto uma determinada condição (especificada através de uma expressão lógica) for satisfeita.

```
enquanto <expressão-lógica> faca
   <seqüência-de-comandos>
fimenquanto
```

| | |
|---|---|
| *<expressão-lógica>* | Esta expressão que é avaliada antes de cada repetição do laço. Quando seu resultado for VERDADEIRO, *<seqüência-de-comandos>* é executada. |
| fimenquanto | Indica o fim da *<seqüência-de-comandos>* que será repetida. Cada vez que a execução atinge este ponto, volta-se ao início do laço para que *<expressão-lógica>* seja avaliada novamente. Se o resultado desta avaliação for VERDADEIRO, a *<seqüência-de-comandos>* será executada mais uma vez; caso contrário, a execução prosseguirá a partir do primeiro comando após fimenquanto. |

O mesmo exemplo anterior pode ser resolvido com esta estrutura de repetição:

```
algoritmo "Números de 1 a 10 (com enquanto...faca)"
var j: inteiro
inicio
j <- 1
enquanto j <= 10 faca
   escreva (j:3)
   j <- j + 1
```

```
fimenquanto
fimalgoritmo
```

**Importante:** Como o laço `enquanto...faca` testa sua condição de parada **antes** de executar sua seqüência de comandos, esta seqüência poderá ser executada **zero ou mais vezes**.

**Repita ... até**

Esta estrutrura repete uma seqüência de comandos até que uma determinada condição (especificada através de uma expressão lógica) seja satisfeita.

```
repita
   <seqüência-de-comandos>
ate <expressão-lógica>
```

| | |
|---|---|
| `repita` | Indica o início do laço. |
| `ate <expressão-lógica>` | Indica o fim da `<seqüência-de-comandos>` a serem repetidos. Cada vez que o programa chega neste ponto, `<expressão-lógica>` é avaliada: se seu resultado for FALSO, os comandos presentes entre esta linha e a linha `repita` são executados; caso contrário, a execução prosseguirá a partir do primeiro comando após esta linha. |

Considerando ainda o mesmo exemplo:

```
algoritmo "Números de 1 a 10 (com repita)"
var j: inteiro
inicio
j <- 1
repita
   escreva (j:3)
   j <- j + 1
ate j > 10
fimalgoritmo
```

**Importante:** Como o laço `repita...ate` testa sua condição de parada **depois** de executar sua seqüência de comandos, esta seqüência poderá ser executada **uma ou mais vezes**.

**Comando Interrompa**

As três estruturas de repetição acima permitem o uso do comando `interrompa`, que causa uma saída imediata do laço. Embora esta técnica esteja de certa forma em desacordo com os princípios da programação estruturada, o comando `interrompa` foi incluído no VisuAlg por ser encontrado na literatura de introdução à programação e mesmo em linguagens como o Object Pascal (Delphi/Kylix), Clipper, VB, etc. Seu uso é exemplificado a seguir:

```
algoritmo "Números de 1 a 10 (com interrompa)"
var x: inteiro
inicio
x <- 0
repita
   x <- x + 1
   escreva (x:3)
   se x = 10 entao
      interrompa
   fimse
ate falso
fimalgoritmo
```

O VisuAlg permite ainda uma forma alternativa do comando `repita...ate`, com a seguinte sintaxe:

```
algoritmo "Números de 1 a 10 (com interrompa) II"
var x: inteiro
inicio
x <- 0
repita
   x <- x + 1
   escreva (x:3)
   se x = 10 entao
      interrompa
   fimse
fimrepita
fimalgoritmo
```

Com esta sintaxe alternativa, o uso do `interrompa` é obrigatório, pois é a única maneira de se sair do laço `repita...fimrepita`; caso contrário, este laço seria executado indeterminadamente.

*Subprograma* é um programa que auxilia o programa principal através da realização de uma determinada subtarefa. Também costuma receber os nomes de *sub-rotina*, *procedimento, método* ou *módulo*. Os subprogramas são chamados dentro do corpo do programa principal como se fossem *comandos*. Após seu término, a execução continua a partir do ponto onde foi chamado. É importante compreender que a chamada de um subprograma simplesmente gera um **desvio provisório no fluxo de execução**.

Há um caso particular de subprograma que recebe o nome de *função*. Uma *função*, além de executar uma determinada tarefa, retorna um valor para quem a chamou, que é o resultado da sua execução. Por este motivo, a chamada de uma função aparece no corpo do programa principal como uma *expressão*, e não como um comando.

Cada subprograma, além de ter acesso às variáveis do programa que o chamou (são as variáveis *globais*), pode ter suas próprias variáveis (são as variáveis *locais*), que existem apenas durante sua chamada.

Ao se chamar um subprograma, também é possível passar-lhe determinadas informações que recebem o nome de *parâmetros* (são valores que, na linha de chamada, ficam entre os parênteses e que estão separados por vírgulas). A quantidade dos parâmetros, sua seqüência e respectivos tipos não podem mudar: devem estar de acordo com o que foi especificado na sua correspondente declaração.

Para se criar subprogramas, é preciso descrevê-los após a declaração das variáveis e antes do corpo do programa principal. O VisuAlg possibilita declaração e chamada de subprogramas nos moldes da linguagem Pascal, ou seja, procedimentos e funções com passagem de parâmetros por valor ou referência. Isso será explicado a seguir.

**Procedimentos**

Em VisuAlg, procedimento é um subprograma que não retorna nenhum valor (corresponde ao *procedure* do Pascal). Sua declaração, que deve estar entre o final da declaração de variáveis e a linha `inicio` do programa principal, segue a sintaxe abaixo:

```
procedimento <nome-de-procedimento> [(<seqüência-de-declarações-de-parâmetros>)]
// Seção de Declarações Internas
inicio
// Seção de Comandos
fimprocedimento
```

O `<nome-de-procedimento>` obedece as mesmas regras de nomenclatura das variáveis. Por outro lado, a `<seqüência-de-declarações-de-parâmetros>` é uma seqüência de

```
[var] <seqüência-de-parâmetros>: <tipo-de-dado>
```

separadas por ponto e vírgula. A presença (opcional) da palavra-chave `var` indica passagem de parâmetros por referência; caso contrário, a passagem será por valor.

Por sua vez, *<seqüência-de-parâmetros>* é uma seqüência de nomes de parâmetros (também obedecem a mesma regra de nomenclatura de variáveis) separados por vírgulas.

De modo análogo ao programa principal, a seção de declaração internas começa com a palavra-chave `var`, e continua com a seguinte sintaxe:

```
<lista-de-variáveis> : <tipo-de-dado>
```

Nos próximos exemplos, através de um subprograma `soma`, será calculada a soma entre os valores 4 e –9 (ou seja, será obtido o resultado 13) que o programa principal imprimirá em seguida. No primeiro caso, um **procedimento sem parâmetros** utiliza uma variável local `aux` para armazenar provisoriamente o resultado deste cálculo (evidentemente, esta variável é desnecessária, mas está aí apenas para ilustrar o exemplo), antes de atribuí-lo à variável global `res`:

```
procedimento soma
var aux: inteiro
inicio
// n, m e res são variáveis globais
aux <- n + m
res <- aux
fimprocedimento
```

No programa principal deve haver os seguintes comandos:

```
n <- 4
m <- -9
soma
escreva(res)
```

A mesma tarefa poderia ser executada através de um **procedimento com parâmetros**, como descrito abaixo:

```
procedimento soma (x,y: inteiro)
inicio
// res é variável global
res <- x + y
fimprocedimento
```

No programa principal deve haver os seguintes comandos:

```
n <- 4
m <- -9
soma(n,m)
escreva(res)
```

A passagem de parâmetros do exemplo acima chama-se **passagem por valor**. Neste caso, o subprograma simplesmente recebe um valor que utiliza durante sua execução. Durante essa execução, os parâmetros passados por valor são análogos às suas variáveis locais, mas com uma única diferença: receberam um valor inicial no momento em que o subprograma foi chamado.

**Funções**

Em VisuAlg, função é um subprograma que retorna um valor (corresponde ao *function* do Pascal). De modo análogo aos procedimentos, sua declaração deve estar entre o final da declaração de variáveis e a linha `inicio` do programa principal, e segue a sintaxe abaixo:

```
funcao <nome-de-função> [(<seqüência-de-declarações-de-parâmetros>)]: <tipo-de-dado>
// Seção de Declarações Internas
inicio
// Seção de Comandos
fimfuncao
```

O *<nome-de-função>* obedece as mesmas regras de nomenclatura das variáveis. Por outro lado, a *<seqüência-de-declarações-de-parâmetros>* é uma seqüência de

`[var]` *<seqüência-de-parâmetros>*: *<tipo-de-dado>*

separadas por ponto e vírgula. A presença (opcional) da palavra-chave `var` indica passagem de parâmetros por referência; caso contrário, a passagem será por valor.

Por sua vez, *<seqüência-de-parâmetros>* é uma seqüência de nomes de parâmetros (também obedecem a mesma regra de nomenclatura de variáveis) separados por vírgulas.

O valor retornado pela função será do tipo especificado na sua declaração (logo após os dois pontos). Em alguma parte da função (de modo geral, no seu final), este valor deve ser retornado através do comando `retorne`.

De modo análogo ao programa principal, a seção de declaração internas começa com a palavra-chave `var`, e continua com a seguinte sintaxe:

*<lista-de-variáveis>* : *<tipo-de-dado>*

Voltando ao exemplo anterior, no qual calculamos e imprimimos a soma entre os valores 4 e –9, vamos mostrar como isso poderia ser feito através de uma **função sem parâmetros**. Ela também utiliza uma variável local `aux` para armazenar provisoriamente o resultado deste cálculo, antes de atribuí-lo à variável global `res`:

```
funcao soma: inteiro
var aux: inteiro
inicio
// n, m e res são variáveis globais
aux <- n + m
retorne aux
fimfuncao
```

No programa principal deve haver os seguintes comandos:

```
n <- 4
m <- -9
res <- soma
escreva(res)
```

Se realizássemos essa mesma tarefa com uma **função com parâmetros passados por valor**, poderia ser do seguinte modo:

```
funcao soma (x,y: inteiro): inteiro
inicio
retorne x + y
fimfuncao
```

No programa principal deve haver os seguintes comandos:

```
n <- 4
m <- -9
res <- soma(n,m)
escreva(res)
```

**Passagem de Parâmetros por Referência**

Há ainda uma outra forma de passagem de parâmetros para subprogramas: é a passagem por referência. Neste caso, o subprograma não recebe apenas um valor, mas sim o **endereço** de uma variável global. Portanto, qualquer modificação que for realizada no conteúdo deste parâmetro afetará também a variável global que está associada a

ele. Durante a execução do subprograma, os parâmetros passados por referência são análogos às variáveis globais. No VisuAlg, de forma análoga a Pascal, essa passagem é feita através da palavra `var` na declaração do parâmetro.

Voltando ao exemplo da soma, o procedimento abaixo realiza a mesma tarefa utilizando passagem de parâmetros por referência:

```
procedimento soma (x,y: inteiro; var result: inteiro)
inicio
result <- x + y
fimprocedimento
```

No programa principal deve haver os seguintes comandos:

```
n <- 4
m <- -9
soma(n,m,res)
escreva(res)
```

**Recursão e Aninhamento**

A atual versão do VisuAlg permite recursão, isto é, a possibilidade de que um subprograma possa chamar a si mesmo. A função do exemplo abaixo calcula recursivamente o fatorial do número inteiro que recebe como parâmetro:

```
funcao fatorial (v: inteiro): inteiro
inicio
se v <= 2 entao
retorne v
senao
retorne v * fatorial(v-1)
fimse
fimfuncao
```

Em Pascal, é permitido o aninhamento de subprogramas, isto é, cada subprograma também pode ter seus próprios subprogramas. No entanto, esta característica dificulta a elaboração dos compiladores e, na prática, não é muito importante. Por este motivo, ela não é permitida na maioria das linguagens de programação (como C, por exemplo), e o VisuAlg não a implementa.

O VisuAlg implementa algumas extensões às linguagens "tradicionais" de programação, com o intuito principal de ajudar o seu uso como ferramenta de ensino. Elas são mostradas a seguir.

**Comando Aleatório**

Muitas vezes a digitação de dados para o teste de um programa torna-se uma tarefa entediante. Com o uso do comando `aleatorio` do VisuAlg, sempre que um comando `leia` for encontrado, a digitação de valores numéricos e/ou caracteres é substituída por uma geração aleatória. Este comando não afeta a leitura de variáveis lógicas: com certeza, uma coisa pouco usual em programação...

Este comando tem as seguintes sintaxes:

| aleatorio [on] | Ativa a geração de valores aleatórios que substituem a digitação de dados. A palavra-chave on é opcional. A faixa padrão de valores gerados é de 0 a 100 inclusive. Para a geração de dados do tipo caractere, não há uma faixa pré-estabelecida: os dados gerados serão sempre *strings* de 5 letras maiúsculas. |
|---|---|
| aleatorio *<valor1 >* [, *<valor2 >* ] | Ativa a geração de dados numéricos aleatórios estabelecendo uma faixa de valores mínimos e máximos. Se apenas *< valor1>* for fornecido, a faixa será de 0 a *<valor1>* inclusive; caso contrário, a faixa será de *<valor1>* a *<valor2>* inclusive. Se *<valor2>* for menor que *<valor1>*, o VisuAlg |

| | os trocará para que a faixa fique correta.<br>**Importante:** *<valor1>* e *<valor2>* devem ser **constantes numéricas**, e não expressões. |
|---|---|
| `aleatorio off` | Desativa a geração de valores aleatórios. A palavra-chave `off` é obrigatória. |

### Comando Arquivo

Muitas vezes é necessário repetir os testes de um programa com uma série igual de dados. Para casos como este, o VisuAlg permite o armazenamento de dados em um arquivo-texto, obtendo deles os dados ao executar os comandos `leia`.
Esta característica funciona da seguinte maneira:
1) Se **não existir** o arquivo com nome especificado, o VisuAlg fará uma leitura de dados através da digitação, armazenando os dados lidos neste arquivo, na ordem em que forem fornecidos.
2) Se o arquivo **existir**, o VisuAlg obterá os dados deste arquivo até chegar ao seu fim. Daí em diante, fará as leituras de dados através da digitação.
3) Somente um comando `arquivo` pode ser empregado em cada pseudocódigo, e ele deverá estar na seção de declarações (dependendo do "sucesso" desta característica, em futuras versões ela poderá ser melhorada...).
4) Caso não seja fornecido um caminho, o VisuAlg irá procurar este arquivo na pasta de trabalho corrente (geralmente, é a pasta onde o programa VISUALG.EXE está). Este comando não prevê uma extensão padrão; portanto, a especificação do nome do arquivo deve ser completa, inclusive com sua extensão (por exemplo, .txt, .dat, etc.).

A sintaxe do comando é:

```
arquivo <nome-de-arquivo>
```

*<nome-de-arquivo>* é uma constante caractere (entre aspas duplas). Veja o exemplo a seguir:

```
algoritmo "lendo do arquivo"
arquivo "teste.txt"
var x,y: inteiro
inicio
para x de 1 ate 5 faca
   leia (y)
fimpara
fimalgoritmo
```

### Comando Timer

Embora o VisuAlg seja um interpretador de pseudocódigo, seu desempenho é muito bom: o tempo gasto para interpretar cada linha digitada é apenas uma fração de segundo. Entretanto, por motivos educacionais, pode ser conveniente exibir o fluxo de execução do pseudocódigo comando por comando, em "câmera lenta". O comando `timer` serve para este propósito: insere um atraso (que pode ser especificado) antes da execução de cada linha. Além disso, realça em fundo azul o comando que está sendo executado, da mesma forma que na execução passo a passo.

Sua sintaxe é a seguinte:

| `timer on` | Ativa o *timer*. |
|---|---|
| `timer <tempo-de-atraso>` | Ativa o *timer* estabelecendo seu tempo de atraso em milissegundos. O valor padrão é 500, que equivale a meio segundo. O argumento *<tempo-de-atraso>* deve ser uma constante inteira com valor entre 0 e 10000. Valores menores que 0 são corrigidos para 0, e maiores que 10000 para 10000. |
| `timer off` | Desativa o *timer*. |

Ao longo do pseudocódigo, pode haver vários comandos `timer`. Todos eles devem estar na seção de comandos. Uma vez ativado, o atraso na execução dos comandos será mantido até se chegar ao final do pseudocódigo ou até ser encontrado um comando `timer off`.

**Comandos de Depuração**

Nenhum ambiente de desenvolvimento está completo se não houver a possibilidade de se inserir pontos de interrupção (*breakpoints*) no pseudocódigo para fins de depuração. VisuAlg implementa dois comandos que auxiliam a depuração ou análise de um pseudocódigo: o comando `pausa` e o comando `debug`.

**Comando Pausa**

Sua sintaxe é simplesmente:

```
pausa
```

Este comando insere uma interrupção incondicional no pseudocódigo. Quando ele é encontrado, o VisuAlg pára a execução do pseudocódigo e espera alguma ação do programador. Neste momento, é possível: analisar os valores das variáveis ou das saídas produzidas até o momento; executar o pseudocódigo passo a passo (com F8); prosseguir sua execução normalmente (com F9); ou simplesmente terminá-lo (com Ctrl-F2). Com exceção da alteração do texto do pseudocódigo, todas as funções do VisuAlg estão disponíveis.

**Comando Debug**

Sua sintaxe é:

```
debug <expressão-lógica>
```

Se a avaliação de *<expressão-lógica>* resultar em valor VERDADEIRO, a execução do pseudocódigo será interrompida como no comando `pausa`. Dessa forma, é possível a inserção de um *breakpoint* condicional no pseudocódigo.

**Comando Eco**

Sua sintaxe é:

```
eco on | off
```

Este comando ativa (`eco on`) ou desativa (`eco off`) a impressão dos dados de entrada na saída-padrão do VisuAlg, ou seja, na área à direita da parte inferior da tela. Esta característica pode ser útil quando houver uma grande quantidade de dados de entrada, e se deseja apenas analisar a saída produzida. Convém utilizá-la também quando os dados de entrada provêm de um arquivo já conhecido.

**Comando Cronômetro**

Sua sintaxe é:

```
cronometro on | off
```

Este comando ativa (`cronometro on`) ou desativa (`cronometro off`) o cronômetro interno do VisuAlg. Quando o comando `cronometro on` é encontrado, o VisuAlg imprime na saída-padrão a informação "Cronômetro iniciado.", e começa a contar o tempo em milissegundos. Quando o comando `cronometro off` é encontrado, o VisuAlg imprime na saída-padrão a informação "Cronômetro terminado. Tempo decorrido: xx segundo(s) e xx ms". Este comando é útil na análise de desempenho de algoritmos (ordenação, busca, etc.).

**Comando Limpatela**

Sua sintaxe é

```
limpatela
```

Este comando simplesmente limpa a tela DOS do Visualg (a simulação da tela do computador). Ele não afeta a "tela" que existe na parte inferior direita da janela principal do Visualg.