INSTITUTO DE ESTUDOS SUPERIORES DA AMAZÔNIA

Algoritmos e Programação

Sistemas de Informação

Profa. Lenilda Pinheiro
Lenilda@blm.serpro.gov.br

I Semestre/2000

SUMÁRIO

PARTE I - INTRODUÇÃO

1. ABORDAGEM CONTEXTUAL

O uso de algoritmos é quase tão antigo quanto a matemática. Com o passar do tempo, entretanto, ele foi bastante esquecido pela matemática. Com o advento das máquinas de calcular e mais tarde os computadores, o uso de algoritmos ressurgiu com grande vigor, como uma forma de indicar o caminho para a solução dos mais variados problemas.

Algoritmo não é a solução do problema, pois, se assim fosse, cada problema teria um único algoritmo. Algoritmo é o caminho para a solução de um problema, e em geral, os caminhos que levam a uma solução são muitos.

Ao longo dos anos surgiram muitas formas de representar os algoritmos, alguns utilizando linguagens semelhantes às linguagens de programação e outras utilizando formas gráficas.

O aprendizado de algoritmos não se consegue a não ser através de muitos exercícios.

Algoritmos não se aprendem:
– Copiando algoritmos
– Estudando algoritmos
Algoritmos só se aprendem:
– Construindo algoritmos
– Testando algoritmos

1.1 Conceito de Algoritmo

A automação é o processo em que uma tarefa deixa de ser desempenhada pelo homem e passa a ser realizada por máquinas, sejam estas dispositivos mecânicos (como as máquinas industriais), eletrônicos (como os computadores), ou de natureza mista (como os robôs).

Para que a automação de uma tarefa seja bem-sucedida é necessário que a máquina que passará a realizá-la seja capaz de desempenhar cada uma das etapas constituintes do processo a ser automatizado com eficiência, de modo a garantir a repetibilidade do mesmo. Assim, é necessário que seja especificado com clareza e exatidão o que deve ser realizado em cada uma das fases do processo a ser automatizado, bem como a seqüência em que estas fases devem ser realizadas.

À especificação da seqüência ordenada de passos que deve ser seguida para a realização de um tarefa, garantindo a sua repetibilidade, dá-se o nome de **algoritmo**.

Embora esta definição de algoritmo seja correta, podemos definir algoritmo, de maneira informal e completa como:

"**Algoritmo é um conjunto finito de regras, bem definidas, para a solução de um problema em um tempo finito e com um número finito de passos**."

Informalmente, um algoritmo é qualquer procedimento computacional bem definido que toma algum valor ou conjunto de valores como entrada e produz algum valor ou conjunto de valores como saída.

Um algoritmo deve sempre possuir pelo menos um resultado, normalmente chamado de **saída**, e satisfazer a propriedade da efetividade, isto é, todas as operações especificadas no algoritmo devem ser suficientemente básicas para que possam ser executadas de maneira exata e num tempo finito.

Na prática não é importante ter-se apenas um algoritmo, mas sim, um bom algoritmo. O mais importante de um algoritmo é a sua **correção**, isto é, se ele resolve realmente o problema proposto e o faz exatamente.

Para se ter um algoritmo, é necessário:

1. Que se tenha um número finito de passos
2. Que cada passo esteja precisamente definido, sem possíveis ambigüidades
3. Que existam zero ou mais entradas tomadas de conjuntos bem definidos
4. Que existam uma ou mais saídas
5. Que exista uma condição de fim sempre atingida para quaisquer entradas e num tempo finito.

Para que um computador possa desempenhar uma tarefa é necessário que esta seja detalhada passo a passo, numa forma compreensível pela máquina, utilizando aquilo que se chama de **programa**. Neste sentido, um programa de computador nada mais é que um algoritmo escrito numa forma compreensível pelo computador.

2. FORMAS DE REPRESENTAÇÃO DE ALGORITMOS

Existem diversas formas de representação de algoritmos, mas não há um consenso com relação à melhor delas.

Algumas formas de representação de algoritmos tratam dos problemas apenas em nível lógico, abstraindo-se de detalhes de implementação muitas vezes relacionados com alguma linguagem de programação específica. Por outro lado, existem formas de representação de algoritmos que possuem uma maior riqueza de detalhes e muitas vezes acabam por obscurecer a idéia principal, o algoritmo, dificultando seu entendimento.

Dentre as formas de representação de algoritmos mais conhecidas, sobressaltam:

- a **Descrição Narrativa**
- o **Fluxograma Convencional**
- o **Diagrama de Chapin**
- o **Pseudocódigo**, também conhecido como **Linguagem Estruturada** ou **Portugol**.

2.1 Descrição Narrativa

Nesta forma de representação os algoritmos são expressos diretamente em **linguagem natural**. Como por exemplo, têm-se os algoritmos seguintes:

– Troca de um pneu furado:

– Afrouxar ligeiramente as porcas
– Suspender o carro
– Retirar as porcas e o pneu
– Colocar o pneu reserva
– Apertar as porcas
– Abaixar o carro
– Dar o aperto final nas porcas

– Cálculo da média de um aluno:

– Obter as notas da primeira e da segunda prova
– Calcular a média aritmética entre as duas
– Se a média for maior ou igual a 7, o aluno foi aprovado, senão ele foi reprovado

Esta representação é pouco usada na prática porque o uso de linguagem natural muitas vezes dá oportunidade a más interpretações, ambigüidades e imprecisões.

Por exemplo, a instrução “afrouxar ligeiramente as porcas” no algoritmo da troca de pneus está sujeita a interpretações diferentes por pessoas distintas. Uma instrução mais precisa seria: “afrouxar a porca, girando-a de 30º no sentido anti-horário”.

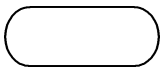
2.2 Fluxograma Convencional e Diagrama de Blocos

É uma representação gráfica de algoritmos onde formas geométricas diferentes implicam ações (instruções, comandos) distintos. Tal propriedade facilita o entendimento das idéias contidas nos algoritmos.

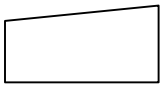
Nota-se que os fluxogramas convencionais preocupam-se com detalhes de nível físico da implementação do algoritmo. Por exemplo, figuras geométricas diferentes são adotadas para representar operações de saída de dados realizadas em dispositivos distintos, como uma unidade de armazenamento de dados ou um monitor de vídeo. A figura 2.1 mostra as principais formas geométricas usadas em fluxogramas.

De modo geral, o fluxograma se resume a um único símbolo inicial, por onde a execução do algoritmo começa, e um ou mais símbolos finais, que são pontos onde a execução do algoritmo se encerra. Partindo do símbolo inicial, há sempre um único caminho orientado a ser seguido, representando a existência de uma única seqüência de execução das instruções. Isto pode ser melhor visualizado pelo fato de que, apesar de vários caminhos poderem convergir para uma mesma figura do diagrama, há sempre um único caminho saindo desta. Exceções a esta regra são os símbolos finais, dos quais não há nenhum fluxo saindo, e os símbolos de decisão, de onde pode haver mais de um caminho de saída (normalmente dois caminhos), representando uma bifurcação no fluxo.

Um diagrama de blocos é uma forma de fluxograma usada e desenvolvida por profissionais da programação, tendo como objetivo descrever o método e a seqüência do processo dos planos num computador. Pode ser desenvolvido em qualquer nível de detalhe que seja necessário. Quando se desenvolve um diagrama para o programa principal, por exemplo, seu nível de detalhamento pode chegar até as instruções. Esta ferramenta usa diversos símbolos geométricos, os quais, estabelecerão as seqüências de operações a serem efetuadas em um processamento computacional. Após a elaboração do diagrama de bloco, é realizada a codificação do programa. A figura 2.1 mostra o exemplo de um diagrama de blocos ou fluxogramas.



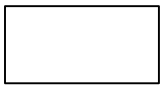
= Início e final do fluxograma



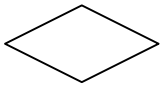
= Operação de entrada de dados

= Operação de saída de dados em impressora

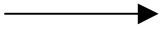
= Operação de saída de dados em vídeo



= Operações de atribuição e chamada ou retorno de subalgoritmo



= Decisão



= Seta do Fluxo de Dados

= Conector utilizado quando é preciso particionar o diagrama, colocando uma letra ou número no símbolo para identificar os pares da conexão

= Conector específico para indicar conexão do fluxo de execução em outra página

= Preparação - operações que utilizam uma chave que modifica a execução do programa

Figura 2.1 Principais formas geométricas usadas em fluxogramas

A figura 2.2 a seguir mostra a representação do algoritmo de cálculo da média de um aluno sob a forma de um fluxograma.

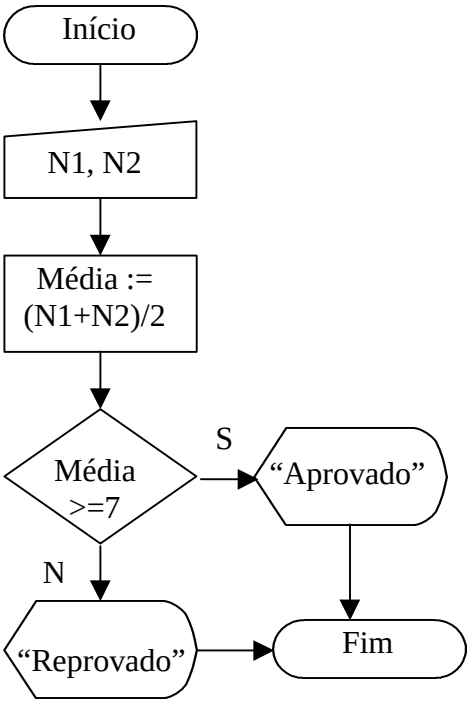


Figura 2.2 Exemplo de um fluxograma convencional

2.3 Diagrama de Chapin

O diagrama foi criado por Ned Chapin a partir de trabalhos de Nassi-Shneiderman, os quais resolveram substituir o fluxograma tradicional por um diagrama que apresenta uma visão hierárquica e estruturada da lógica do programa. A grande vantagem de usar este tipo de diagrama é a representação das estruturas que tem um ponto de entrada e um ponto de saída e são compostas pelas estruturas básicas de controle de seqüência, seleção e repartição. Enquanto é difícil mostrar o embutimento e a recursividade com o fluxograma tradicional, torna-se mais simples mostrá-lo com o diagrama de Chapin, bem como codificá-lo futuramente na conversão de código português estruturado ou pseudocódigos. A figura 2.3 apresenta um exemplo do tipo de diagrama de Chapin para o algoritmo de cálculo da média de um aluno.

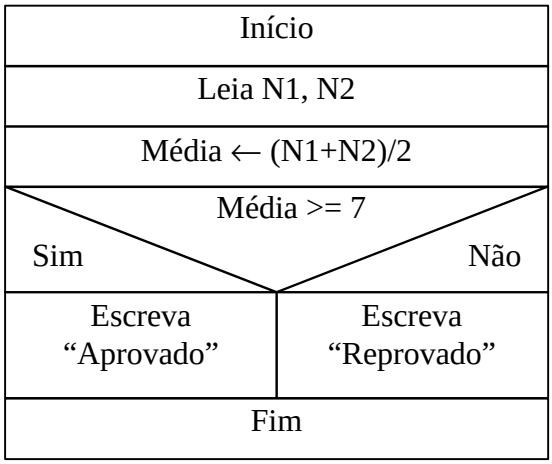


Figura 2.3 Diagrama de Chapin para o algoritmo do cálculo da média de um aluno

2.4 Pseudocódigo

Esta forma de representação de algoritmos, também conhecida como português estruturado ou portugol, é bastante rica em detalhes e, por assemelhar-se bastante à forma em que os programas são escritos, encontra muita aceitação, sendo portanto a forma de representação de algoritmos que será adotada nesta disciplina.

Na verdade, esta representação é suficientemente geral para permitir que a tradução de um algoritmo nela representado para uma linguagem de programação específica seja praticamente direta.

2.4.1 Representação de Um Algoritmo na Forma de Pseudocódigo

A representação de um algoritmo na forma de pseudocódigo é a seguinte:

```
Algoritmo <nome_do_algoritmo>
<declaração_de_variáveis>
<subalgoritmos>
Início
     <corpo_do_algoritmo>
Fim.
```

onde:

Algoritmo é uma palavra que indica o início da definição de um algoritmo em forma de pseudocódigo.

<***nome_do_algoritmo***> é um nome simbólico dado ao algoritmo com a finalidade de distinguí-lo dos demais.

<***declaração_de_variáveis***> consiste em uma porção opcional onde são declaradas as variáveis globais usadas no algoritmo principal e, eventualmente, nos subalgoritmos.

<***subalgoritmos***> consiste de uma porção opcional do pseudocódigo onde são definidos os subalgoritmos.

Início e Fim são respectivamente as palavras que delimitam o início e o término do conjunto de instruções do corpo do algoritmo.

Como exemplo, a seguir é mostrado a representação do algoritmo de cálculo da média de um aluno na forma de um pseudocódigo.

```
Algoritmo Média
Var N1, N2, Média
Início
      Leia N1, N2
      Média := (N1+N2)/2
      Se Média >= 7 Então
            Escreva “Aprovado”
      Senão
            Escreva “Reprovado”
Fim.
```

PARTE II - TÉCNICAS BÁSICAS DE PROGRAMAÇÃO

3. TIPOS DE DADOS

Todo o trabalho realizado por um computador é baseado na manipulação das informações contidas em sua memória. Estas informações podem ser classificadas em dois tipos:

– As **instruções**, que comandam o funcionamento da máquina e determinam a maneira como devem ser tratados os dados.

– Os **dados** propriamente ditos, que correspondem à porção das informações a serem processadas pelo computador.

A classificação apresentada a seguir não se aplica a nenhuma linguagem de programação específica; pelo contrário, ela sintetiza os padrões utilizados na maioria das linguagens.

3.1 Tipos Inteiros

São caracterizados como tipos inteiros, os dados numéricos positivos ou negativos. Excluindo-se destes qualquer número fracionário. Como exemplo deste tipo de dado, tem-se os valores: 35, 0, -56, 1024 entre outros.

3.2 Tipos Reais

São caracterizados como tipos reais, os dados numéricos positivos e negativos e números fracionários. Como exemplo deste tipo de dado, tem-se os valores: 35, 0, -56, 1.2, -45.987 entre outros.

3.3 Tipos Caracteres

São caracterizados como tipos caracteres, as seqüências contendo letras, números e símbolos especiais. Uma seqüência de caracteres deve ser indicada entre aspas (“”). Este tipo de dado também é conhecido como alfanumérico, string, literal ou cadeia. Como exemplo deste tipo de dado, tem-se os valores: “Programação”, “Rua Alfa, 52 Apto 1”, “Fone 574-9988”, “04387-030”, “ ”, “7” entre outros.

3.4 Tipos Lógicos

São caracterizados como tipos lógicos os dados com valor **verdadeiro** e **falso**, sendo que este tipo de dado poderá representar apenas um dos dois valores. Ele é chamado por alguns de **tipo booleano**, devido à contribuição do filósofo e matemático inglês George Boole na área da lógica matemática.

4. VARIÁVEIS E CONSTANTES

4.1 Armazenamento de Dados na Memória

Para armazenar os dados na memória, imagine que a memória de um computador é um grande arquivo com várias gavetas, onde cada gaveta pode armazenar apenas um único valor (seja ele numérico, caractere ou lógico). Se é um grande arquivo com várias gavetas, é necessário identificar com um nome a gaveta que se pretende utilizar. Desta forma o valor armazenado pode ser utilizado a qualquer momento.

4.2 Conceito e Utilidade de Variáveis

Têm-se como definição de variável tudo aquilo que é sujeito a variações, que é incerto, instável ou inconstante. E quando se fala de computadores, temos que ter em mente que o volume de informações a serem tratadas é grande e diversificado. Desta forma, os dados a serem processados serão bastante variáveis.

Como visto anteriormente, informações correspondentes a diversos tipos de dados são armazenadas nas memórias dos computadores. Para acessar individualmente cada uma destas informações, em princípio, seria necessário saber o tipo de dado desta informação (ou seja, o número de bytes de memória por ela ocupados) e a posição inicial deste conjunto de bytes na memória.

Percebe-se que esta sistemática de acesso a informações na memória é bastante ilegível e difícil de se trabalhar. Para contornar esta situação criou-se o conceito de **variável**, que é uma entidade destinada a guardar uma informação.

Basicamente, uma variável possui três atributos: um **nome**, um **tipo de dado** associado à mesma e a **informação** por ela guardada.

Toda variável possui um **nome** que tem a função de diferenciá-la das demais. Cada linguagem de programação estabelece suas próprias regras de formação de nomes de variáveis.

Adotaremos para os algoritmos, as seguintes regras:

- um nome de variável deve necessariamente começar com uma letra;
- um nome de variável não deve conter nenhum símbolo especial, exceto a sublinha (_) e nenhum espaço em branco;
- um nome de variável não poderá ser uma palavra reservada a uma instrução de programa.

Exemplos de nomes de variáveis:

Salário	– correto
1ANO	– errado (não começou uma letra)

ANO1	– correto
a casa	– errado (contém o caractere branco)
SAL/HORA	– errado (contém o caractere “/”)
SAL_HORA	– correto
_DESCONTO	– errado (não começou com uma letra)

Obviamente é interessante adotar nomes de variáveis relacionados às funções que serão exercidas pela mesmas dentro de um programa.

Outro atributo característico de uma variável é o **tipo de dado** que ela pode armazenar. Este atributo define a natureza das informações contidas na variável. Por último há o atributo **informação**, que nada mais é do que a informação útil contida na variável.

Uma vez definidos, os atributos **nome** e **tipo de dado** de uma variável **não** podem ser alterados e assim permanecem durante toda a sua existência, desde que o programa que a utiliza não seja modificado. Por outro lado, o atributo **informação** está constantemente sujeito a mudanças de acordo com o fluxo de execução do programa.

Em resumo, o conceito de variável foi criado para facilitar a vida dos programadores, permitindo acessar informações na memória dos computadores por meio de um nome, em vez do endereço de uma célula de memória.

4.3 Definição de Variáveis em Algoritmos

Todas as variáveis utilizadas em algoritmos devem ser definidas antes de serem utilizadas. Isto se faz necessário para permitir que o compilador reserve um espaço na memória para as mesmas.

Mesmo que algumas linguagens de programação (como BASIC e FORTRAN) dispensam esta definição, uma vez que o espaço na memória é reservado à medida que novas variáveis são encontradas no decorrer do programa, nos algoritmos usaremos a definição de variáveis por assemelhar-se com as principais linguagens de programação como Pascal e C.

Nos algoritmos, todas as variáveis utilizadas serão definidas no início do mesmo, por meio de um comando de uma das seguintes formas:

VAR <*nome_da_variável*> : <*tipo_da_variável*>
ou
VAR <*lista_de_variáveis*> : <*tipo_das_variáveis*>

- a palavra-chave **VAR** deverá estar presente sempre e será utilizada um única vez na definição de um conjunto de uma ou mais variáveis;

- numa mesma linha poderão ser definidas uma ou mais variáveis do mesmo tipo; Para tal, deve-se separar os nomes das mesmas por vírgulas;

- variáveis de tipos diferentes devem ser declaradas em linhas diferentes.

Exemplos de definição de variáveis:

VAR nome: caracter[30]
idade: inteiro
salário: real
tem_filhos: lógico

No exemplo acima foram declaradas quatro variáveis:

- a variável **nome**, capaz de armazenar dados caractere de comprimento 35 (35 caracteres);
- a variável **idade**, capaz de armazenar um número inteiro;
- a variável **salário**, capaz de armazenar um número real;
- a variável **tem_filhos**, capaz de armazenar uma informação lógica.

4.4 Conceito e Utilidade de Constantes

Têm-se como definição de constante tudo aquilo que é fixo ou estável. Existirão vários momentos em que este conceito deverá estar em uso, quando desenvolvermos programas.

É comum definirmos uma constante no início do programa, e a utilizarmos no decorrer do programa, para facilitar o entendimento, a programação ou então para poupar tempo no caso de ter que alterar o seu valor, de modo que alterando uma única vez a declaração da constante, todos os comandos e expressões que a utilizam são automaticamente atualizados.

4.5 Definição de Constantes em Algoritmos

Nos algoritmos, todas as constante utilizadas serão definidas no início do mesmo, por meio de um comando da seguinte forma:

CONST *<nome_da_constante> = <valor>*

Exemplo de definição de constantes:

CONST pi = 3.14159
nome_da_empresa = “Enxuga Gelo SA”

5. EXPRESSÕES E OPERADORES

5.1 Operadores

Operadores são elementos fundamentais que atuam sobre **operandos** e produzem um determinado resultado. Por exemplo, a expressão **3 + 2** relaciona dois operandos (os números 3 e 2) por meio do operador (+) que representa a operação de adição.

De acordo com o número de operandos sobre os quais os operadores atuam, os últimos podem ser classificados em:

- **binários**, quando atuam sobre dois operandos. Esta operação é chamada diádica. Ex.: os operadores das operações aritméticas básicas (soma, subtração, multiplicação e divisão).

- **unários**, quando atuam sobre um único operando. Esta operação é chamada monádica. Ex.: o sinal de (-) na frente de um número, cuja função é inverter seu sinal.

Outra classificação dos operadores é feita considerando-se o tipo de dado de seus operandos e do valor resultante de sua avaliação. Segundo esta classificação, os operandos dividem-se em **aritméticos**, **lógicos** e **literais**. Esta divisão está diretamente relacionada com o tipo de expressão onde aparecem os operadores.

Um caso especial é o dos operadores **relacionais**, que permitem comparar pares de operandos de tipos de dados iguais, resultando sempre num valor lógico.

5.1.1 Operadores de Atribuição

Um operador de atribuição serve para atribuir um valor a uma variável.

Em Algoritmo usamos o operador de atribuição:

:=

A sintaxe de um comando de atribuição é:

NomedaVariável := expressão

A expressão localizada no lado direito do sinal de igual é avaliada e armazenado o valor resultante na variável à esquerda. O nome da variável aparece sempre sozinho, no lado esquerdo do sinal de igual deste comando.

5.1.2 Operadores Aritméticos

Os operadores aritméticos se relacionam às operações aritméticas básicas, conforme a tabela abaixo:

Operador	Tipo	Operação	Prioridade
+	Binário	Adição	4
-	Binário	Subtração	4
*	Binário	Multiplicação	3
/	Binário	Divisão	3
MOD	Binário	Resto da Divisão	3
DIV	Binário	Divisão Inteira	3
**	Binário	Exponenciação	2
+	Unário	Manutenção do Sinal	1
-	Unário	Inversão do Sinal	1

A prioridade entre operadores define a ordem em que os mesmos devem ser avaliados dentro de uma mesma expressão.

5.1.3 Operadores Relacionais

Os operadores relacionais são operadores binários que devolvem os valores lógicos verdadeiro e falso.

Operador	Comparação
>	maior que
<	menor que
>=	maior ou igual
<=	menor ou igual
=	igual
<>	diferente

Estes valores são somente usados quando se deseja efetuar comparações. Comparações só podem ser feitas entre objetos de mesma natureza, isto é variáveis do mesmo tipo de dado. O resultado de uma comparação é sempre um valor lógico

Por exemplo, digamos que a variável inteira *escolha* contenha o valor 7. A primeira das expressões a seguir fornece um valor falso, e a segunda um valor verdadeiro:

```
escolha <= 5
escolha > 5
```

Com valores string, os operadores relacionais comparam os valores ASCII dos caracteres correspondentes em cada string. Uma string é dita "menor que" outra se os caracteres correspondentes tiverem os números de códigos ASCII menores. Por exemplo, todas as expressões a seguir são verdadeiras:

“algoritmo” > “ALGORITMO”
“ABC” < “EFG”
“Pascal” < “Pascal compiler”

Observe que as letras minúsculas têm códigos ASCII maiores do que os das letras maiúsculas. Observe também que o comprimento da string se torna o fator determinante na comparação de duas strings, quando os caracteres existentes na string menor são os mesmos que os caracteres correspondentes na string maior. Neste caso, a string maior é dita “maior que” a menor.

5.1.4 Operadores Lógicos

Os operadores lógicos ou booleanos são usados para combinar expressões relacionais. Também devolvem como resultado valores lógicos verdadeiro ou falso.

Operador	Tipo	Operação	Prioridade
OU	Binário	Disjunção	3
E	Binário	Conjunção	2
NÃO	Unário	Negação	1

Uma expressão relacional ou lógica retornará **falso** para o valor lógico **falso** e **verdadeiro** para o valor lógico **verdade**.

Fornecendo dois valores ou expressões lógicas, representadas por ***expressão1*** *e* ***expressão2***, podemos descrever as quatro operações lógicas a seguir:

expressão1 E ***expressão2*** *é* verdadeiro somente se ambas, ***expressão1*** e ***expressão2***, forem verdadeiras. Se uma for falsa, ou se ambas forem falsas, a operação E também será falsa.

expressão1 *OU* ***expressão2*** é verdadeiro se tanto a ***expressão1*** como a ***expressão2*** forem verdadeiras. As operações OU só resultam em valores falsos se ambas, ***expressão1*** e ***expressão2***, forem falsas.

NÃO ***expressão1*** avalia verdadeiro se ***expressão1*** for falsa; de modo contrário, a expressão NÃO resultará em falso, se ***expressão1*** for verdadeira.

5.1.5 Operadores Literais

Os operadores que atuam sobre caracteres variam muito de uma linguagem para outra. O operador mais comum e mais usado é o operador que faz a concatenação de strings: toma-se duas strings e acrescenta-se (concatena-se) a segunda ao final da primeira.

O operador que faz esta operação é: **+**

Por exemplo, a concatenação das strings “ALGO” e “RITMO” é representada por:

“ALGO” + ”RITMO”

e o resultado de sua avaliação é: “ALGORITMO”

5.2 Expressões

O conceito de expressão em termos computacionais está intimamente ligado ao conceito de expressão ou fórmula matemática, onde um conjunto de variáveis e constantes numéricas relacionam-se por meio de operadores aritméticos compondo uma fórmula que, uma vez avaliada, resulta num valor.

5.2.1 Expressões Aritméticas

Expressões aritméticas são aquelas cujo resultado da avaliação é do tipo numérico, seja ele inteiro ou real. Somente o uso de operadores aritméticos, variáveis numéricas e parênteses é permitido em expressões deste tipo

5.2.2 Expressões Lógicas

Expressões lógicas são aquelas cujo resultado da avaliação é um valor lógico verdadeiro ou falso.

Nestas expressões são usados os operadores relacionais e os operadores lógicos, podendo ainda serem combinados com expressões aritméticas.

Quando forem combinadas duas ou mais expressões que utilizem operadores relacionais e lógicos, os mesmos devem utilizar os parênteses para indicar a ordem de precedência.

5.2.3 Expressões Literais

Expressões literais são aquelas cujo resultado da avaliação é um valor literal (caractere). Neste tipo de expressões só é usado o operador de literais (+).

5.2.4 Avaliação de Expressões

Expressões que apresentam apenas um único operador podem ser avaliadas diretamente. No entanto, à medida que as mesmas vão tornando-se mais complexas com o aparecimento de mais de um operando na mesma expressão, é necessária a avaliação da mesma passo a passo, tomando um operador por vez. A seqüência destes passos é definida de acordo com o formato geral da expressão, considerando-se a prioridade (precedência) de avaliação de seus operadores e a existência ou não de parênteses na mesma.

As seguintes regras são essenciais para a correta avaliação de expressões:

1. Deve-se observar a prioridade dos operadores, conforme mostrado nas tabelas de operadores: operadores de maior prioridade devem ser avaliados primeiro. Se houver empate com relação à precedência, então a avaliação se faz da esquerda para a direita.

2. Os parênteses usado em expressões tem o poder de “roubar” prioridade dos demais operadores, forçando a avaliação da subexpressão em seu interior.

3. Entre os quatro grupos de operadores existentes, a saber, aritmético, lógico, literal e relacional, há uma certa prioridade de avaliação: os aritméticos e literais devem ser avaliados primeiro; a seguir, são avaliadas as subexpressões com operadores relacionais e, por último os operadores lógicos são avaliados.

Exercícios

1. Dados as variáveis e operações:
   ```
   v1 := 32
   v2 := 5 + v1
   v1 := v2 * 2
   ```
   Como fazer para segurar e mostrar o valor inicial da variável v1 no final das operações?

2. Como fazer para passar o valor de uma variável para outra e vice-versa?

3. Quais as operações necessárias para intercambiar os valores de 3 variáveis **a**, **b** e **c** de modo que **a** fique com o valor de **b**; **b** fique com o valor de **c** e **c** fique com o valor de **a**?

4. Se **X** possui o valor 15 e foram executadas as seguintes instruções:
   ```
   X := X + 3
   X := X - 6
   X := X / 2
   X := 3 * X
   ```
   Qual será o valor armazenado em **X**?

6. INSTRUÇÕES PRIMITIVAS

Como o próprio nome diz, **instruções primitivas** são os comandos básicos que efetuam tarefas essenciais para a operação dos computadores, como entrada e saída de dados (comunicação com o usuário e com dispositivos periféricos), e movimentação dos mesmos na memória. Estes tipos de instrução estão presentes na absoluta maioria das linguagens de programação.

Antes de passar à descrição das instruções primitiva, é necessária a definição de alguns termos que serão utilizados:

- **dispositivo de entrada** é o meio pelo qual as informações (mais especificamente os dados) são transferidos pelo usuário ou pelos níveis secundários de memória ao computador. Os exemplos mais comuns são o teclado, o mouse, leitora ótica, leitora de código de barras, as fitas e discos magnéticos.

- **dispositivo de saída** é o meio pelo qual as informações (geralmente os resultados da execução de um programa) são transferidos pelo computador ao usuário ou aos níveis secundários de memória. Os exemplos mais comuns são o monitor de vídeo, impressora, fitas e discos magnéticos.

- **sintaxe** é a forma como os comandos devem ser escritos, a fim de que possam ser entendidos pelo tradutor de programas. A violação das regras sintáticas é considerada um erro sujeito à pena do não reconhecimento por parte do tradutor

- **semântica** é o significado, ou seja, o conjunto de ações que serão exercidas pelo computador durante a execução do referido comando.

Daqui em diante, todos os comando novos serão apresentados por meio de sua sintaxe e sua semântica, isto é, a forma como devem ser escritos e a(s) ação(ões) que executam.

6.1 Comandos de Atribuição

O **comando de atribuição** ou simplesmente **atribuição**, é a principal maneira de armazenar uma informação numa variável. Sua sintaxe é:

<nome_da_variável> **:=** ***<expressão>***

Ex:
```
Nome := “Jenoveva”
preco := 15.85
quant := 5
total : preco * quant
imposto := total * 17 / 100
```

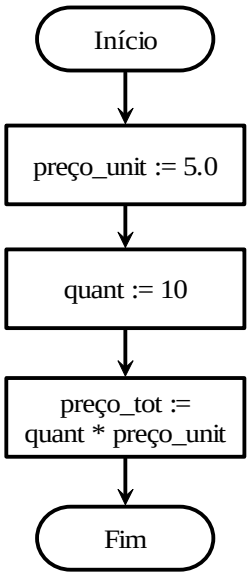
O modo de funcionamento (semântica) de uma atribuição consiste:

1) na avaliação da expressão

2) no armazenamento do valor resultante na variável que aparece à esquerda do comando.

A seguir temos um exemplo de um algoritmo utilizando o comando de atribuição:

```
Algoritmo exemplo_comando_de_atriuição
Var  preço_unit, preço_tot : real
     quant : inteiro
Início
     preço_unit := 5.0
     quant := 10
     preço_tot := preço_unit * quant
Fim.
```

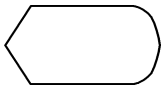


6.2 Comandos de Saída de Dados

Os **comandos de saída de dados** são o meio pelo qual informações contidas na memória dos computadores são colocadas nos dispositivos de saída, para que os usuários possam apreciá-las.

No diagrama de blocos o comando de saída de dados é representado por:

= Operação de saída de dados em impressora



= Operação de saída de dados em vídeo

Há quatro sintaxes possíveis para esta instrução:

- **ESCREVA** ***<variável>***
  Ex: **ESCREVA** X

- **ESCREVA** ***<lista_de_variáveis>***
  Ex: **ESCREVA** nome, endereco, cidade

- **ESCREVA** ***<literal>***
  Ex: **ESCREVA** "Algoritmo é o máximo!"

- **ESCREVA** ***<literal>*****, <*****variável*****>, ... ,<*****literal*****>, <variável>**
  Ex: **ESCREVA** "Meu nome é:", nome, "e meu endereço é:", endereco

Daqui por diante, **ESCREVA** será considerada uma palavra reservada e não mais poderá ser utilizada como nome de variável, de modo que toda a vez que for encontrada em algoritmos, será identificada como um comando de saída de dados.

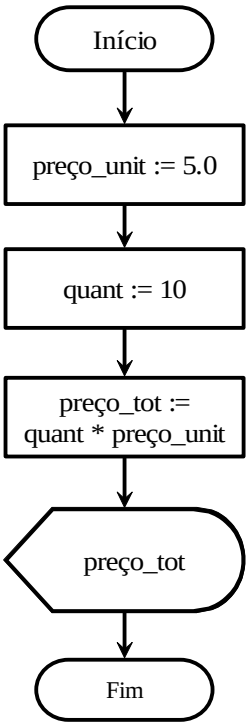
Uma ***lista_de_variáveis*** é um conjunto de nomes de variáveis separados por vírgulas. Um ***literal*** é simplesmente um dado do tipo literal (string ou cadeia de caracteres) delimitado por aspas.

A semântica da instrução primitiva de saída de dados é muito simples: os argumentos do comando são enviados para o dispositivo de saída. No caso de uma lista de variáveis, o conteúdo de cada uma delas é pesquisado na memória e enviado para o dispositivo de saída. No caso de argumentos do tipo literal ou string, estes são enviados diretamente ao referido dispositivo.

Há ainda a possibilidade de se misturar nomes de variáveis com literais na lista de um mesmo comando. O efeito obtido é bastante útil e interessante: a lista é lida da esquerda para a direita e cada elemento da mesma é tratado separadamente; se um nome de variável for encontrado, então a informação da mesma é colocada no dispositivo de saída; no caso de um literal, o mesmo é escrito diretamente no dispositivo de saída.

A seguir temos um exemplo de um algoritmo utilizando o comando de saída de dados:

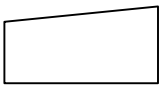
```
Algoritmo exemplo_comando_de_saída_de_dados
Var  preço_unit, preço_tot : real
     quant : inteiro
Início
     preco_unit := 5.0
     quant := 10
     preço_tot := preço_unit * quant
     Escreva preço_tot
Fim.
```



6.3 Comandos de Entrada de Dados

Os **comandos de entrada de dados** são o meio pelo qual as informações dos usuários são transferidas para a memória dos computadores, para que possam ser usadas nos programas.

No diagrama de blocos o comando de entrada de dados é representado por:

 = Operação de entrada de dados

Há duas sintaxes possíveis para esta instrução:

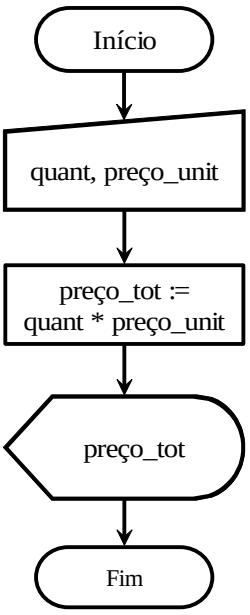
- **LEIA** <***variável***>
  Ex: **LEIA** X

- **LEIA** <***lista_de_variáveis***>
  Ex: **LEIA** nome, endereco, cidade

Da mesma forma que **Escreva**, daqui por diante **Leia** será tratada como uma palavra-reservada e não mais poderá ser usada como nome variável em algoritmos. A **lista_de_variáveis** é um conjunto de um ou mais nomes de variáveis separados por vírgulas.

A semântica da instrução de entrada (ou leitura) de dados é, de certa forma, inversa à da instrução de escrita: os dados são fornecidos ao computador por meio de um dispositivo de entrada e armazenados nas posições de memória das variáveis cujos nomes aparecem na **lista_de_variáveis**.

A seguir temos um exemplo de um algoritmo utilizando o comando de entrada de dados:



```
Algoritmo exemplo_comando_de_entrada_de_dados
Var  preço_unit, preço_tot : real
     quant : inteiro
Início
     Leia preco_unit, quant
     preço_tot := preço_unit * quant
     Escreva preço_tot
Fim.
```

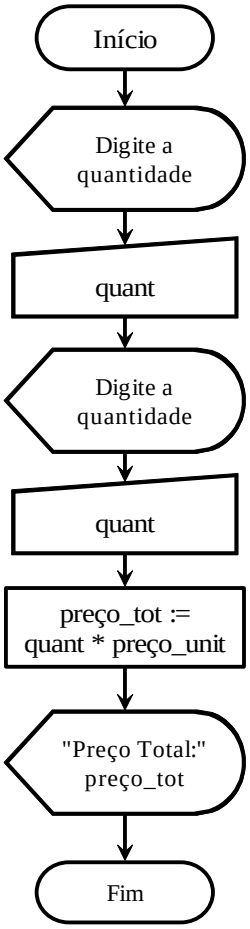
Uma preocupação constante de um bom programador deve ser a de conceber um programa “amigo do usuário”. Esta preocupação é traduzida no planejamento de uma **interface com o usuário** (meio pelo qual um programa e o usuário “conversam”) bastante amigável. Em termos práticos, isto se resume à aplicação de duas regras básicas:

- toda vez que um programa estiver esperando que o usuário forneça a ele um determinado dado (operação de leitura), ele deve antes enviar uma mensagem dizendo ao usuário o que ele deve digitar, por meio de uma instrução de saída de dados;

- antes de enviar qualquer resultado ao usuário, um programa deve escrever uma mensagem explicando o significado do mesmo.

Estas medidas tornam o diálogo entre o usuário e o programador muito mais fácil.

A seguir temos um exemplo do algoritmo anterior, utilizando as regras de construção de uma interface amigável:



```
Algoritmo exemplo_interface_amigavel
Var  preço_unit, preço_tot : real
     quant : inteiro
Início
     Escreva "Digite o preço unitário:"
     Leia preco_unit
     Escreva "Digite a quantidade:"
     Leia quant
     preço_tot := preço_unit * quant
     Escreva "Preço total: ", preço_tot
Fim.
```

6.4 Entrada, Processamento e Saída

Para se criar um programa que seja executável dentro de um computador, você deverá ter em mente três pontos de trabalho: a entrada de dados, o seu processamento e a saída dos mesmos. Sendo assim, todo programa estará trabalhando com estes três conceitos. Se os dados forem entrados de forma errada, serão consequentemente processados de forma errada e resultarão em respostas erradas. Desta forma, dizer a alguém que foi erro do computador é ser um tanto "medíocre". E isto é o que mais ouvimos quando nosso saldo está errado e vamos ao banco fazer uma reclamação, ou quando recebemos uma cobrança indevida. Se houve algum erro, é porque foi causado por falha humana. Realmente é impossível um computador errar por vontade própria, pois vontade é uma coisa que os computadores não têm.

Uma entrada e uma saída poderão ocorrer dentro de um computador de diversas formas. Por exemplo, uma entrada poderá ser feita via teclado, modem, leitores óticos, disco, entre outras. Uma saída poderá ser feita em vídeo, impressora, disco, entre outras formas.

6.5 Funções Matemáticas

ABS (x)	Retorna o valor absoluto de uma expressão
ARCTAN (x)	Retorna o arco de tangente do argumento utilizado
COS (r)	Retorna o valor do co-seno
EXP (r)	Retorna o valor exponencial
FRAC (r)	Retorna a parte fracionária
LN (r)	Retorna o logaritmo natural
PI	Retorna o valor de PI
SIN (r)	Retorna o valor do seno
SQR (r)	Retorna o parâmetro elevado ao quadrado.

SQRT (r) Retorna a raiz quadrada

Nem todas as funções que necessitamos estão prontas e às vezes é necessário utilizar uma fórmula equivalente:

$\mathbf{Y^X}$ **= EXP (LN (Y) * X)**

$\sqrt[X]{Y}$ **= EXP (LN (Y) * (1 / X))**

LOG (x) **= LN (** X **) / LN (10)**

7. ESTRUTURAS DE CONTROLE DO FLUXO DE EXECUÇÃO

Até o momento os algoritmos estudados utilizam apenas instruções primitivas de atribuição, e de entrada e saída de dados. Qualquer conjunto de dados fornecido a um algoritmo destes será submetido ao mesmo conjunto de instruções, executadas sempre na mesma seqüência.

No entanto, na prática muitas vezes é necessário executar ações diversas em função dos dados fornecidos ao algoritmo. Em outras palavras, dependendo do conjunto de dados de entrada do algoritmo, deve-se executar um conjunto diferente de instruções. Além disso, pode ser necessário executar um mesmo conjunto de instruções um número repetido de vezes. Em resumo é necessário controlar o fluxo de execução das instruções (a seqüência em que as instruções são executadas num algoritmo) em função dos dados fornecidos como entrada do mesmo.

De acordo com o modo como o controle do fluxo de instruções de um algoritmo é feito, as estruturas básicas de controle são classificadas em:

- estruturas seqüenciais
- estruturas de decisão
- estruturas de repetição

7.1 Comandos Compostos

Um **comando composto** é um conjunto de zero ou mais comandos (ou instruções) simples, como atribuições e instruções primitivas de entrada ou saída de dados, ou alguma das construções apresentadas neste capítulo.

Este conceito é bastante simples e será útil e conveniente nos itens seguintes, na definição das estruturas básicas de controle de execução.

7.2 Estrutura Seqüencial

Na estrutura seqüencial os comandos de um algoritmo são executados numa seqüência pré-estabelecida. Cada comando é executado somente após o término do comando anterior.

Uma estrutura seqüencial é delimitada pelas palavras-reservadas **Início** e **Fim** e contém basicamente comandos de atribuição, comandos de entrada e comandos de saída. Os algoritmos do capítulo anterior são algoritmos que utilizam uma única estrutura seqüencial.

Um algoritmo puramente seqüencial é aquele cuja execução é efetuada em ordem ascendente dos números que identificam cada passo. A passagem de um passo ao seguinte é natural e automática, e cada passo é executado uma única vez.

7.3 Estruturas de Decisão

Neste tipo de estrutura o fluxo de instruções a ser seguido é escolhido em função do resultado da avaliação de uma ou mais condições. Uma **condição** é uma expressão lógica.

A classificação das estruturas de decisão é feita de acordo com o número de condições que devem ser testadas para que se decida qual o caminho a ser seguido. Segundo esta classificação, têm-se 3 tipos de estruturas de decisão:

- Estrutura de Decisão Simples (*Se ... então*)
- Estrutura de Decisão Composta (*Se ... então ... senão*)
- Estrutura de Decisão Múltipla do Tipo Escolha (*Escolha ... Caso ... Senão*)

Os algoritmos puramente seqüenciais podem ser usados na solução de um grande número de problemas, porém existem problemas que exigem o uso de alternativas de acordo com as entradas do mesmo.

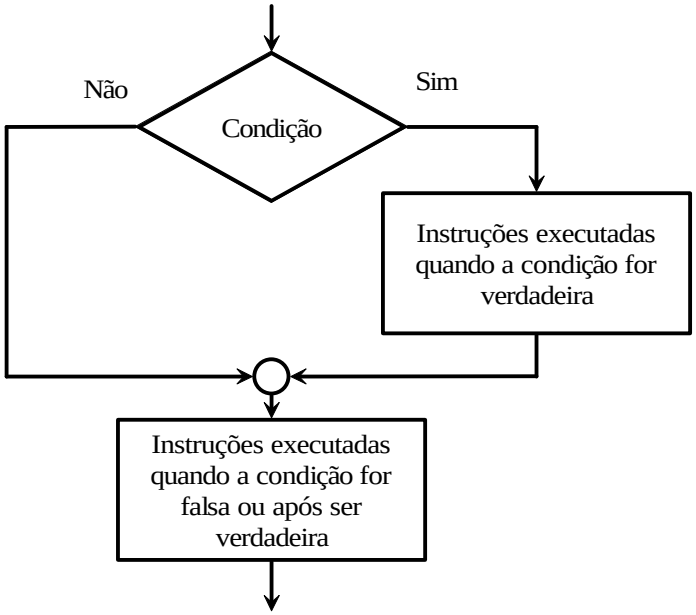
Nestes algoritmos, as situações são resolvidas através de passos cuja execução é subordinada a uma condição. Assim, o algoritmo conterá passos que são executados somente se determinadas condições forem observadas.

Um algoritmo em que se tem a execução de determinados passos subordinada a uma condição é denominado **algoritmo com seleção**.

7.3.1 Estruturas de Decisão Simples (*Se ... então*)

Nesta estrutura uma única condição (expressão lógica) é avaliada. Dependendo do resultado desta avaliação, um comando ou conjunto de comandos serão executados (se a avaliação for verdadeira) ou não serão executados (se a avaliação for falsa).

No diagrama de blocos a estrutura para instrução se...então é representado por:



Há duas sintaxes possíveis para a estrutura de decisão simples:

- **SE** <***condição***> **ENTÃO** <***comando_único***>
  Ex: **SE** X > 10 **ENTÃO** Escreva "X é maior que 10"

- **SE** <***condição***> **ENTÃO**
  INÍCIO
  <***comando_composto***>
  FIM
  Ex: **SE** X > 10 **ENTÃO**
  INÍCIO
  cont := cont + 1
  soma := soma + x
  Escreva "X é maior que 10"
  FIM

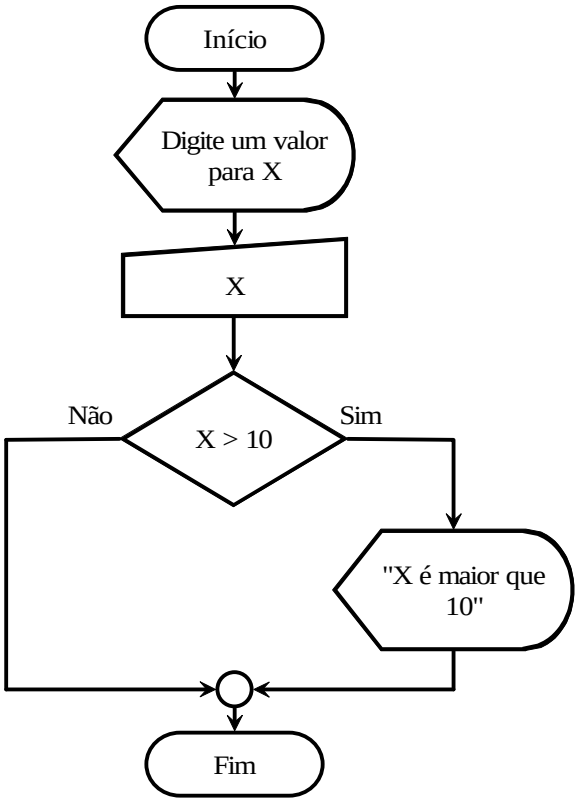
A semântica desta construção é a seguinte: a condição é avaliada:

– Se o resultado for verdadeiro, então o ***comando_único*** ou o conjunto de comandos (***comando_composto***) delimitados pelas palavras-reservadas ***início*** e ***fim*** serão executados. Ao término de sua execução o fluxo do algoritmo prossegue pela instrução seguinte à construção, ou seja, o primeiro comando após o ***comando_único*** ou a palavra-reservada ***fim***.

– No caso da condição ser falsa, o fluxo do algoritmo prossegue pela instrução seguinte à construção, ou seja, o primeiro comando após o ***comando_único*** ou a palavra-reservada ***fim***, sem executar o ***comando_único*** ou o conjunto de comandos (***comando_composto***) entre as palavras-reservadas ***início*** e ***fim***.

Exemplo de algoritmo que lê um número e escreve se o mesmo é maior que 10:

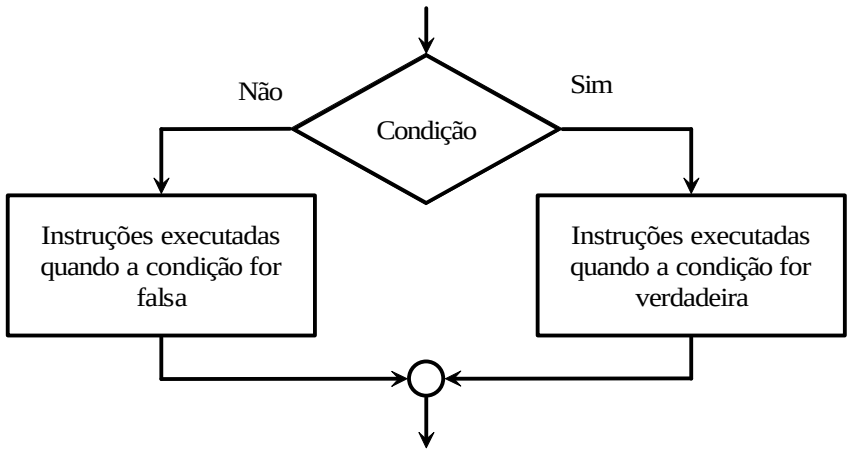
```
Algoritmo exemplo_estrutura_de_decisão_simples
Var X : inteiro
Início
Escreva “Digite um valor”
Leia X
Se X > 10 Então Escreva “X é maior que 10”
Fim.
```



7.3.2 Estruturas de Decisão Composta (*Se ... então ... senão*)

Nesta estrutura uma única condição (expressão lógica) é avaliada. Se o resultado desta avaliação for verdadeiro, um comando ou conjunto de comandos serão executados. Caso contrário, ou seja, quando o resultado da avaliação for falso, um outro comando ou um outro conjunto de comandos serão executados.

No diagrama de blocos a estrutura para instrução se...então...senão é representado por:



Há duas sintaxes possíveis para a estrutura de decisão composta:

- **SE** <***condição***> **ENTÃO** <***comando_único_1***>
  SENÃO <***comando_único_2***>

  Ex: **SE** X > 100 **ENTÃO** Escreva “X é maior que 100”
  SENÃO Escreva “X não é maior que 100”

- **SE** <***condição***> **ENTÃO**
  INÍCIO
  <***comando_composto_1***>
  FIM
  SENÃO
  INÍCIO
  <***comando_composto_2***>
  FIM

  Ex: **SE** X > 100 **ENTÃO**
  INÍCIO
  cont_a := cont_a + 1
  soma_a := soma_a + x
  Escreva “X é maior que 100”
  FIM
  SENÃO
  INÍCIO
  cont_b := cont_b + 1

soma_b := soma_b + x
Escreva “X não é maior que 100”
FIM

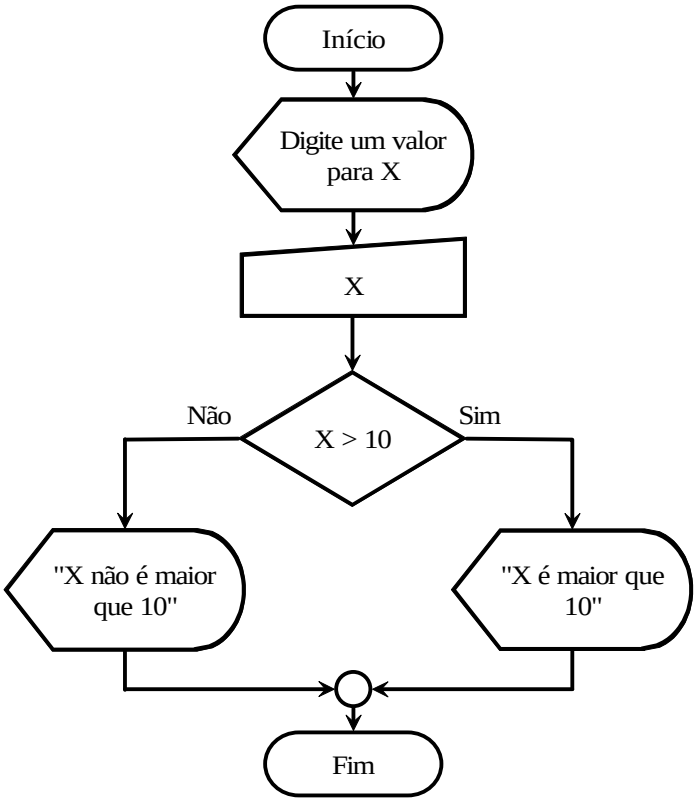
A semântica desta construção é a seguinte: a condição é avaliada:

– Se o resultado for verdadeiro, então o ***comando_único_1*** ou o conjunto de comandos (***comando_composto_1***) delimitados pelas palavras-reservadas ***início*** e ***fim*** serão executados. Ao término de sua execução o fluxo do algoritmo prossegue pela instrução seguinte à construção, ou seja, o primeiro comando após o ***comando_único_2*** ou a palavra-reservada ***fim*** do ***comando_composto_2***.

– Nos casos em que a condição é avaliada como falsa, o ***comando_único_2*** ou o conjunto de comandos (***comando_composto_2***) delimitados pelas palavras-reservadas ***início*** e ***fim*** serão executados. Ao término de sua execução o fluxo do algoritmo prossegue pela instrução seguinte à construção, ou seja, o primeiro comando após o ***comando_único_2*** ou a palavra-reservada ***fim*** do ***comando_composto_2***.

Exemplo de algoritmo que lê um número e escreve se o mesmo é ou não maior que 100:

```
Algoritmo exemplo_estrutura_de_decisão_composta
Var X : inteiro
Início
Leia X
Se X > 10 Então Escreva “X é maior que 10”
            Senão Escreva “X não é maior que 10”
Fim.
```



Nos algoritmos, a correta formulação de condições, isto é, expressões lógicas, é de fundamental importância, visto que as estruturas de seleção são baseadas nelas. As diversas formulações das condições podem levar a algoritmos distintos. Considerando o problema:

*"Dado um par de valores **x**, **y**, que representam as coordenadas de um ponto no plano, determinar o quadrante ao qual pertence o ponto, ou se está sobre um dos eixos cartesianos."*

A solução do problema consiste em determinar todas as combinações de ***x*** e ***y*** para as classes de valores positivos, negativos e nulos.

Os algoritmos podem ser baseados em estruturas concatenadas uma em seqüência a outra ou em estruturas aninhadas uma dentro da outra, de acordo com a formulação da condição.

O algoritmo a seguir utiliza estruturas concatenadas.

```
Algoritmo estruturas_concatenadas
Var x, y : inteiro
Início
Ler x, y
Se x=0  e y=0  Então Escrever "Ponto na origem"
Se x=0  e y<>0 Então Escrever "Ponto sobre o eixo y"
Se x<>0 e y=0  Então Escrever "Ponto sobre o eixo x"
Se x>0  e y>0  Então Escrever "Ponto no quandrante 1"
Se x<0  e y>0  Então Escrever "Ponto no quandrante 2"
Se x<0  e y<0  Então Escrever "Ponto no quandrante 3"
Se x>0  e y<0  Então Escrever "Ponto no quandrante 4"
Fim.
```

O algoritmo a seguir utiliza estruturas aninhadas ou encadeadas.

```
Algoritmo estruturas_aninhadas
Var x, y : inteiro
Início
Ler x, y
Se x<>0
Então Se y=0
       Então Escrever "Ponto sobre o eixo x"
       Senão Se x>0
              Então Se y>0
                     Então Escrever "Ponto no quandrante 1"
                     Senão Escrever "Ponto no quandrante 4"
              Senão Se y>0
                     Então Escrever "Ponto no quandrante 2"
                     Senão Escrever "Ponto no quandrante 3"
Senão Se y=0
       Então Escrever "Ponto na origem"
       Senão Escrever "Ponto sobre o eixo y"
Fim.
```

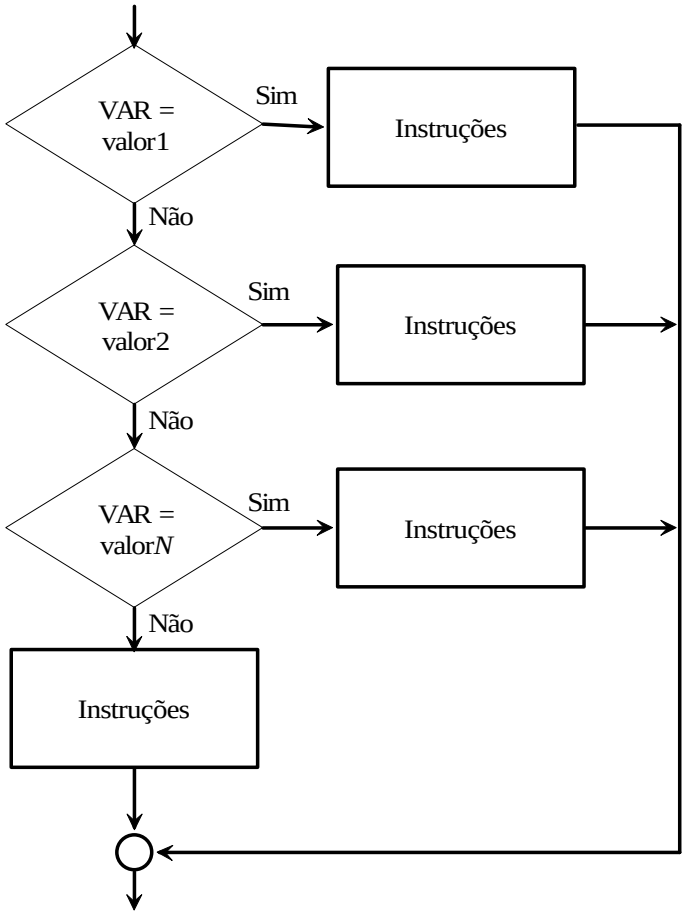
As estruturas concatenadas tem a vantagem de tornar o algoritmo mais legível, facilitando a correção do mesmo em caso de erros. As estruturas aninhadas ou encadeadas tem a vantagem de tornar o algoritmo mais rápido pois são efetuados menos testes e menos comparações, o que resulta num menor número de passos para chegar ao final do mesmo.

Normalmente se usa estruturas concatenadas nos algoritmos devido à facilidade de entendimento das mesmas e estruturas aninhadas ou encadeadas somente nos casos em que seu uso é fundamental.

7.3.3 Estruturas de Decisão Múltipla do Tipo Caso (*Caso ... fim_caso ... senão*)

Este tipo de estrutura é uma generalização da construção **Se**, onde somente uma condição era avaliada e dois caminhos podiam ser seguidos. Na estrutura de decisão do tipo **Caso** pode haver uma ou mais condições a serem testadas e um comando diferente associado a cada uma destas.

No diagrama de blocos a estrutura para instrução caso...fim_caso...senão é representado por:



A sintaxe da construção de **Caso** é:

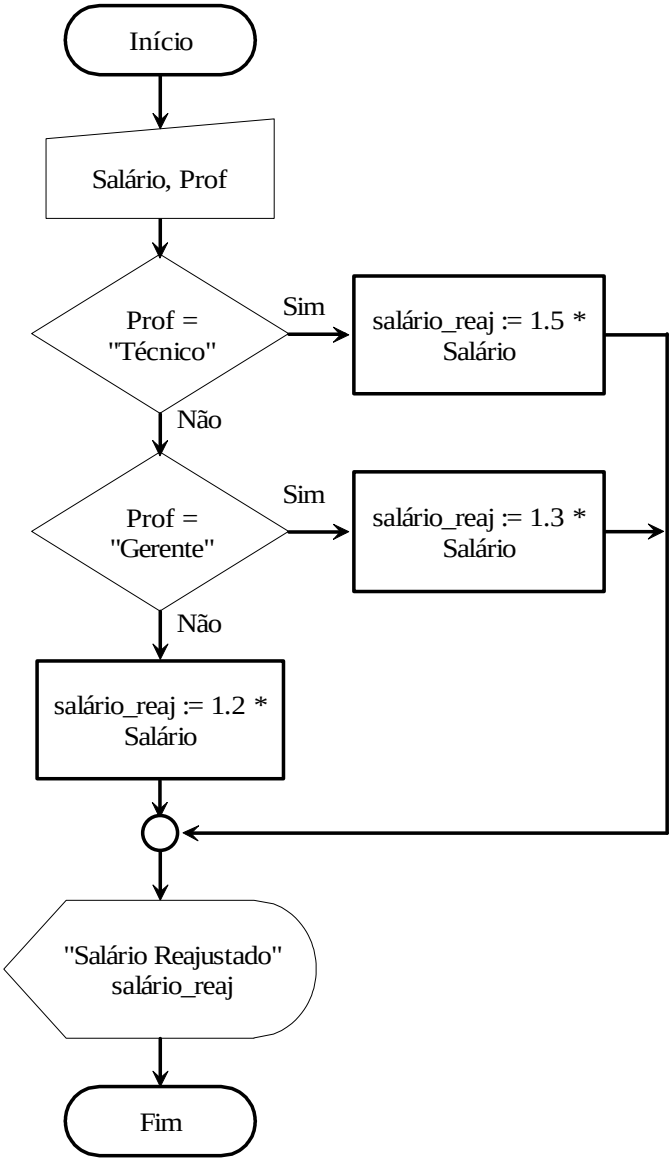
CASO <***variável***>
    SEJA <***condição_1***> **FAÇA**
        <***comando_composto_1***>
    SEJA <***condição_2***> **FAÇA**
        <***comando_composto_2***>
    ...
    SEJA <***condição_n***> **FAÇA**
        <***comando_composto_n***>
    SENÃO
        <***comando_composto_s***>
FIM

Seu funcionamento é o seguinte: ao entrar-se numa construção do tipo **Caso**, a ***condição_1*** é testada com a variável: se for verdadeira, o ***comando_composto_1*** é executado e após seu término, o fluxo de execução prossegue pela primeira instrução após o final da construção (***fim***); se a ***condição_1*** for falsa, a ***condição_2*** é testada: se esta for verdadeira, o ***comando_composto_2*** é executado e ao seu término, a execução prossegue normalmente pela instrução seguinte ao final da construção (***fim***). O mesmo raciocínio é estendido a todas as condições da construção. No caso em que todas as condições são avaliadas como falsas, o ***comando_composto_s*** (correspondente ao **Senão** da construção) é executado.

Um caso particular desta construção é aquele em que o ***comando_composto_s*** não contém nenhuma instrução. Isto ocorre nas situações que não se deseja efetuar nenhuma ação quando todas as condições são falsas. Assim, pode-se dispensar o uso do **Senão** na construção **Caso**.

Um exemplo de aplicação desta construção é o algoritmo para reajustar o salário de acordo com a função. Se for técnico, aumentar o salário 50%, se for gerente, aumentar 30% e se for outro cargo, aumentar 20%.

```
Algoritmo exemplo_estrutura_do_tipo_escolha
Var salário, salário_reaj : real
    prof: caracter[20]
Início
   Leia salário, prof
   Caso prof
       Seja “Técnico” faça salário_reaj := 1.5 * salário
       Seja “Gerente” faça salário_reaj := 1.3 * salário
       Senão salário_reaj := 1.2 * salário
   Fim
   Escreva “Salário Reajustado = ”, salário_reaj
Fim.
```


Início
Salário, Prof
Prof = "Técnico"
Sim
salário_reaj := 1.5 * Salário
Não
Prof = "Gerente"
Sim
salário_reaj := 1.3 * Salário
Não
salário_reaj := 1.2 * Salário
"Salário Reajustado" salário_reaj
Fim

7.4 Estruturas de Repetição

São muito comuns as situações em que se deseja repetir um determinado trecho de um programa um certo número de vezes. Por exemplo, pode-se citar o caso de um algoritmo que calcula a soma dos números impares entre 500 e 1000 ou então um algoritmo que escreve os números maiores que 0 enquanto a sua soma não ultrapasse 1000.

As estruturas de repetição são muitas vezes chamadas de **Laços** ou também de **Loops**.

A classificação das estruturas de repetição é feito de acordo com o conhecimento prévio do número de vezes que o conjunto de comandos será executado. Assim os **Laços** se dividem em:

- **Laços Contados**, quando se conhece previamente quantas vezes o comando composto no interior da construção será executado;

- **Laços Condicionais**, quando não se conhece de antemão o número de vezes que o conjunto de comandos no interior do laço será repetido, pelo fato do mesmo estar amarrado a uma condição sujeita à modificação pelas instruções do interior do laço.

Todo algoritmo que possui um ou mais de seus passos repetidos um determinado número de vezes denomina-se **algoritmo com repetição**.

Com a utilização de estruturas de repetição para a elaboração de algoritmos, torna-se necessário o uso de dois tipos de variáveis para a resolução de diversos tipos de problemas: **variáveis contadoras** e **variáveis acumuladoras**.

Uma **variável contadora** é uma variável que recebe um valor inicial, geralmente 0 (zero) antes do início de uma estrutura de repetição, e é incrementada no interior da estrutura de um valor constante, geralmente 1, conforme o exemplo abaixo:

```
...
cont := 0
<estrutura_de_repetição>
        ...
        cont := cont + 1
        ...
<fim_da_estrutura_de_repetição>
...
```

Uma **variável acumuladora** é uma variável que recebe um valor inicial, geralmente 0 (zero) antes do início de uma estrutura de repetição, e é incrementada no interior da estrutura de um valor variável, geralmente a variável usada na estrutura de controle, conforme o exemplo abaixo:

```
...
soma := 0
<estrutura_de_repetição_com_variável_x>
        ...
        soma := soma + x
        ...
```

```
<fim_da_estrutura_de_repetição>
...
```

7.4.1 Laços Condicionais

Laços condicionais são aqueles cujo conjunto de comandos em seu interior é executado até que uma determinada condição seja satisfeita. Ao contrário do que acontece nos laços contados, nos laços condicionais não se sabe de antemão quantas vezes o corpo do laço será executado.

As construções que implementam laços condicionais mais comuns nas linguagens de programação modernas são:

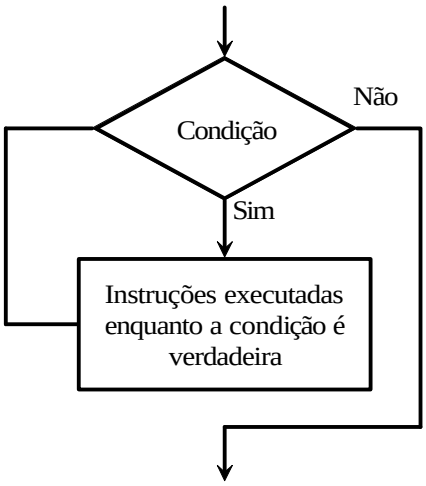
- **Enquanto** - laço condicional com teste no início
- **Repita** - laço condicional com teste no final

Nos laços condicionais a variável que é testada, tanto no início quanto no final do laço, dever sempre estar associada a um comando que a atualize no interior do laço. Caso isso não ocorra, o programa ficará repetindo indefinidamente este laço, gerando uma situação conhecida como “laço infinito”.

7.4.1.1 Laços Condicionais com Teste no Início *(Enquanto ... faça)*

Caracteriza-se por uma estrutura que efetua um teste lógico no início de um laço, verificando se é permitido ou não executar o conjunto de comandos no interior do laço.

No diagrama de blocos a estrutura **Enquanto ... faça** é representada por:



A sintaxe é mostrada a seguir:

ENQUANTO <***condição***> **FAÇA**
INÍCIO
<***comando_composto***>

FIM

Sua semântica é a seguinte: ao início da construção **Enquanto** a condição é testada. Se seu resultado for falso, então o comando composto no seu interior não é executado e a execução prossegue normalmente pela instrução seguinte à palavra-reservada ***fim*** que identifica o final da construção.

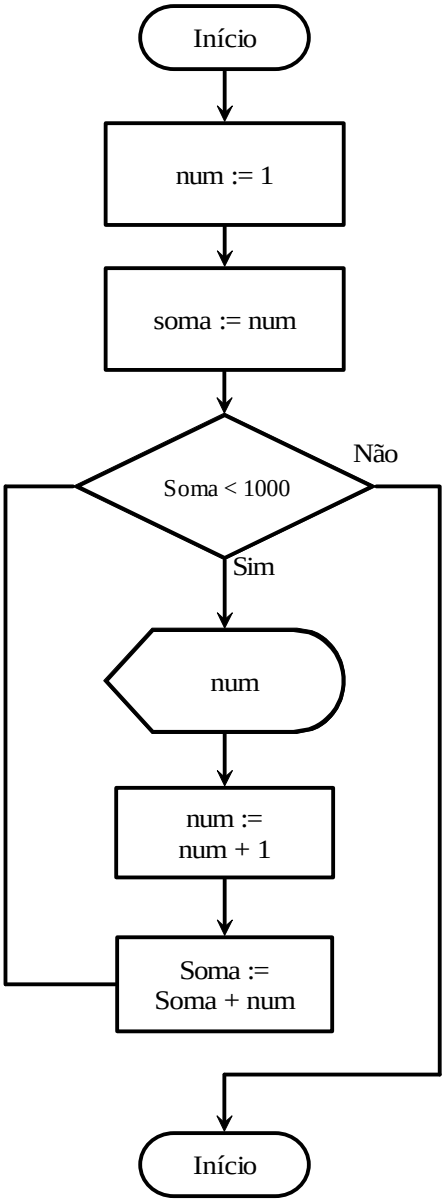
Se a condição for verdadeira o comando composto é executado e ao seu término retorna-se ao teste da condição. Assim, o processo acima será repetido **enquanto** a condição testada for verdadeira. Quando esta for falsa, o fluxo de execução prossegue normalmente pela instrução seguinte à palavra-reservada ***fim*** que identifica o final da construção.

Uma vez dentro do corpo do laço, a execução somente abandonará o mesmo quando a condição for falsa. O usuário deste tipo de construção deve estar atento à necessidade de que em algum momento a condição deverá ser avaliada como falsa. Caso contrário, o programa permanecerá indefinidamente no interior do laço (laço infinito).

Neste tipo de laço condicional a variável a ser testada deve possuir um valor associado antes da construção do laço.

O algoritmo que escreve os números maiores que 0 enquanto a sua soma não ultrapasse 1000 é um exemplo deste tipo de laço:

```
Algoritmo exemplo_enquanto
Var soma, num : inteiro
Início
num := 1
soma:= num
Enquanto soma < 1000 Faça
       Início
       Escreva num
       num := num + 1
       soma := soma + num
       Fim
Fim.
```



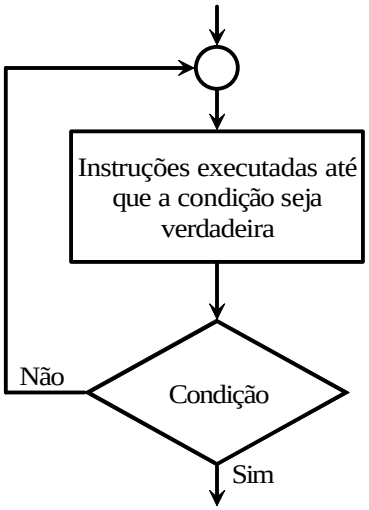
7.4.1.2 Laços Condicionais com Teste no Final *(Repita ... até que)*

Caracteriza-se por uma estrutura que efetua um teste lógico no final de um laço, verificando se é permitido ou não executar novamente o conjunto de comandos no interior do mesmo.

A sintaxe é mostrada a seguir:

REPITA
<*comando_composto*>
ATÉ QUE <*condição*>

No diagrama de blocos a estrutura **Repita ... até que** é representada por:



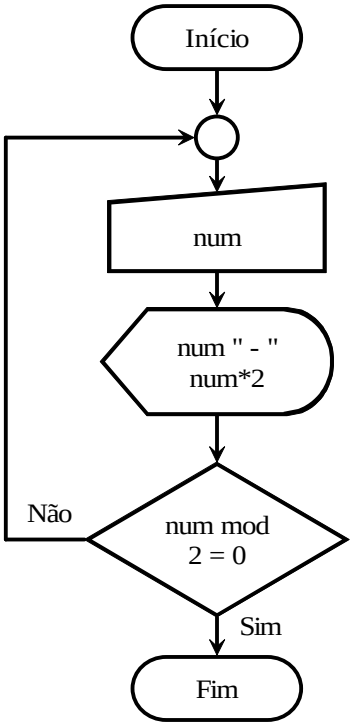
Seu funcionamento é bastante parecido ao da construção **Enquanto**. O comando é executado uma vez. A seguir, a condição é testada: se ela for falsa, o comando composto é executado novamente e este processo é repetido até que a condição seja verdadeira, quando então a execução prossegue pelo comando imediatamente seguinte ao final da construção.

Esta construção difere da construção **Enquanto** pelo fato de o comando composto ser executado **uma** ou mais vezes (pelo menos uma vez), ao passo que na construção **Enquanto** o comando composto é executado **zero** ou mais vezes (possivelmente nenhuma). Isto acontece porque na construção **Repita** o teste é feito no final da construção, ao contrário do que acontece na construção **Enquanto**, onde o teste da condição é efetuado no início da mesma.

A construção **Repita** também difere da construção **Enquanto** no que se refere à inicialização da variável, visto que na construção **Repita** a variável pode ser inicializada ou lida dentro do laço.

O algoritmo que lê um número não determinado de vezes um valor do teclado e escreve o valor e o seu quadrado, até que seja digitado um valor par, é um exemplo desta estrutura:

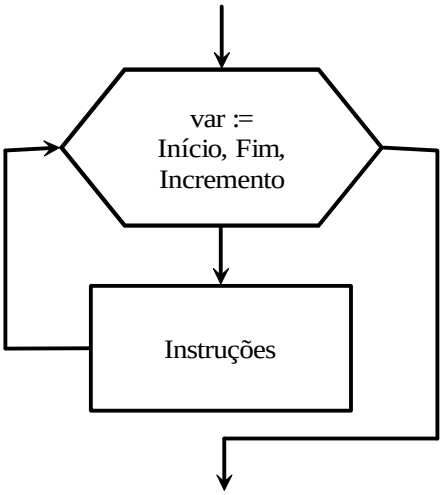
```
Algoritmo exemplo_repita
Var num : inteiro
Início
Repita
       Ler num
       Escrever num, " - ", num * num
Até que num mod 2 = 0
Fim.
```



7.4.2 Laços Contados *(Para ... faça)*

Os laços contados são úteis quando se conhece previamente o número exato de vezes que se deseja executar um determinado conjunto de comandos. Então, este tipo de laço nada mais é que uma estrutura dotada de mecanismos para contar o número de vezes que o corpo do laço (ou seja, o comando composto em seu interior) é executado.

No diagrama de blocos a estrutura para instrução **Para** é representado por:



Há duas sintaxes possíveis usadas em algoritmos para os laços contados:

- **PARA** <***variável***> **:=** <***início***> **ATÉ** <***final***> **FAÇA**
  <***comando_único***>

  Ex.: **PARA** i := 1 **ATÉ** 10 **FAÇA**
  ESCREVER i, " x 7 = ", i * 7

- **PARA** <***variável***> **:=** <***início***> **ATÉ** <***final***> **FAÇA**
  INÍCIO
  <***comando_composto***>
  FIM

  Ex.:
  soma := 0
  PARA i := 1 **ATÉ** 10 **FAÇA**
  INÍCIO
  soma := soma + i
  ESCREVER i, " x 7 = ", i * 7
  ESCREVER "Soma acumulada = ", soma
  FIM

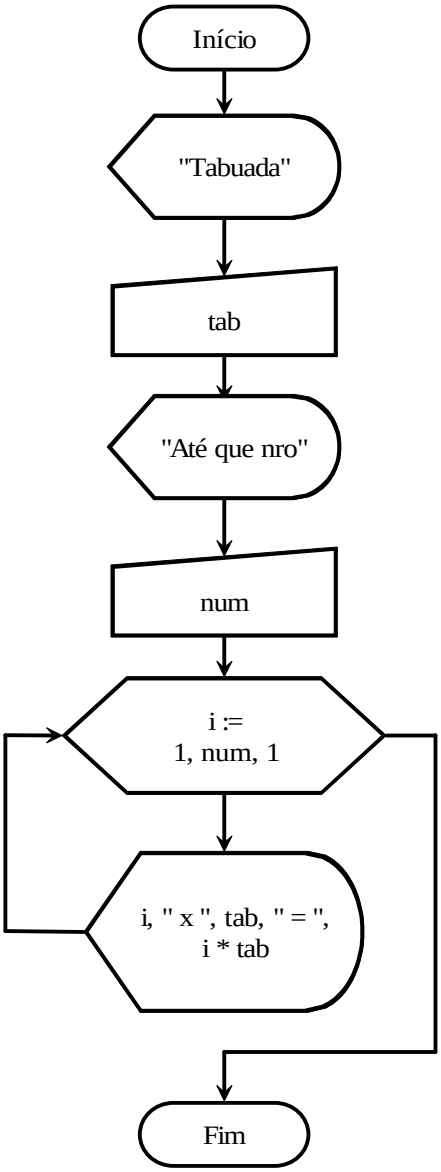
A semântica do laço contado é a seguinte: no início da execução da construção o valor ***início*** é atribuído à variável ***var***. A seguir, o valor da variável ***var*** é comparado com o valor ***final***. Se ***var*** for maior que ***final***, então o comando composto não é executado e a execução do algoritmo prossegue pelo primeiro comando seguinte ao ***comando_único*** ou à palavra-reservada ***fim*** que delimita o final da construção. Por outro lado, se o valor de ***var*** for menor ou igual a ***final***, então o comando composto no interior da construção é executado e, ao final do mesmo a variável ***var*** é incrementada em 1 unidade. Feito isso, retorna-se à comparação entre ***var*** e ***final*** e repete-se o processo até que ***var*** tenha um valor maior que ***final***, quando o laço é finalizado e a execução do algoritmo prossegue pela instrução imediatamente seguinte à construção.

Existe uma condição especial em que a contagem deve ser de forma decrescente, onde o valor a variável é decrementado em uma unidade. A sintaxe deste laço é a seguinte:

PARA <***variável***> **:=** <***início***> **ATÉ** <***final***> **PASSO -1 FAÇA**
INÍCIO
<***comando_composto***>
FIM

Exemplo de um algoritmo que escreve a tabuada de um número específico:

```
Algoritmo tabuada
Var i, tab, num : inteiro
Início
Escrever “Tabuada: ”
Ler tab
Escrever “Até que número: ”
Ler num
Para i := 1 Até num Faça
       Início
       Escrever i, “ x ”, tab, “ = ”, i * tab
       Fim
Fim.
```

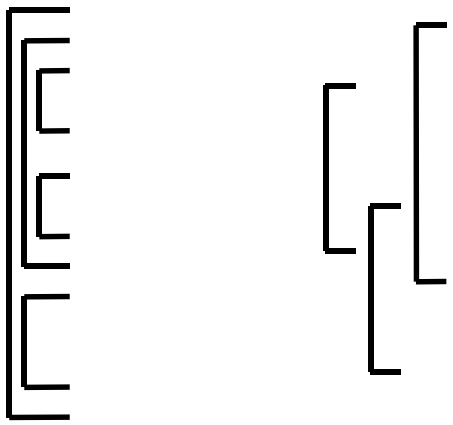


7.5 Estruturas de Controle Encadeadas ou Aninhadas

Um aninhamento ou encadeamento é o fato de se ter qualquer um dos tipos de construção apresentados anteriormente dentro do conjunto de comandos (comando composto) de uma outra construção.

Em qualquer tipo de aninhamento é necessário que a construção interna esteja completamente embutida na construção externa.

A figura 7.1 a seguir ilustra aninhamentos válidos e inválidos.



(*a*) (*b*)

Figura 7.1 Exemplos de aninhamentos (*a*) válidos e (*b*) inválidos

8. ESTRUTURAS DE DADOS HOMOGÊNEAS

As estruturas de dados homogêneas permitem agrupar diversas informações dentro de uma mesma variável. Este agrupamento ocorrerá obedecendo sempre ao mesmo tipo de dado, e é por esta razão que estas estruturas são chamadas homogêneas.

A utilização deste tipo de estrutura de dados recebe diversos nomes, como: variáveis indexadas, variáveis compostas, variáveis subscritas, arranjos, vetores, matrizes, tabelas em memória ou *arrays*. Os nomes mais usados e que utilizaremos para estruturas homogêneas são: matrizes (genérico) e vetores (matriz de uma linha e várias colunas).

8.1 Matrizes de Uma Dimensão ou Vetores

Este tipo de estrutura em particular é também denominado por profissionais da área como matrizes unidimensionais. Sua utilização mais comum está vinculada à criação de tabelas. Caracteriza-se por ser definida uma única variável vinculada dimensionada com um determinado tamanho. A dimensão de uma matriz é constituída por constantes inteiras e positivas. Os nomes dados às matrizes seguem as mesmas regras de nomes utilizados para indicar as variáveis simples.

A sintaxe do comando de definição de vetores é a seguinte:

Var
<*nome_da_variável*> : MATRIZ [<*coluna_inicial*> .. <*coluna_final*>] DE <*tipo_de_dado*>

Ex.: **VAR**
M : **MATRIZ** [1 .. 10] **DE INTEIRO**

8.1.1 Operações Básicas com Matrizes do Tipo Vetor

Do mesmo modo que acontece com variáveis simples, também é possível operar com variáveis indexadas (matrizes). Contudo não é possível operar diretamente com o conjunto completo, mas com cada um de seus componentes isoladamente.

O acesso individual a cada componente de um vetor é realizado pela especificação de sua posição na mesma por meio do seu índice. No exemplo anterior foi definida uma variável **M** capaz de armazenar 10 número inteiros. Para acessar um elemento deste vetor deve-se fornecer o nome do mesmo e o índice do componente desejado do vetor (um número de 1 a 10, neste caso).

Por exemplo, **M[1]** indica o primeiro elemento do vetor, **M[2]** indica o segundo elemento do vetor e **M[10]** indica o último elemento do vetor.

Portanto, não é possível operar diretamente sobre vetores como um todo, mas apenas sobre seus componentes, um por vez. Por exemplo, para somar dois vetores é necessário somar

cada um de seus componentes dois a dois. Da mesma forma as operações de atribuição, leitura e escrita de vetores devem ser feitas elemento a elemento.

8.1.1.1 Atribuição de Uma Matriz do Tipo Vetor

No capítulo sobre as instruções primitivas, o comando de atribuição foi definido como:

<*nome_da_variável*> := <*expressão*>

No caso de vetores (variáveis indexadas), além do nome da variável deve-se necessariamente fornecer também o índice do componente do vetor onde será armazenado o resultado da avaliação da expressão.

Ex.: M[1] **:=** 15
M[2] **:=** 150
M[5] **:=** 10
M[10] **:=** 35

8.1.1.2 Leitura de Dados de Uma Matriz do Tipo Vetor

A leitura de um vetor é feita passo a passo, um de seus componentes por vez, usando a mesma sintaxe da instrução primitiva da entrada de dados, onde além do nome da variável, deve ser explicitada a posição do componente lido:

LEIA <*nome_da_variável*> [<*índice*>]

Uma observação importante a ser feita é a utilização da construção **Para** a fim de efetuar a operação de leitura repetidas vezes, em cada uma delas lendo um determinado componente do vetor. De fato esta construção é muito comum quando se opera com vetores, devido à necessidade de se realizar uma mesma operação com os diversos componentes dos mesmos. Na verdade, são raras as situações que se deseja operar isoladamente com um único componente do vetor.

O algoritmo a seguir exemplifica a operação de leitura de um vetor:

```
Algoritmo exemplo_leitura_de_vetor
Var
numeros : matriz[1..10] de inteiro
i : inteiro
Início
Para i de 1 até 10 faça
       Início
       Leia numeros[i]
       Fim
Fim.
```

8.1.1.3 Escrita de Dados de Uma Matriz do Tipo Vetor

A escrita de um vetor obedece à mesma sintaxe da instrução primitiva de saída de dados e também vale lembrar que, além do nome do vetor, deve-se também especificar por meio do índice o componente a ser escrito:

ESCREVA ***<nome_da_variável>*** **[** ***<índice>*** **]**

O algoritmo a seguir exemplifica a operação de leitura e escrita de um vetor, utilizando a construção **Para**:

```
Algoritmo exemplo_escrita_de_vetor
Var
numeros : matriz[1..10] de inteiro
i : inteiro
Início
Para i de 1 até 10 faça
       Início
       Leia numeros[i]
       Fim
Para i de 1 até 10 faça
       Início
       Escreva numeros[i]
       Fim
Fim.
```

Um exemplo mais interessante é mostrado a seguir, onde um vetor de dez números é lido e guardado no vetor **numeros**. Paralelamente, a soma destes números é calculada e mantida na variável **soma**, que posteriormente é escrita.

```
Algoritmo exemplo_escrita_de_vetor_com_soma
Var
numeros : matriz[1..10] de inteiro
i : inteiro
soma : inteiro
Início
soma := 0
Para i de 1 até 10 faça
       Início
       Leia numeros[i]
       soma := soma + numeros[i]
       Fim
Para i de 1 até 10 faça
       Início
       Escreva numeros[i]
       Fim
Escrever “Soma = ”, soma
Fim.
```

8.1.2 Exemplos de Aplicação de Vetores

O espectro de aplicação de vetores em algoritmos é muito extenso, mas normalmente os vetores são usados em duas tarefas muito importantes no processamento de dados: **pesquisa** e **classificação**.

A **pesquisa** consiste na verificação da existência de um valor dentro de um vetor. Trocando em miúdos, pesquisar um vetor consiste em procurar dentre seus componentes um determinado valor.

A **classificação** de um vetor consiste em arranjar seus componentes numa determinada ordem, segundo um critério específico. Por exemplo, este critério pode ser a ordem alfabética de um vetor de dados caracter, ou então a ordem crescente ou decrescente para um vetor de dados numéricos. Há vários métodos de classificação, mas o mais conhecido é o **método da bolha** de classificação (**Bubble Sort**).

8.1.2.1 O Método da Bolha de Classificação

Este método não é o mais eficiente, mas é um dos mais populares devido à sua simplicidade.

A filosofia básica deste método consiste em "varrer" o vetor, comparando os elementos vizinhos entre si. Caso estejam fora de ordem, os mesmos trocam de posição entre si. Procede-se assim até o final do vetor. Na primeira "varredura" verifica-se que o último elemento do vetor já está no seu devido lugar (no caso de ordenação crescente, ele é o maior de todos). A segunda "varredura" é análoga à primeira e vai até o penúltimo elemento. Este processo é repetido até que seja feito um número de varreduras igual ao número de elementos a serem ordenados menos um. Ao final do processo o vetor está classificado segundo o critério escolhido.

O exemplo a seguir ilustra o algoritmo ***bubble sort*** para ordenar 50 número inteiros em ordem crescente:

```
Algoritmo Bubble_Sort
Var
numeros : matriz [1..50] de inteiros
aux, i, j: inteiro
Início
Para i de 1 até 50 faça
       Início
       Ler numeros[i]
       Fim
j := 50
Enquanto j > 1 faça
       Início
       Para i de 1 até j-1 faça
           Início
           Se numeros[i] > numeros[i+1] Então
                  Início
                  aux := numeros[i];
                  numeros[i] := numeros[j];
                  numeros[j] := aux;
                  Fim
           Fim
       j:=j-1;
       Fim
Escreva "vetor ordenado: "
Para i de 1 até 50 faça
       Início
       Escrever numeros[i]
       Fim
```

```
Fim.
```

Uma outra variação deste algoritmo, com as mesmas características utiliza duas construções **Para**, fazendo a comparação iniciando do primeiro elemento até o penúltimo, comparando com o imediatamente seguinte até o último elemento:

```
Algoritmo Variação_do_Bubble_Sort
Var
numeros : matriz [1..50] de inteiros
aux, i, j: inteiro
Início
Para i de 1 até 50 faça
       Início
       Ler numeros[i]
       Fim
Para i de 1 até 49 faça
       Início
       Para j de i + 1 até 50 faça
           Início
           Se numeros[i] > numeros[j] Então
                  Início
                  aux := numeros[i];
                  numeros[i] := numeros[j];
                  numeros[j] := aux;
                  Fim
           Fim
       Fim
Escreva "vetor ordenado: "
Para i de 1 até 50 faça
       Início
       Escrever numeros[i]
       Fim
Fim.
```

Podemos observar também que para ordenar o vetor em ordem decrescente basta inverter o sinal de comparação no teste da condição lógica **Se numeros[i]** > **numeros[j]**, para:

Se numeros[i] < numeros[j]

8.2 Matrizes com Mais de Uma Dimensão

Este tipo de estrutura também tem sua principal utilização vinculada à criação de tabelas. Caracteriza-se por ser definida uma única variável vinculada dimensionada com um determinado tamanho. A dimensão de uma matriz é constituída por constantes inteiras e positivas. Os nomes dados às matrizes seguem as mesmas regras de nomes utilizados para indicar as variáveis simples.

A sintaxe do comando de definição de matrizes de duas dimensões é a seguinte:

Var
<nome_da_variável>* : MATRIZ [*<linha_inicial>* .. *<linha_final>* , *<coluna_inicial>* .. *<coluna_final>*] DE *<tipo_de_dado>

Ex.: **VAR**
M : **MATRIZ** [1 .. 5 , 1 .. 10] **DE INTEIRO**

Também é possível definir matrizes com várias dimensões, por exemplo:

Ex.: **VAR**
N : **MATRIZ** [1 .. 4] **DE INTEIRO**
O : **MATRIZ** [1 .. 50 , 1 .. 4] **DE INTEIRO**
P : **MATRIZ** [1 .. 5, 1 .. 50 , 1 .. 4] **DE INTEIRO**
Q : **MATRIZ** [1 .. 3 , 1 .. 5, 1 .. 50 , 1 .. 4] **DE INTEIRO**
R : **MATRIZ** [1 .. 2 , 1 .. 3 , 1 .. 5, 1 .. 50 , 1 .. 4] **DE INTEIRO**
S : **MATRIZ** [1 .. 2 , 1 .. 2 , 1 .. 3 , 1 .. 5, 1 .. 50 , 1 .. 4] **DE INTEIRO**

A utilidade de matrizes desta forma é muito grande. No exemplo acima, cada matriz pode ser utilizada para armazenar uma quantidade maior de informações:

- a matriz **N** pode ser utilizada para armazenar 4 notas de um aluno
- a matriz **O** pode ser utilizada para armazenar 4 notas de 50 alunos.
- a matriz **P** pode ser utilizada para armazenar 4 notas de 50 alunos de 5 disciplinas.
- a matriz **Q** pode ser utilizada para armazenar 4 notas de 50 alunos de 5 disciplinas, de 3 turmas.
- a matriz **R** pode ser utilizada para armazenar 4 notas de 50 alunos de 5 disciplinas, de 3 turmas, de 2 colégios.
- a matriz **S** pode ser utilizada para armazenar 4 notas de 50 alunos de 5 disciplinas, de 3 turmas, de 2 colégios, de 2 cidades

8.2.1 Operações Básicas com Matrizes de Duas Dimensões

Do mesmo modo que acontece com os vetores, não é possível operar diretamente com o conjunto completo, mas com cada um de seus componentes isoladamente.

O acesso individual a cada componente de uma matriz é realizado pela especificação de sua posição na mesma por meio do seu índice. No exemplo anterior foi definida uma variável **M** capaz de armazenar 10 número inteiros em cada uma das 5 linhas. Para acessar um elemento desta matriz deve-se fornecer o nome da mesma e o índice da linha e da coluna do componente desejado da matriz (um número de 1 a 5 para a linha e um número de 1 a 10 para a coluna, neste caso).

Por exemplo, **M[1,1]** indica o primeiro elemento da primeira linha da matriz, **M[1,2]** indica o segundo elemento da primeira linha da matriz, **M[1,10]** indica o último elemento da primeira linha da matriz e **M[5,10]** indica o último elemento da última linha da matriz

Da mesma forma como vetores, não é possível operar diretamente sobre matrizes como um todo, mas apenas sobre seus componentes, um por vez. Por exemplo, para somar duas matrizes é necessário somar cada um de seus componentes dois a dois. Da mesma forma as operações de atribuição, leitura e escrita de matrizes devem ser feitas elemento a elemento.

8.2.1.1 Atribuição de Uma Matriz de Duas Dimensões

Na atribuição de matrizes, da mesma forma que nos vetores, além do nome da variável deve-se necessariamente fornecer também o índice do componente da matriz onde será armazenado o resultado da avaliação da expressão. O índice referente ao elemento é composto por tantas informações quanto o número de dimensões da matriz. No caso de ter duas dimensões, o primeiro número se refere à linha e o segundo número se refere à coluna da matriz em que se encontra a informação

```
Ex.:   M[1,1]  := 15
       M[1,10] := 10
       M[3,5]  := 20
       M[5,10] := 35
```

8.2.1.2 Leitura de Dados de Uma Matriz de Duas Dimensões

A leitura de uma matriz é feita passo a passo, um de seus componentes por vez, usando a mesma sintaxe da instrução primitiva da entrada de dados, onde além do nome da variável, deve ser explicitada a posição do componente lido:

LEIA ***<nome_da_variável>*** **[** ***<índice_1>, ... , < índice_n>*** **]**

Uma observação importante a ser feita é a utilização de construções **Para** aninhadas ou encadeada a fim de efetuar a operação de leitura repetidas vezes, em cada uma delas lendo um determinado componente da matriz. Esta construção é muito comum quando se opera com matrizes, devido à necessidade de se realizar uma mesma operação com os diversos componentes das mesmas.

O algoritmo a seguir exemplifica a operação de leitura de uma matriz:

```
Algoritmo exemplo_leitura_de_matriz
Var
numeros : matriz[1..5, 1..10] de inteiro
i, j : inteiro
Início
Para i de 1 até 5 faça
       Início
       Para j de 1 até 10 faça
           Início
           Leia numeros[i,j]
           Fim
       Fim
Fim.
```

8.2.1.3 Escrita de Dados de Uma Matriz de Duas Dimensões

A escrita de uma matriz obedece à mesma sintaxe da instrução primitiva de saída de dados e também vale lembrar que, da mesma forma que com vetores, além do nome da matriz, deve-se também especificar por meio do índice o componente a ser escrito:

ESCREVA <***nome_da_variável***> **[** <***índice_1***> **, ... ,** <***índice_n***> **]**

O algoritmo a seguir exemplifica a operação de leitura e escrita de uma matriz, utilizando as construções **Para** aninhadas ou encadeadas:

```
Algoritmo exemplo_escrita_de_matriz
Var
numeros : matriz[1..5,1..10] de inteiro
i, j : inteiro
Início
Para i de 1 até 5 faça
       Início
       Para i de 1 até 10 faça
           Início
           Leia numeros[i,j]
           Fim
       Fim
Para i de 1 até 5 faça
       Início
       Para j de 1 até 10 faça
           Início
           Escreva numeros[i,j]
           Fim
       Fim
Fim.
```

Um exemplo mais interessante é mostrado a seguir, onde uma matriz de 5 linhas por 10 colunas é lida e guardada na matriz **numeros**. A seguir é efetuada e escrita a soma dos elementos da 2ª linha e também a soma dos elementos da 3ª coluna

```
Algoritmo exemplo_escrita_de_matriz_com_soma
Var
numeros : matriz[1..5,1..10] de inteiro
i, j : inteiro
somal2, somac3 : inteiro
Início
```

```
Para i de 1 até 5 faça
       Início
       Para i de 1 até 10 faça
           Início
           Leia numeros[i,j]
           Fim
       Fim
Para i de 1 até 5 faça
       Início
       Para i de 1 até 10 faça
           Início
           Escreva numeros[i,j]
           Fim
       Fim
somal2 := 0
somac3 := 0
Para j de 1 até 10 faça
       Início
       somal2 := somal2 + numeros[2,j]
       Fim
Para i de 1 até 5 faça
       Início
       somac3 := somac3 + numeros[i,3]
       Fim
Escrever “Soma Linha 2 = ”, somal2
Escrever “Soma Coluna 3 = ”, somac3
Fim.
```

9. SUBALGORITMOS

A complexidade dos algoritmos está intimamente ligada à da aplicação a que se destinam. Em geral, problemas complicados exigem algoritmos extensos para sua solução.

Sempre é possível dividir problemas grandes e complicados em problemas menores e de solução mais simples. Assim, pode-se solucionar cada um destes pequenos problemas separadamente, criando algoritmos para tal (subalgoritmos). Posteriormente, pela justaposição destes subalgoritmos elabora-se "automaticamente" um algoritmo mais complexo e que soluciona o problema original. Esta metodologia de trabalho é conhecida como **Método de Refinamentos Sucessivos**, cujo estudo é assunto de cursos avançados sobre técnicas de programação.

Um **subalgoritmo** é um nome dado a um trecho de um algoritmo mais complexo e que, em geral, encerra em si próprio um pedaço da solução de um problema maior – o algoritmo a que ele está subordinado. Este conceito é essencial numa ciência bastante recente: a Engenharia de Software.

Em resumo, os subalgoritmos são importantes na:

- subdivisão de algoritmos complexos, facilitando o seu entendimento;
- estruturação de algoritmos, facilitando principalmente a detecção de erros e a documentação de sistemas; e
- modularização de sistemas, que facilita a manutenção de softwares e a reutilização de subalgoritmos já implementados.

A idéia da reutilização de software tem sido adotada por muitos grupos de desenvolvimento de sistemas de computador, devido à economia de tempo e trabalho que proporcionam. Seu princípio é o seguinte: um conjunto de algoritmos destinado a solucionar uma série de tarefas bastante corriqueiras é desenvolvido e vai sendo aumentado com o passar do tempo, com o acréscimo de novos algoritmos. A este conjunto dá-se o nome de **biblioteca**. No desenvolvimento de novos sistemas, procura-se ao máximo basear sua concepção em subalgoritmos já existentes na biblioteca, de modo que a quantidade de software realmente novo que deve ser desenvolvido é minimizada.

Muitas vezes os subalgoritmos podem ser úteis para encerrar em si uma certa seqüência de comandos que é repetida várias vezes num algoritmo. Nestes casos, os subalgoritmos proporcionam uma diminuição do tamanho de algoritmos maiores. Antigamente, esta propriedade era tida como a principal utilidade dos subalgoritmos.

9.1 Mecanismo de Funcionamento

Um algoritmo completo é dividido num **algoritmo principal** e diversos **subalgoritmos** (tantos quantos forem necessários e/ou convenientes). O **algoritmo principal** é aquele por onde

a execução do algoritmo sempre se inicia. Este pode eventualmente invocar os demais subalgoritmos.

O corpo do algoritmo principal é sempre o último trecho do pseudocódigo de um algoritmo. As definições dos subalgoritmos estão sempre colocadas no trecho após a definição das variáveis globais e antes do corpo do algoritmo principal:

ALGORITMO ***<nome do algoritmo>***

Var ***<definição das variáveis globais>***

<definições dos subalgoritmos>

Inicio

<corpo do algoritmo principal>

Fim.

Durante a execução do algoritmo principal, quando se encontra um comando de invocação de um subalgoritmo, a execução do mesmo é interrompida. A seguir, passa-se à execução dos comandos do corpo do subalgoritmo. Ao seu término, retoma-se a execução do algoritmo que o chamou (no caso, o algoritmo principal) no ponto onde foi interrompida (comando de chamada do subalgoritmo) e prossegue-se pela instrução imediatamente seguinte.

Note, também, que é possível que um subalgoritmo chame outro através do mesmo mecanismo.

9.2 Definição de Subalgoritmos

A definição de um subalgoritmo consta de:

- um **cabeçalho**, onde estão definidos o **nome** e o **tipo** do subalgoritmo, bem como os seus **parâmetros** e **variáveis locais**;

- um **corpo**, onde se encontram as instruções (comandos) do subalgoritmo.

O **nome** de um subalgoritmo é o nome simbólico pelo qual ele é chamado por outro algoritmo.

O **corpo** do subalgoritmo contém as instruções que são executadas cada vez que ele é invocado.

Variáveis locais são aquelas definidas dentro do próprio subalgoritmo e só podem ser utilizadas pelo mesmo.

Parâmetros são canais por onde os dados são transferidos pelo algoritmo chamador a um subalgoritmo, e vice-versa. Para que possa iniciar a execução das instruções em seu corpo, um subalgoritmo às vezes precisa receber dados do algoritmo que o chamou e, ao terminar sua tarefa, o subalgoritmo deve fornecer ao algoritmo chamador os resultados da mesma. Esta comunicação bidirecional pode ser feita de dois modos que serão estudados mais à frente: por meio de **variáveis globais** ou por meio da **passagem de parâmetros**.

O **tipo** de um subalgoritmo é definido em função do número de valores que o subalgoritmo retorna ao algoritmo que o chamou. Segundo esta classificação, os algoritmos podem ser de dois tipos:

- **funções**, que retornam um, e somente um, valor ao algoritmo chamador;
- **procedimentos**, que retornam zero (nenhum) ou mais valores ao algoritmo chamador.

Na realidade, a tarefa desempenhada por um subalgoritmo do tipo **função** pode perfeitamente ser feita por outro do tipo **procedimento** (o primeiro é um caso particular deste). Esta diferenciação é feita por razões históricas, ou, então, pelo grande número de subalgoritmos que se encaixam na categoria de **funções**.

9.3 Funções

O conceito de **Função** é originário da idéia de função matemática (por exemplo, raiz quadrada, seno, cosseno, tangente, logaritmo, entre outras), onde um valor é calculado a partir de outro(s) fornecido(s) à função.

A sintaxe da definição de uma função é dada a seguir:

FUNÇÃO <***nome***> **(** <***parâmetros***> **)** <***tipo_de_dado***>

VAR <***variáveis locais***>

INÍCIO

<***comando composto***>

FIM

Temos que:

- <***nome***> é o nome simbólico pelo qual a função é invocada por outros algoritmos;
- <***parâmetros***> são os parâmetros da função;
- <***tipo de dado***> é o tipo de dado da informação retornado pela função ao algoritmo chamador;
- <***variáveis locais***> consiste na definição das variáveis locais à função. Sua forma é análoga à da definição de variáveis num algoritmo;
- <***comando composto***> é o conjunto de instruções do corpo da função.

Dentro de um algoritmo, o comando de invocação de um subalgoritmo do tipo função sempre aparece dentro de uma expressão do mesmo tipo que o do valor retornado pela função.

A invocação de uma função é feita pelo simples aparecimento do nome da mesma, seguido pelos respectivos parâmetros entre parênteses, dentro de uma expressão. A função é executada e, ao seu término, o trecho do comando que a invocou é substituído pelo valor

retornado pela mesma dentro da expressão em que se encontra, e a avaliação desta prossegue normalmente.

Dentro de uma função, e somente neste caso, o comando **Retorne <*expressão*>** é usado para retornar o valor calculado pela mesma. Ao encontrar este comando, a expressão entre parênteses é avaliada, a execução da função é terminada neste ponto e o valor da expressão é retornado ao algoritmo chamador. Vale lembrar que uma expressão pode ser uma simples constante, uma variável ou uma combinação das duas por meio de operadores. Esta expressão deve ser do mesmo tipo que o valor retornado pela função.

O algoritmo a seguir é um exemplo do emprego de função para calcular o valor de um número elevado ao quadrado.

```
Algoritmo Exemplo_de_função
Var X, Y : real

Função Quad(w : real) : real
Var Z : real
Inicio
       Z := w * w
Retorne Z
Fim

Início
Escreva "Digite um número”
Leia X
Y := Quad(X)
Escreva X, " elevado ao quadrado é = ", Y
Fim.
```

Do exemplo anterior é importante notar que:

– a função **QUAD** toma W como parâmetro do tipo real, retorna um valor do tipo real e possui Z como uma variável local real;

– o comando de invocação da função **QUAD** aparece no meio de uma expressão, no comando de atribuição dentro do algoritmo principal.

9.4 Procedimentos

Um procedimento é um subalgoritmo que retorna zero (nenhum) ou mais valores ao (sub)algoritmo chamador. Estes valores são sempre retornados por meio dos parâmetros ou de variáveis globais, mas nunca explicitamente, como no caso de funções. Portanto, a chamada de um procedimento nunca surge no meio de expressões, como no caso de funções. Pelo contrário, a chamada de procedimentos só é feita em comandos isolados dentro de um algoritmo, como as instruções de entrada (Leia) e saída (Escreva) de dados.

A sintaxe da definição de um procedimento é:

PROCEDIMENTO <***nome***> **(** <***parâmetros***> **)**

Var <***variáveis locais***>

Inicio

<***comando composto***>

Fim.

Temos que:

- <***nome***> é o nome simbólico pelo qual o procedimento é invocado por outros algoritmos;
- <***parâmetros***> são os parâmetros do procedimento;
- <***variáveis locais***> são as definições das variáveis locais ao procedimento. Sua forma é análoga à da definição de variáveis num algoritmo;
- <***comando composto***> é o conjunto de instruções do corpo do procedimento, que é executado toda vez que o mesmo é invocado.

O exemplo a seguir é um exemplo simples, onde um procedimento é usado para escrever o valor das componentes de um vetor.

```
Algoritmo Exemplo_procedimento

Var vet : matriz[1..10] de real

Procedimento ESC_VETOR()
Var i : inteiro
Inicio
Para i de 1 até 10 faça
       Início
       Escreva vet[i]
       Fim
Fim

Procedimento LER_VETOR()
Var i : inteiro
Inicio
Para i de 1 até 10 faça
       Início
       Leia vet[i]
```

```
        Fim
Fim

Inicio
LER_VETOR()
ESC_VETOR()
Fim.
```

No exemplo é conveniente observar:

– a forma de definição dos procedimentos LER_VETOR( ) e ESC_VETOR( ), que não possuem nenhum parâmetro, e usam a variável local **i** para "varrer" os componentes do vetor, lendo e escrevendo um valor por vez; e

– a forma de invocação dos procedimentos, por meio do seu nome, seguido de seus eventuais parâmetros (no caso, nenhum), num comando isolado dentro do algoritmo principal.

9.5 Variáveis Globais e Locais

Variáveis globais são aquelas declaradas no início de um algoritmo. Estas variáveis são **visíveis** (isto é, podem ser usadas) no algoritmo principal e por todos os demais subalgoritmos.

Variáveis locais são aquelas definidas dentro de um subalgoritmo e, portanto, somente **visíveis** (utilizáveis) dentro do mesmo. Outros subalgoritmos, ou mesmo o algoritmo principal, não podem utilizá-las.

No exemplo a seguir são aplicados estes conceitos.

```
Algoritmo Exemplo_variáveis_locais_e_globais
Var X : real
I : inteiro

Função FUNC() : real
Var X : matriz[1..5] de inteiro
Y : caracter[10]
Inicio
...
Fim

Procedimento PROC
Var Y : lógico
Inicio
...
X := 4 * X
I := I + 1
...
Fim

Início
...
X := 3.5
...
Fim.
```

É importante notar no exemplo anterior que:

- as variáveis **X** e **I** são globais e visíveis a todos os subalgoritmos, à exceção da função **FUNC**, que redefine a variável **X** localmente;

- as variáveis **X** e **Y** locais ao procedimento **FUNC** não são visíveis ao algoritmo principal ou ao procedimento **PROC**. A redefinição local do nome simbólico **X** como uma matriz[5] de inteiro sobrepõe (somente dentro da função **FUNC**) a definição global de **X** como uma variável do tipo real;

- a variável **Y** dentro do procedimento **PROC**, que é diferente daquela definida dentro da função **FUNC**, é invisível fora deste procedimento;

- a instrução **X := 8.5** no algoritmo principal, bem como as instruções **X := 4 * X** e **I := I + 1** dentro do procedimento **PROC**, atuam sobre as variáveis globais **X** e **I**.

9.6 Parâmetros

Parâmetros são canais pelos quais se estabelece uma comunicação bidirecional entre um subalgoritmo e o algoritmo chamador (o algoritmo principal ou outro subalgoritmo). Dados são passados pelo algoritmo chamador ao subalgoritmo, ou retornados por este ao primeiro por meio de parâmetros.

Parâmetros formais são os nomes simbólicos introduzidos no cabeçalho de subalgoritmos, usados na definição dos parâmetros do mesmo. Dentro de um subalgoritmo trabalha-se com estes nomes da mesma forma como se trabalha com variáveis locais ou globais.

```
Função Média(X, Y : real) : real
Inicio
Retorne (X + Y) / 2
Fim
```

No exemplo anterior, **X** e **Y** são parâmetros formais da função **Média**.

Parâmetros reais são aqueles que substituem os parâmetros formais quando da chamada de um subalgoritmo. Por exemplo, o trecho seguinte de um algoritmo invoca a função Média com os parâmetros reais 8 e 7 substituindo os parâmetros formais X e Y.

```
Z := Média(8, 7)
```

Assim, os parâmetros formais são úteis somente na definição (formalização) do subalgoritmo, ao passo que os parâmetros reais substituem-nos a cada invocação do subalgoritmo. Note que os parâmetros reais podem ser diferentes a cada invocação de um subalgoritmo.

9.7 Mecanismos de Passagem de Parâmetros

Como foi visto anteriormente, os parâmetros reais substituem os formais no ato da invocação de um subalgoritmo. Esta substituição é denominada passagem de parâmetros e pode se dar segundo dois mecanismos distintos: passagem por valor (ou por cópia) ou passagem por referência.

9.7.1 Passagem de Parâmetros por Valor

Na passagem de parâmetros por valor (ou por cópia) o parâmetro real é calculado e uma cópia de seu valor é fornecida ao parâmetro formal, no ato da invocação do subalgoritmo. A execução do subalgoritmo prossegue normalmente e todas as modificações feitas no parâmetro formal não afetam o parâmetro real, pois trabalha-se apenas com uma cópia do mesmo.

```
Algoritmo Exemplo_parametro_por_valor
Var X : inteiro

Procedimento PROC(Y : inteiro)
Início
Y := Y + 1
Escreva "Durante Y = ”, Y
Fim

Início
X := 1
Escreva "Antes X = ”, X
PROC(X)
Escreva "Depois X = ", X
Fim.
```

O algoritmo anterior fornece o seguinte resultado:

Antes X = 1
Durante Y = 2
Depois X = 1

Isto certifica que o procedimento não alterou o valor do parâmetro real X durante sua execução.

Este tipo de ação é possível porque, neste mecanismo de passagem de parâmetros, é feita uma reserva de espaço em memória para os parâmetros formais, para que neles seja armazenada uma cópia dos parâmetros reais.

9.7.2 Passagem de Parâmetros por Referência

Neste mecanismo de passagem de parâmetros não é feita uma reserva de espaço em memória para os parâmetros formais. Quando um subalgoritmo com parâmetros passados por referência é chamado, o espaço de memória ocupado pelos parâmetros reais é compartilhado pelos parâmetros formais correspondentes. Assim, as eventuais modificações feitas nos parâmetros formais também afetam os parâmetros reais correspondentes.

Um mesmo subalgoritmo pode utilizar diferentes mecanismos de passagem de parâmetros, para parâmetros distintos. Para diferenciar uns dos outros, convencionou-se colocar o prefixo **VAR** antes da definição dos parâmetros formais passados por referência. Se por exemplo um procedimento tem o seguinte cabeçalho:

```
Procedimento PROC( X, Y : inteiro; Var Z : real; J: real)
```

Então:

- **X** e **Y** são parâmetros formais do tipo inteiro e são passados por valor;
- **Z** é um parâmetro formal do tipo real passado por referência;
- **J** é um parâmetro formal do tipo real passado por valor.

O exemplo do item anterior, alterado para que o parâmetro **Y** do procedimento seja passado por referência, torna-se:

```
Algoritmo Exemplo_parametro_por_referencia
Var X : inteiro

Procedimento PROC(Y : inteiro)
Início
Y := Y + 1
Escreva "Durante Y = ”, Y
Fim

Início
X := 1
Escreva "Antes X = ”, X
PROC(X)
Escreva "Depois X = ", X
Fim.
```

O resultado do algoritmo modificado é:

Antes X = 1
Durante Y = 2
Depois X = 2

Como podemos observar, depois de chamar o procedimento com o parâmetro por referência o valor original da variável foi alterado.

9.8 Refinamentos Sucessivos

De acordo com o modelo cascata, o ciclo de vida de um software consta das seguintes fases: Análise de requisitos, Arquitetura geral, Projeto detalhado, Programação, Integração e testes, e manutenção.

O **Método de Refinamentos Sucessivos** para a elaboração de algoritmos dita que um dado problema deve ser subdividido em problemas menores repetidamente, até que seja possível encontrar um algoritmo (subalgoritmo ou comando composto) para resolver cada um destes

subproblemas. Assim, o algoritmo para resolver o problema original será composto pela composição destes algoritmos.

O método apresentado pode ser subdividido na fase de **Análise Top-Down**, que procura resolver o problema a nível de subalgoritmos e comandos compostos, e na fase de **Programação Estruturada**, que detalha os subalgoritmos e comandos compostos até o nível das instruções primitivas e construções de controle de fluxo.

Por fim, o uso do método é vantajoso devido ao seu estímulo à modularização de sistemas, que proporciona a criação de programas claros, fáceis de entender e, portanto, de manutenção mais barata.

10. BIBLIOGRAFIA

CORMEN, Thomas H. & LEISERSON, Charles E. & RIVEST, Ronald L. **Introduction to Algorithms**. New York. McGraw-Hill, 1990.

FARRER, Harry et alli **Algoritmos Estruturados**. Rio de Janeiro. Editora Guanabara Koogan S.A, 1989. 252p.

GOTTFRIED, Byron S. **Programação em Pascal**. Lisboa. McGraw Hill, 1994. 567p.

MANZANO, José Augusto N. G. & OLIVEIRA, Jayr Figueiredo. **Algoritmos: Lógica Para Desenvolvimento de Programação**. São Paulo. Érica, 1996. 270p.

MECLER, Ian & MAIA, Luiz Paulo. **Programação e Lógica com Turbo Pascal**. Rio de Janeiro, Campus, 1989. 223p.

ORTH, Afonso Inácio. **Algoritmos**. Porto Alegre. Editora Pallotti, 1985. 130p.

SALIBA, Walter Luís Caram. **Técnicas de Programação: Uma Abordagem Estrutura**. São Paulo. Makron, McGraw-Hill, 1992. 141p.