Notas de Aula de
Algoritmos e Programação de Computadores

FLÁVIO KEIDI MIYAZAWA

com a colaboração de

TOMASZ KOWALTOWSKI

Instituto de Computação - UNICAMP

Versão 2000.2

Instituto de Computação
UNICAMP
Caixa Postal 6176
13083–970 Campinas–SP
{fkm,tomasz}@ic.unicamp.br

Sumário

1 Introdução à Computação

1.1 Organização do Computador

Um computador é uma coleção de componentes que realizam operações lógicas e aritméticas sobre um grande volume de dados. Na figura 1 apresentamos uma organização básica em um computador seqüêncial.

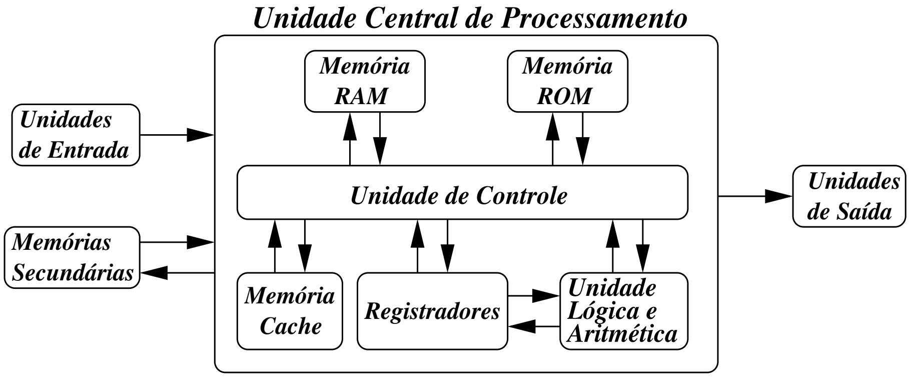


Figura 1: Organização Básica de um Computador Seqüêncial.

A seguir descreveremos cada uma destas partes.

Unidade de Entrada São os componentes que permitem a entrada de informações exteriores para serem processadas pelo computador. Exemplo: teclado, mouse, câmera de vídeo, etc.

Unidade de Saída São os componentes que permitem a apresentações de informações processadas para o meio externo. Exemplo: monitor, impressora, etc.

Unidade Central de Processamento Também conhecida como CPU *(Central Processing Unit)*. É responsável pela execução dos programas e pelo comportamento das outras unidades no sistema. É capaz de fazer contas matemáticas e fazer decisões simples. As principais partes da CPU são: a Unidade Lógica e Aritmética, Unidade de Controle e Memórias (Registradores, Memória Principal (ou Memória RAM), Memória ROM e Cache).

Unidade Lógica e Aritmética Parte da CPU que realiza operações aritméticas (soma, subtração, multiplicação, divisão, resto, troca de sinal, etc) e operações lógicas (*and, or, not, xor,* etc).

Memória Principal É usado na CPU para manter instruções e dados. Também conhecido como Memória RAM *(Random Access Memory)*. A recuperação dos dados é feita através de circuitos lógicos e por isso é rápida. Não é tão grande, já que depende muito da tecnologia de integração destes circuitos. É uma memória volátil, i.e., quando o computador é desligado, todos os dados nesta memória se perdem.

Memória ROM ROM *(Read Only Memory)* é uma memória que contém dados e códigos de execução que não podem ser alterados. Uma das aplicações desta memória é manter código de execução para a leitura e execução de um sistema operacional.

Memória Cache Memória rápida projetada para guardar dados que foram recentemente acessados. Para buscar um certo dado na memória RAM, é considerado se este pode estar na memória cache, e em caso positivo a busca na memória RAM é interrompido e este é recuperado diretamente da memória cache. Tem tempo de acesso mais rápido que a memória RAM.

Registradores Memórias de alta velocidade ligada a operações de cálculos lógicos e aritméticos. Em geral em quantidade e tamanhos pequenos.

Unidade de Controle Parte da CPU que busca na memória a próxima instrução e a decodifica para ser executada. Dependendo da instrução, pode-se ter uma transferência do controle para a unidade lógica e aritmética ou o envio de dados para os componentes externos à CPU.

Memória Secundária Memória para armazenamento a longo prazo. Os dados armazenados nesta memória não são perdidos quando se desliga o computador. Em geral de dimensões maiores que a Memória RAM mas de acesso mais lento, já que envolvem o uso de dispositivos mecânicos. Ex. Discos rígidos, disquetes, fitas magnéticas, etc.

Podemos ver que há diversos tipos de memórias em um computador. Cada uma destas memórias usa tecnologia que reflete no custo, na velocidade de acesso e na quantidade de armazenamento. A seguinte ordem apresenta algumas memórias ordenadas, de maneira crescente, pela quantidade de armazenamento:

Registrador – Memória Cache – Memória RAM – Discos Rígidos.

Esta mesma ordem apresenta o custo relativo e a velocidade de maneira decrescente.

1.2 Alguns Termos Técnicos

Hardware Componentes mecânicos e eletro- eletrônicos que compõem o computador. Parte *dura* do computador.

Software Seqüência de instruções e comandos que fazem o computador realizar determinada tarefa., também chamados de *programas de computador*. Devem estar armazenados em algum tipo de memória.

Periférico É qualquer componente do computador *(hardware)* que não seja a CPU. Exemplos: leitoras de disquete, monitores, teclados, vídeo, impressoras, etc.

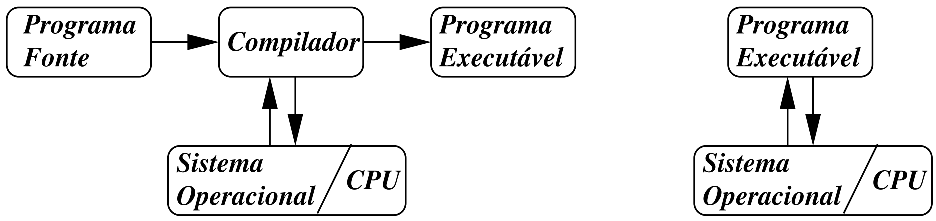
Sistema Operacional Coleção de programas que gerencia e aloca recursos de hardware e software. Exemplos de tarefas que um sistema operacional realiza são: leitura de dados pelo teclado, impressão de informações no vídeo, gerenciamento da execução de vários programas pela CPU, gerenciamento da memória principal e da memória secundária para uso dos programas em execução, etc. Exemplos: Linux, Unix, Windows98, OS2, MS-DOS, etc.

Linguagem de Máquina Conjunto de instruções que podem ser interpretados e executados diretamente pela CPU de um dado computador. É específica para cada computador.

Linguagem Assembler *(Linguagem de Baixo Nível)* Representação da linguagem de máguina através de códigos mnemônicos. Também é específica de cada máquina.

Linguagem de alto nível Linguagem que independe do conjunto de instruções da linguagem de máquina do computador. Cada instrução de alto nível equivale a várias instruções da linguagem de máquina, sendo assim mais produtiva. Ex.: Pascal, C, Algol, BASIC, Lisp, Prolog, etc.

Compilador Tradutor de programas escritos em uma linguagem de programação para programas em linguagem de máquina. Uma vez que o programa foi convertido para código de máquina, este pode ser executado independente do compilador e do programa original. Veja a figura 2.



(a) Geração do Programa Executável ***(b) Execução do Programa***

Figura 2: Etapas para execução de um programa compilado.

Interpretador É um programa que executa outros programas escritos em alguma linguagem de programação. A execução de um programa interpretado é em geral mais lenta que o programa compilado. Por outro lado, o uso de programas interpretados permite que trechos de códigos possam ser trocados por novos facilmente, fazendo com que o programa fonte possa mudar durante sua execução. Este é um dos grandes motivos de se usar programas interpretados em sistemas especialistas. Duas linguagens para as quais podemos encontrar interpretadores são Lisp e Prolog. Veja a figura 3.

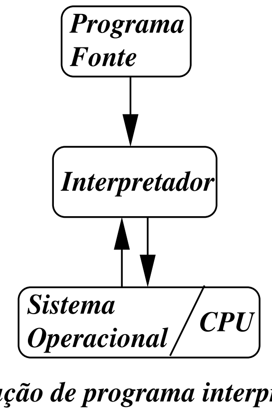


Figura 3: Execução de um programa interpretado.

Algoritmo É a descrição de uma seqüência de ações para realizar alguma tarefa. Em geral, estaremos interessados em algoritmos computacionais, que descrevem uma seqüência de ações que podem ser traduzidos para alguma linguagem de programação.

Uma maneira para calcular o MDC (Máximo Divisor Comum) de dois números inteiros positivos m e n é através do Algoritmo de Euclides apresentado no quadro seguinte.

Passo 1:	Adote $x = m$ e $y = n$;
Passo 2:	Adote $r =$(resto de x dividido por y);
Passo 3:	Adote novos valores $x = y$ e $y = r$;
Passo 4:	Se r é diferente de 0, volte ao passo 2; senão pare com a resposta x.

Algoritmo de Euclides

O seguinte programa apresenta uma versão mais estilizada:

Passo1:	Dados: m e n.
Passo2:	$x \leftarrow m$.
Passo3:	$y \leftarrow n$.
Passo4:	Repita
Passo4.1:	$\quad r \leftarrow \mathrm{resto}(x, y)$;
Passo4.2:	$\quad x \leftarrow y$;
Passo4.3:	$\quad y \leftarrow r$;
Passo4.4:	Até que $r = 0$.
Passo5:	Imprima o resultado x.

Algoritmo de Euclides Estilizado.

O seguinte programa apresenta uma versão na linguagem Pascal:

```
Program Euclides;
var x, y, r, m, n : integer;
begin
    Readln(m,n);
    x := m;
    y := n;
    repeat
        r := (x mod y);
        x := y;
        y := r;
    until r = 0;
    writeln(x);
end.
```

Implementação do Algoritmo de Euclides em Pascal

1.3 Bits e Bytes

A menor unidade de informação usada pelo computador é o *bit*. Este tem atribuições lógicas **0** ou **1**. Cada um destes estados pode, internamente, ser representado por meios eletro-magnéticos (negativo/positivo, ligado/desligado, etc). É por isso que é mais fácil para armazenar dados em formato binário. Assim, todos os dados do computador são representados de forma binária. Mesmo os números são comumente representados na base 2, em vez da base 10, e suas operações são feitas na base 2.

Um conjunto de 8 bits é chamado de *byte*. Um byte pode ter até $2^8 = 256$ configurações diferentes. O principal padrão usado para representar caracteres (*'a','b','c',...,'A','B','C',...,'!','@','#','$',...*) é o padrão ASCII (*American Standard Code for Information Interchange*), usada na maioria dos computadores. Cada um destes caracteres é representado por um byte. A tabela 1 apresenta o código de alguns caracteres no código ASCII: Observe que:

1. As codificações para letras em maiúsculas e minúsculas são diferentes.
2. A codificação de 'B' é a codificação de 'A' somado de 1; a codificação de 'C' é a codificação de 'B' somado de 1; assim por diante. Esta codificação permite poder comparar facilmente se um caracter vem antes do outro ou não. Internamente, verificar se o caracter 'a' vem antes do 'b', é verificar se o número binário correspondente a 'a' é menor que o número binário correspondente a 'b'.
3. As letras maiúsculas vem antes das minúsculas.

As seguintes denominações são comumente usadas na área de informática

nome	*memória*
bit	$\{0, 1\}$
byte	8 bits
kilobyte (kbyte)	2^{10} bytes (pouco mais de mil bytes ($2^{10} = 1024$))
megabyte	2^{20} bytes (pouco mais de um milhão de bytes)
gigabyte	2^{30} bytes (pouco mais de um bilhão de bytes)

Atualmente, configurações de computador com 64 megabytes de memória RAM, 4,2 gigabytes de disco rígido, disco flexivel de 1,44 megabytes são muito comuns no mercado. Certamente esta configuração já será considerada pequena dentro de um ou dois anos, devido ao contínuo avanço da tecnologia nesta área.

Vejamos alguns exemplos de quanto é esta memória. Uma página de um livro, armazenada em formato ASCII, tem em torno de 50 linhas e 80 caracteres por linha. Assim, um livro de 1000 páginas teria algo em torno de 4.000.000 de caracteres, que poderiam ser guardados em 4 megabytes. Assim, um disco rígido de 4,2 gigabytes poderia guardar

Caracter	Representação em ASCII	Valor na base decimal
$\vdots$	$\vdots$	$\vdots$
(	00101000	40
)	00101001	41
$\star$	00101010	42
+	00101011	43
$\vdots$	$\vdots$	$\vdots$
0	00110000	48
1	00110001	49
2	00110010	50
3	00110011	51
$\vdots$	$\vdots$	$\vdots$
A	01000001	65
B	01000010	66
C	01000011	67
D	01000100	68
$\vdots$	$\vdots$	$\vdots$
a	01100001	97
b	01100010	98
c	01100011	99
d	01100100	100
$\vdots$	$\vdots$	$\vdots$

Tabela 1:

em torno de 1.000 livros deste tipo. Isto aparenta uma quantidade bastante grande de dados. Por outro lado, a maioria das aplicações atuais está fazendo uso cada vez maior de imagens, gráficos e sons. Estas aplicações demandam muita memória. Por exemplo, se você quiser representar uma imagem de tamanho 1000×1000 pontos (10^6 pontos), cada ponto com uma cor entre 65000 cores possíveis (dois bytes por ponto), gastaremos algo como 2 megabytes para armazenar apenas uma imagem deste tipo. A quantidade de memória aumenta quando armazenamos filmes, que usam em torno de 30 imagens por segundo. Apesar do uso de métodos de compressão sobre estes tipos de dados a necessidade de grande quantidade de memória ainda é crucial para muitas aplicações.

1.4 Base Binária, Base Decimal, ...

Como vimos, é muito mais fácil armazenar os dados na base binária que na base decimal. Assim, muitas das operações usadas no computador são feitas na base binária.

Muito provavelmente, nós usamos a base decimal porque temos 10 dedos nas duas mãos. E se tivéssemos 8 dedos em vez de 10 ? Neste caso, provavelmente estaríamos usando a base *octal*. Bom, agora imagine que você tem apenas dois dedos. Neste raciocínio, usaremos o sistema binário !!

Primeiro, vamos lembrar o que representa o número 4027 na base decimal.

$$4 \cdot 10^3 + 0 * 10^2 + 2 * 10^1 + 7 * 10^0$$

Agora um número binário tem apenas os dígitos **0** e **1**. O número 100110_b no sistema binário representa o número:

$$1 * 2^5 + 0 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0$$

Isto nos dá o número

$$32 + 4 + 2 = 38$$

Assim o número 100110_b no sistema binário é igual ao número 38 no sistema decimal.

As operações aritméticas também podem ser feitas em binário. Por exemplo: Vamos somar o número acima (100110_b) com (111_b).

$$\begin{array}{r} 100110_b \\ +\ 111_b \\ \hline 101101_b \end{array}$$

Agora vamos conferir o resultado: $111_b = 4 + 2 + 1 = 7$. E $101101_b = 32 + 8 + 4 + 1 = 45$. De fato, este número está correto. Em decimal seria $38 + 7 = 45$.

Exercício 1.1 *Dado um número no sistema decimal, encontre uma maneira de escrevê-lo no sistema binário.*

Exercício 1.2 *Como seria a multiplicação de dois números no sistema binário ?*

Assim, em um byte (8 bits), é possível representar os números de 0 até 255 ($255 = 2^8 - 1$).

binário	decimal
00000000	0
00000001	1
00000010	2
00000011	3
00000100	4
⋮	⋮
11111110	254
11111111	255

Da mesma forma, em dois bytes (16 bits) é possível representar os números de 0 até $65535 = 2^{16} - 1$.

Muitas vezes um compilador (ou mesmo o próprio computador) tem uma estrutura para definir sinais (se negativo ou positivo).

Um exemplo disso é usar um bit para representar o sinal do número. Por exemplo, um número de 16 bits pode ter a representação interna com um bit para sinal e os outros 15 bits para o número propriamente dito. Neste exemplo, poderíamos representar números de $-(2^{15} - 1)$ até $2^{15} - 1$ (i.e., de -32767 até 32767). Note que neste exemplo o número 0 é representado duas vezes ($+0$ e -0). Na prática, as representações internas levam isto em consideração de forma que cada número é representado apenas uma vez, neste caso podendo ir de -32768 até 32767.

É interessante observar que números positivos não nulos que são potência de 2 são números que têm todos os bits iguais a 0, exceto um bit. Assim, a multiplicação de um número inteiro x por um inteiro positivo não nulo que é potência de 2 faz apenas um deslocamento dos bits de x de algumas casas. Por exemplo, a multiplicação de x por 8 (2^3) faz o deslocamento dos bits de x de 3 casas para a esquerda.

$$\begin{array}{llllllllllllll}
x = & & & & b_1 & b_2 & b_3 & b_4 & \ldots & b_{n-4} & b_{n-3} & b_{n-2} & b_{n-1} & b_n \\
2^3 = & & & & & & & & & & 1 & 0 & 0 & 0 \\
\hline
 & & & & 0 & 0 & 0 & 0 & \ldots & 0 & 0 & 0 & 0 & 0 \\
 & & & 0 & 0 & 0 & 0 & \ldots & 0 & 0 & 0 & 0 & 0 & \\
+ & & 0 & 0 & 0 & 0 & \ldots & 0 & 0 & 0 & 0 & 0 & & \\
 & b_1 & b_2 & b_3 & b_4 & \ldots & b_{n-4} & b_{n-3} & b_{n-2} & b_{n-1} & b_n & & & \\
\hline
x \cdot 2^3 = & b_1 & b_2 & b_3 & b_4 & \ldots & b_{n-4} & b_{n-3} & b_{n-2} & b_{n-1} & b_n & 0 & 0 & 0
\end{array}$$

Assim, muitos compiladores, ao encontrar a multiplicação de um inteiro por uma potência de 2, trocam esta multiplicação por um deslocamento de bits.

Quando a operação é para obter a parte inteira da divisão de um inteiro x por uma potência de dois, digamos 2^k, basta deslocar os bits de x de k casas para a direita, perdendo k bits que estão mais a direita.

Também é sabido que a multiplicação de inteiros em geral leva mais tempo que somas e deslocamento de bits. Assim, uma possível otimização feita por compiladores é trocar a multiplicação de um inteiro por somas e deslocamento de bits. Por exemplo, digamos que desejamos obter $x \cdot 10$. O inteiro 10 não é potência de 2, mas em vez de fazermos a multiplicação por 10, podemos reescrever esta multiplicação por $x \cdot (8+2)$. I.e., podemos fazer $x \cdot 2^3 + x \cdot 2^1$. Desta maneira trocamos uma multiplicação de um inteiro por dois deslocamentos e uma soma, o que em muitos computadores é feito de forma mais rápida que uma multiplicação direta. Obs.: É possível mostrar que podemos fazer a multiplicação $x \cdot c$, onde x é inteiro e c é uma constante inteira aplicando este método fazendo no máximo $\log_2(c)$ somas, onde c é a constante inteira a multiplicar.

Outro sistema muito usado na literatura é a base 16 (*hexadecimal*). Neste sistema temos 16 dígitos usados na seguinte ordem: 0_h, 1_h, 2_h, 3_h, 4_h, 5_h, 6_h, 7_h, 8_h, 9_h, A_h, B_h, C_h, D_h, E_h, F_h.

Assim, o número $(\mathrm{F}+1)_h$ é igual a 10_h (10_h em hexadecimal é igual a 16 no sistema decimal).

Exercício 1.3 *Quanto é* $A9B_h$ *em decimal ?*

1.5 Álgebra Booleana

Alguns comandos de programação estão estreitamente relacionados com um sistema de álgebra, chamado *álgebra de boole*, desenvolvido por George Boole. Neste tipo de álgebra podemos operar sobre proposições que podem ser verdadeiras ou falsas, resultando em um resultado que também é verdadeiro ou falso. Em 1930, Turing mostrou que três funções lógicas (e *(and)*, ou *(or)* e não *(not)*) são suficientes para representar estas proposições lógicas. Uma das principais vantagens deste tipo de álgebra é que ela pode ser implementada eficientemente através de componentes eletrônicos.

Usando as letras F como falso e V como verdadeiro, apresentamos na tabela 2 os valores para as funções *(and)*, *or* e *not*.

x	y	$(x$ *and* $y)$
V	V	V
V	F	F
F	V	F
F	F	F

x	y	$(x$ *or* $y)$
V	V	V
V	F	V
F	V	V
F	F	F

x	(*not* x)
V	F
F	V

Tabela 2: Funções booleanas **and**, **or** e **not**.

Com estas três funções podemos construir funções mais complexas. Por exemplo, considere variáveis booleanas x e y, e uma função booleana $f(x, y)$ que assume os valores conforme a tabela a seguir.

x	y	$f(x,y)$
V	V	F
V	F	V
F	V	V
F	F	F

Para construir a função $f(x, y)$, podemos considerar a tabela acima, com todas as entradas possíveis de x e y, e construir $f(x, y)$ como uma seqüência de cláusulas ligadas pela função *or*. Cada cláusula corresponde a uma entrada verdadeira para a função $f(x, y)$, feita com as funções *and* e *not*. No exemplo acima, a função $f(x, y)$ pode ser escrita como:

$$f(x, y) = ((x \ and \ (\ not \ y)) \ or \ ((\ not \ x) \ and \ y))$$

Exercício 1.4 *Construa uma fórmula booleana para a seguinte função* $g(x, y, z)$ *dada pela seguinte tabela:*

x	y	z	$f(x,y,z)$
V	V	V	F
V	V	F	V
V	F	V	F
V	F	F	V
F	V	V	V
F	V	F	V
F	F	V	F
F	F	F	F

Exercício 1.5 *Um aluno do curso de Algoritmos e Programação de Computadores vai de sua casa para a escola a pé. Considere x, y, z e t sentenças booleanas representando as condições em um dia que ele vai para a aula e $f(x, y, z, t)$ uma fórmula booleana representando a chegada do aluno à aula sem problemas.*

- x :=*(Está chovendo);*
- y :=*(Tenho guarda-chuva);*
- z :=*(Tenho carona);*
- t :=*(Estou atrasado);*
- $f(x, y, z, t)$:=*(Vou para a aula sem problemas).*

Construa uma fórmula booleana adequada para $f(x, y, z, t)$ e construa a tabela com os valores possíveis de x, y, z e t com os valores da fórmula $f(x, y, z, t)$.

2 Primeiros Programas em Pascal

Apesar da metodologia de fluxogramas ser antiga, ela ainda é muito usada para explicar o seqüência de instruções em programas e algoritmos. Há vários símbolos que fazem parte de um fluxograma. Para nós, este tipo de estrutura será importante apenas para representar a estrutura seqüencial dos algoritmos e programas de computador. Em um fluxograma, um *passo* ou *módulo* é representado por um retângulo. As *setas* indicam o próximo comando a ser executado. Um *losango* indica uma condição e conforme a condição seja satisfeita ou não, este pode levar a um de dois outros comandos.

Na figura 4 apresentamos alguns diagramas usados em fluxogramas.

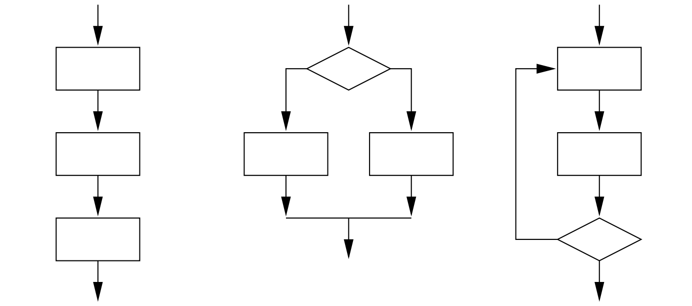


Figura 4: Exemplo de estruturas de controle usadas em programação estruturada.

Suponha que exista um curso cuja avaliação seja feita através de duas provas e um exame, sendo que o resultado final seja dado pela seguintes regras: A primeira prova tem peso 2 e a segunda prova tem peso 3. Seja N a média ponderada destas duas provas. Caso N seja pelo menos 5.0, a nota final do aluno, F, é igual a N. Caso contrário, o aluno deve fazer o exame, digamos com nota E, e sua nota final, é a média aritmética entre N e E. Por fim, caso a nota final do aluno seja pelo menos 5.0, o aluno está aprovado, caso contrário, o aluno está reprovado. A figura 5 apresenta um exemplo de fluxograma para se avaliar um aluno de tal curso.

No início dos tempos da programação de computadores, viu-se que programas que continham quaisquer desvios, de um comando para outro, eram muito mais difíceis de se entender. Os programas mais fáceis de entender eram aqueles onde o fluxograma não tinha setas se cruzando. Provavelmente é esta liberdade que faz com que o fluxograma deva ser considerado com muito cuidado e por isso mesmo esteja em desuso. Um algoritmo descrito como um fluxograma pode ter setas levando para qualquer lugar do programa, podendo o tornar muito confuso e sem nenhuma organização.

As linguagens estruturadas fornecem um número limitado de *estruturas de controle*. Tais estruturas de controle foram desenvolvidas de tal forma que o fluxo de execução não possa ser de qualquer maneira, mas sim de forma bem organizada. Uma representação de um programa que usa estas estruturas de controle permite uma tradução para fluxogramas sem que haja linhas se cruzando. Desta forma a manutenção de um programa fica menos difícil, mesmo que isso seja feito por outra pessoa que não aquela que o implementou inicialmente. A linguagem Pascal é uma linguagem de alto nível onde as técnicas de programação estruturadas são estimuladas através de seus comandos.

Um programa em Pascal tem o seguinte formato:

program nome;
    declarações {Área de Declarações}
begin
    Comandos {Corpo de Execução do Programa Principal}
end.

As palavras **program, begin** e **end** são *palavras-chave* da linguagem Pascal. São palavras reservadas que fazem parte da sintaxe da linguagem e não podem ser usadas para declarar os objetos definidos pelo programador. As palavras **begin** e **end** servem para definir um bloco de instruções, no caso, definem o corpo de execução do programa principal. Na área de declarações, são definidos os objetos que iremos usar no programa, que inclui objetos que representam outros trechos de código.

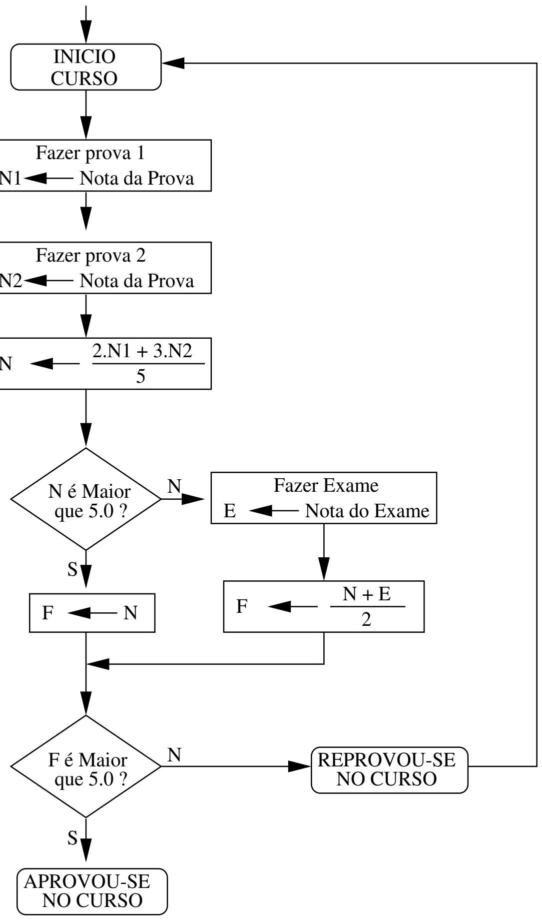


Figura 5: Fluxograma de avaliação de um curso.

Na figura 6 apresentamos um programa bem simples com apenas um comando de execução.

```
program BomDia;
begin
    writeln('Bom Dia!');
end.
```

Figura 6:

O Comando **writeln** na figura 6 escreve o texto 'Bom Dia!' no dispositivo de saída. Depois de escrever o texto, o programa ainda pula uma linha.

Existe também um outro comando, **write**, para imprimir um texto sem pular linha. Assim, um programa equivalente ao programa *BomDia* pode ser dado na figura 7:

```
program BomDia2;
begin
    write('Bom ');
    writeln('Dia!')
end.
```

Figura 7:

Note que há um espaço em branco depois de *Bom*, pois caso não tivesse, as palavras *Bom* e *Dia* seriam impressas juntas (*BomDia*).

```
program
BomDia1;
begin
write     (
'Bom '
)
;
writeln
(
'Dia!');
end.
```

Figura 8: Programa desorganizado

Observações

1. O **;** (ponto e vírgula) serve para separar comandos.
   Obs.: Dois ponto e vírgula seguidos separam um comando vazio.
2. A execução do programa está definida pelos comandos entre **begin** e **end**. Note que logo após a palavra **end** segue um **.** (ponto final).
3. Comandos que são separados por espaços em branco não fazem diferença. Na verdade, cada item do programa pode ser separado: Assim, o programa BomDia poderia ter sido escrito como dado na figura 8. É claro que o programa fica bem mais legível na primeira forma.

2.1 Comentários

Dentro de um programa, descrito em Pascal podemos ter textos que não são considerados para a geração do código de máquina e servem para ajudar a compreensão do programa ou apenas como comentários.

Comentários em Pascal podem ser inseridos colocando-os entre "(*" e "*)" ou colocando-os entre "{" e "}".

Assim, o programa *BomDia2* (figura 9) abaixo é equivalente aos dois programas acima.

```
  { Programa: BomDia.Pas                                   }
  { Aluno: Fulano de Tal                     RA: 999999    }
  { Curso: MC102                                           }
  { Data: 01/01/01                                         }
  { Descrição: Programa para Imprimir "Bom Dia!"           }
program BomDia2;       { Programa para Imprimir "Bom Dia!" }
begin
    write('Bom ');         { Eu sou um comentário }
    writeln('Dia!');       (* Eu também sou um comentário *)
end.
```

Figura 9:

A seguir apresentamos o fluxograma do programa BomDia2.

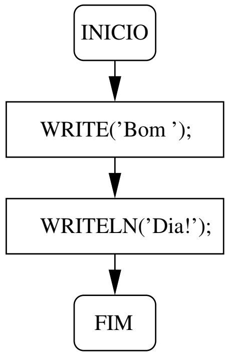


Figura 10: Fluxograma do programa hello3.

2.2 Identificadores e Constantes

No programa *BomDia* imprimimos textos usando os comandos **write** e **writeln**. Além disso, estes comandos também podem ser usados para imprimir expressões com tipos básicos do pascal. Considere o seguinte programa na figura 11:

```
program idade1;
begin
    writeln('Paulo tem ',5,' anos');
    writeln('Pedro tem ',8,' anos');
    writeln('A soma das idades de Paulo e Pedro é:',5+8);
    writeln('A media das idades de Paulo e Pedro é:',(5+8)/2);
end.
```

Figura 11:

O programa acima deve imprimir as linhas:

Paulo tem 5 anos
Pedro tem 8 anos
A soma das idades de Paulo e Pedro é: 13
A media das idades de Paulo e Pedro é: 6.5

Observe que o comando **writeln** pode manipular tanto números inteiros como números reais. Note também que se a idade de Pedro passar de 8 para 9 anos, então todos os lugares onde aparece o número 8, correspondente à idade de Pedro, devemos alterá-lo para 9.

Para facilitar este tipo de mudança podemos fazer uso de um identificador associado à idade de Pedro, igual a 8 anos. Assim, sempre que precisamos referenciar sua idade referenciamos este identificador em vez do valor 8. Isto facilita a atualização da idade de Pedro, que se formos mudar de 8 anos para 9 anos, atualizamos apenas na definição do valor do identificador associado à idade de Pedro. Desta maneira, não precisaremos nos preocupar em procurar todas as ocorrências do número 8, além de verificar se tal 8 é correspondente à idade de Pedro.

Identificadores são nomes simbólicos para os objetos referenciados nos programas em Pascal. Identificadores podem ser formados por uma letra ou caracter sublinhado seguido por qualquer combinação de letras, dígitos ou sublinhados. Na tabela a seguir, damos alguns exemplos de identificadores válidos e inválidos.

Identificadores válidos	Identificadores inválidos
Media	2X
Meu_Primeiro_Programa	A:B:C
Idade_Paulo	1*5
RA_999999	X(9)
Nota_MC102	Nota(102)

Na linguagem Pascal, não há diferenças com as letras maiúsculas e minúsculas usadas no identificador. Assim, as seguintes seqüências de caracteres representam o mesmo identificador:

Nota_MC102	NOTA_MC102	nota_mc102	noTA_mC102

Obs.: Em algumas linguagens, como a *linguagem C*, há distinção entre maiúsculas e minúsculas usadas em identificadores.

Constantes são objetos cujos valores não mudam durante a execução do programa. A definição das constantes deve ser feita na área de declarações, usando a palavra reservada **const** seguida das definições de cada constante. Cada constante é definida como:

IdentificadorDeConstante = constante;

No programa da figura 12, apresentamos o programa da figura 11 declarando as idades de Paulo e Pedro por constantes.

```
program idade2;
const Idade_Paulo = 5;
      Idade_Pedro = 9;
begin
   writeln('Paulo tem ',Idade_Paulo,' anos');
   write('Pedro tem ',Idade_Pedro,' anos');
   writeln('A soma das idades de Paulo e Pedro é:',Idade_Paulo+Idade_Pedro);
   writeln('A media das idades de Paulo e Pedro é:',(Idade_Paulo+Idade_Pedro)/2);
end.
```

Figura 12:

Outros exemplos de declaração de constantes:

```
program constantes;
const
      Min = 100;
      Max = 999;
      Versao = '5.2.3';
      LetraPrincipal = 'X';
      Aniversario = '01/01/2000';
⋮
begin
      writeln(Min,'...',Max);  {Imprime: 100...999 }
      writeln('Versão do Programa é ',Versao);  {Imprime: Versão do Programa é 5.2.3 }
⋮
end.
```

Figura 13:

2.3 Variáveis e Tipos Básicos

O uso de constantes permite que possamos trabalhar com valores previamente fixados e seu uso é feito através de um identificador que está associado àquela constante. Agora se quisermos que um determinado identificador esteja associado à diferentes valores durante a execução do programa, devemos usar o conceito de *variável*.

Variáveis são objetos que podem ter diferentes valores durante a execução do programa. Cada variável corresponde a uma posição de memória. Embora uma variável possa assumir diferentes valores, ela só pode armazenar apenas um valor a cada instante. Cada variável é identificada por um identificador e contém valores de apenas um tipo.

Os *tipos básicos* em Pascal são: **char, integer, boolean, real, string**. A tabela a seguir apresenta exemplos de cada um destes tipos.

Objeto	*tipo de dado*
1	integer
-2488	integer
1.0	real
-13.6	real
0.653	real
-1.23456789E+12	real expresso em notação exponencial
’Algoritmos e Programação de Computadores’	string
’123456’	string
true	boolean
false	boolean
’A’	char
’B’	char
’6’	char
’$’	char

Vamos ver mais sobre cada um destes tipos:

integer: Representa um subconjunto dos números inteiros. O tipo integer no padrão Turbo Pascal representa os inteiros de -32768 a 32767. Já no Padrão Gnu Pascal (GPC), o tipo integer representa inteiros de -2147483648 a 2147483647.

real: Em geral, um número real é representado por duas partes: a *mantissa* e o *expoente*. A precisão dos números reais é determinada pela quantidade de bits usada em cada uma destas partes. Exemplo: o número $10,45$ pode ser representado por $1.045E+01$. O número $0,00056993$ pode ser representado por $5.6993E-04$. Este tipo não permite representar todos os números reais, mesmo que este número seja pequeno. Um exemplo simples disso é o número $\frac{1}{3} = 0{,}33333\ldots$, que certamente deve ter sua precisão computada corretamente até um certo número de casas decimais.

char: Representa um caracter. Ex.: ’A’,’B’,’$’,’ ’ (espaço em branco),...

string: É uma seqüência de caracteres. Uma string pode ser dada por uma seqüência de caracteres entre apóstrofos. Ex: ’Joao Paulo’. A maioria dos compiladores tem a restrição que uma string pode ter até 255 caracteres. Atualmente há compiladores que admitem strings com número bem maior de caracteres. A quantidade de caracteres (bytes) usados pela string é definido com um número entre colchetes logo após a palavra string. Ex.: **string[50]**;
O tipo acima corresponde a uma cadeia de caracteres com capacidade máxima de 50 caracteres. Obs.: uma cadeia de caracteres a ser armazenada neste tipo de string não necessariamente precisa ter 50 caracteres. Para saber mais sobre este tipo, veja a seção 8.

boolean: Uma variável do tipo boolean pode ter dois valores. True (verdadeiro) ou False (falso).

Além destes tipos, vamos considerar como tipo básico o tipo **byte**, que usa apenas um byte de memória e pode armazenar um número de 0 a 255.

A declaração de variáveis é feita usando se a seguinte sintaxe:

```
var
    ListaDeIdentificadoresDoTipo1   :  Tipo1;
    ListaDeIdentificadoresDoTipo2   :  Tipo2;
                                                  ⋮
    ListaDeIdentificadoresDoTipoN   :  TipoN;
```

A palavra **var** é uma palavra reservada da linguagem Pascal que indica que vai começar uma declaração de *variáveis*.

Exemplo 2.1 *No quadro seguinte apresentamos a declaração de algumas variáveis.*

var x, y, z: **real**;
Nome, Sobrenome: **string[50]**;
$i, n, idade$: **integer**;
Sexo: **char**;

2.4 Comando de Atribuição

O comando de atribuição **:=** atribui um valor que está a direita de :=, que pode ser uma expressão, para uma variável (única) que está na parte esquerda do comando, representada pelo seu identificador.

Sempre que houver um comando de atribuição devemos observar os seguintes itens:

- Primeiro é avaliada a parte direita, obtendo-se um valor único. A parte da direita do comando de atribuição pode ser uma expressão bem complicada.
- O valor obtido da expressão da parte direita é atribuído à variável (única) que está na parte esquerda.
- O tipo do valor avaliado da expressão na parte direita deve ser compatível com o tipo da variável que está na parte esquerda. Obs.: um número inteiro também pode ser visto como um número real.
- Se um valor é armazenado em uma variável, o valor anterior que estava nesta variável é perdido.

Exemplo 2.2 *Considere a declaração das variáveis* x, y, z *e* t *dadas a seguir:*

var *nome:* **string[50]**;
x: **integer**;
y: **real**;
z,t: **char**;

Na tabela seguinte apresentamos alguns exemplos de atribuições válidas e inválidas para estas variáveis.

Atribuições válidas	*Atribuições inválidas*
$x := 10$	$x := 3.14$
$y := x + 1.5$	$x := y$
$z :=' a'$	$z := '\text{Computador}'$
$z := t$	$t := t + z$
nome:='Carlos'	*nome:=100;*

Para esclarecer melhor estes conceitos, suponha que você tenha uma variável, chamada idade, definida como sendo inteira:

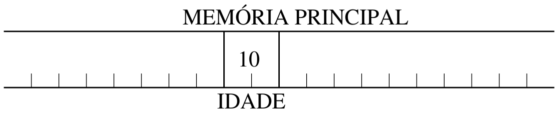
var idade: **integer**;

No momento da execução, o programa deve definir uma certa porção da memória, em bytes, para esta variável. A maioria das linguagens não inicializa as variáveis logo após a definição de sua memória em bytes. Assim, vamos considerar que o valor inicial em uma variável é sempre um valor desconhecido e portanto iremos considerá-lo como um *lixo*.

Após o comando

Idade:=10;

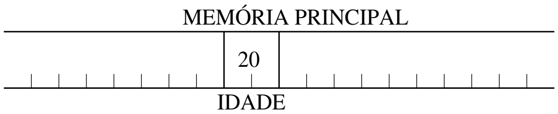
a configuração da memória deve ficar da seguinte forma



Se depois deste comando for dado o comando:

Idade:= 20;

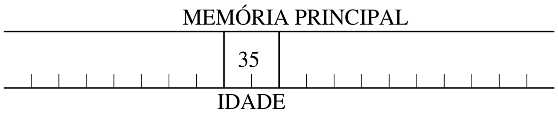
Então a programa toma a valor 20 (parte direita) e o armazena na variável idade (parte esquerda), e a variável idade perde o valor anterior. Neste momento a memória fica com a seguinte configuração:



Se depois deste comando for dado o seguinte comando:

Idade:=Idade + 15;

O programa deve avaliar *primeiro o valor da parte direita*, depois atribui para a variável (que está na parte esquerda). Assim, analisando a parte direita, temos 20+15=35; então o valor 35 é atribuído para a variável Idade (parte esquerda). Assim, a configuração da memória deve ficar da seguinte forma



Exemplo 2.3 *Considere o seguinte programa*

```
program Soma_3_20_100;
var Soma: integer;
begin
    Soma:=3;
    Soma:=Soma+20;
    Soma:=Soma+100;
    writeln('O valor contido em Soma é: ',Soma);
end.
```

1. *No primeiro comando de atribuição, a variável Soma deve receber o valor 3.*
2. *Para executar o segundo comando de atribuição, primeiro é avaliado o lado direito do comando: (Soma+20). Note que neste exato momento, a variável Soma contém o valor 3. Este é somado a 20, resultando em um valor único do lado direito igual a 23. Em seguida, este valor é armazenado na variável que está no lado esquerdo. Assim o valor 23 é armazenado na variável Soma.*
3. *No terceiro comando de atribuição, primeiro é avaliado o valor da expressão do lado direito do comando,* $(23 + 100)$ *resultando em* 123 *e este é atribuído à variável Soma.*
4. *No quarto comando, de impressão, deve ser impresso o seguinte texto:*
   O valor contido em Soma é: 123

Exercício 2.1 *Considere o seguinte programa:*

```
program Produtos;
var Produto : integer;
begin
    Produto:=2;
    Produto:=Produto*Produto;
    Produto:=Produto*Produto;
    Produto:=Produto*Produto;
    writeln('O valor contido em Produto é: ',Produto);
end.
```

O que o programa imprime ?

Obs.: Muitas das linguagens de programação não inicializam automaticamente as variáveis. Assim, é importante inicializar as variáveis em algum momento, antes de referenciar seus valores em expressões.

2.5 Operadores

Operadores Aritméticos

A seguir, apresentamos alguns operadores aritméticos presentes na linguagem Pascal.

- +,-,*,/ (adição, subtração, multiplicação e divisão de números, resp.).
- **mod** (Operador de inteiros: resto de divisão inteira).
- **div** (Operador de inteiros: parte inteira da divisão).

Operandos dos tipos **integer** e **real** podem ser combinados. O tipo do valor resultante de uma operação aritmética dependerá da operação e dos tipos de seus operadores. A tabela a seguir apresenta as operações válidas para tipos de operandos e seus respectivos resultados.

Operandos	*Operador*	*Tipo de resultado*
integer e **integer**	$+, -, *, \mathbf{div}, \mathbf{mod}$	**integer**
integer e **integer**	$/$	**real**
real e **integer**	$+, -, *, /$	**real**
real e **real**	$+, -, *, /$	**real**

Exemplo 2.4 *Exemplos de operações aritméticas:*

```
program OpAritmetico;
var i:integer;
    x:real;
begin
    i := 4 mod 3;  {resultado da operação é inteiro e igual a 1}
    x := 4/2;  {resultado da operação é real e igual a 2.0}
    x := 2 * 3 + 5 * 3 - (123456 mod 123);
end.
```

Operadores Lógicos

A linguagem Pascal oferece os seguintes operadores lógicos:

- **not** (inverte o valor booleano)
- **and** (é verdadeiro se ambos os operandos são verdadeiros)
- **or** (é verdadeiro se um dos operandos for verdadeiro)

Exemplo 2.5 *Exemplos de operações lógicas:*

```
program OpLogico;
var a, b, c: boolean;
begin
    a := false;   b := true;
    c := ( not a and b); {c ←True (verdadeiro)}
    writeln(c);
end.
```

Exercício 2.2 *Algumas implementações da Linguagem Pascal apresentam o operador lógico* **xor**, *sendo que este operador resulta verdadeiro se exatamente um dos operandos for verdadeiro. Implemente este operador usando os operadores* **and**, **or** *e* **not**.

Operadores Relacionais

Os operadores relacionais podem ser usados para os tipos **real, integer, string, char** e **byte**. Naturalmente os dois operandos devem ser de tipos compatíveis.

O resultado de uma operação relacional é sempre lógico (**boolean**), retornando ou **true** ou **false**.

No caso do operador = (igual), também é possível se comparar valores booleanos. Na comparação de strings, os caracteres são comparados dois a dois caracteres nas mesmas posições, até que um caracter seja menor que outro ou até que uma string termine antes que outra. Assim, 'Ave Maria' é maior que 'Ave Cesar'. Pois o primeiro caracter a diferir nas duas strings é o quinto caracter. Na codificação ASCII, 'M' é maior que 'C'. Portanto o resultado da relação

('Ave Maria' $>$ 'Ave Cesar')

é true (verdadeiro). Como a codificação de 'A' em ASCII é menor que 'a', então o resultado da relação

('AAA' $>=$ 'aaa')

é false (falso).

Exemplo 2.6 *Exemplos de operações relacionais:*

```
program OpRelacionais;
var a, b, c: boolean;
    x, y: real;
begin
    x := 10.0;   y := 30.0;   a := false;
    b := not a and ((x >= y) or ('paulo' <= 'pedro'));
    c := ('paulo' <= 'Paulo'); {resultado é falso, já que 'p'>'P'}
    writeln(b);
    writeln(c);
end.
```

Operadores em Strings

O único operador para seqüência de caracteres é o operador de concatenação de strings, **+**, que concatena duas strings.

Exemplo 2.7 *Exemplo da operação de concatenação de strings:*

```
program OpConcatenaString;
var a: string[50];
begin
    a :='Algoritmos';
    a := a+'e';
    a := a+'Programação';
    a := a+'de';
    a := a+'Computadores';
    writeln(a);   {Deve imprimir: AlgoritmoseProgramaçãodeComputadores}
end.
```

Precedência

Ao se avaliar uma expressão, os operadores seguem uma ordem de precedência. Estas regras de precedência são familiares às usadas em operadores algébricos. Se dois operadores possuem o mesmo nível de precedência, então a expressão é avaliada da esquerda para a direita. Parte de expressões que estão contidas entre parênteses são avaliadas antes da expressão que a engloba. A seguinte tabela apresenta a ordem de prioridade dos operadores (primeiro os de maior precedência) na linguagem Pascal.

Expressões dentro de parênteses
Operador unário (positivo ou negativo)
Operador **not**
Operadores multiplicativos: $*, /, \mathbf{div}, \mathbf{mod}, \mathbf{and}$
Operadores aditivos: $+, -, \mathbf{or}, \mathbf{xor}$
Operadores relacionais: $=, <>, <, >, <=, >=, \mathbf{in}$

Sempre que quisermos quebrar a ordem natural das operações, podemos usar parênteses para especificar a nova ordem. Assim, uma atribuição como $Exp := (1 + (3 + 5) * (9 - 20/(3 + 7)))$, pode ter a seguinte ordem de operações

$$\begin{aligned}
Exp &:= (1 + (3 + 5) * (9 - 20/(3 + 7))) \\
Exp &:= (1 + 8 * (9 - 20/(3 + 7))) \\
Exp &:= (1 + 8 * (9 - 20/10)) \\
Exp &:= (1 + 8 * (9 - 2)) \\
Exp &:= (1 + 8 * 7) \\
Exp &:= (1 + 56) \\
Exp &:= 57.
\end{aligned}$$

2.6 Algumas Funções Pré-Definidas

A maioria das linguagens de programação já nos oferece um subconjunto de funções básicas que podemos usar para construir expressões mais complexas. A tabela a seguir apresenta algumas destas funções.

Função	*Tipo do parâmetro*	*Tipo do Resultado*	*Resultado*
ArcTan	**real**	**real**	arco tangente
Cos	**real**	**real**	cosseno
Sin	**real**	**real**	Seno
Abs	**integer**	**integer**	valor absoluto
Abs	**real**	**real**	valor absoluto
Exp	**real**	**real**	exponencial
Frac	**real**	**real**	parte fracionária
Int	**real**	**real**	parte inteira
Ln	**real**	**real**	Logaritmo Natural
Random	N: **integer**	**integer**	Número pseudo aleatório em $[0, \ldots, N-1]$
Random	sem parâmetro	**real**	Número pseudo aleatório em $(0, \ldots, 1)$
Round	**real**	**integer**	Arredondamento
Sqr	**integer**	**integer**	quadrado
Sqr	**real**	**real**	quadrado
Sqrt	**real**	**real**	raiz quadrado
Trunc	**real**	**integer**	Truncamento
Chr	**integer**	**char**	ordinal para caracter
Ord	*tipo escalar, exceto reais*	**integer**	número ordinal do tipo escalar
Length	**string**	**integer**	Quantidade de caracteres da string

Exemplo 2.8 *Uma maneira muito comum de se fazer a potência de um número x, $x > 0$, elevado a outro número y, é através das funções* **Exp** *e* **Ln***, usando a seguinte fórmula:* $\boxed{x^y = \mathbf{Exp}(y * \mathbf{Ln}(x))}$*. O programa a seguir lê dois números, x e y, e imprime x^y.*

```
program Expoente;
var x,y,z : real;
begin
  write('Entre com o valor de x: ');
  readln(x);
  write('Entre com o valor de y: ');
  readln(y);
  z:= Exp(y*ln(x));
  writeln('O valor de ',x,' elevado a ',y,' é igual a ',z);
end.
```

Além destas funções, existem alguns comandos úteis para atualizar o valor de algumas variáveis. Algumas destas são:

Comando	*Tipo da variável de parâmetro*	*Resultado*
Inc	inteiro	Incrementa o valor da variável de 1.
Dec	inteiro	Decrementa o valor da variável de 1.

2.7 Comandos de Escrita

Como vimos, podemos escrever os dados na tela através do comando **writeln**. Este comando tem como parâmetros uma lista de objetos —de tipos básicos—. Os parâmetros devem ser separados por vírgulas. Cada parâmetro é impresso logo após a impressão do parâmetro anterior. A linguagem Pascal tem dois comandos básicos para escrita: o comando **write** e o comando **writeln** (de *write+line*). Ambos imprimem os valores de seus respectivos argumentos, com a diferença que o comando **write** mantém a posição da próxima impressão/leitura logo após a impressão de seu último parâmetro e o comando **writeln** atualiza a posição da próxima impressão/leitura para o início da próxima linha. O comando **writeln** sem argumentos apenas atualiza a posição da próxima impressão/leitura para o início da próxima linha.

Exemplo 2.9 *Considere o programa a seguir:*

```
program ComandosDeEscrita;
var
    a: boolean;
    b: integer;
    c: string[50];
begin
    a := true;
    b := 2 + 3 * 4 - 5;
    c :='Exemplo';
    writeln('Valor de a = ',a,'. Vai para a próxima linha');
    writeln('Valor de b = ',b,'. Vai para a próxima linha');
    writeln('Valor de c = ',c,'. Vai para a próxima linha');
    write('Valor de a = ',a,'. ');
    write('Valor de b = ',b,'. ');
    write('Valor de c = ',c,'.');
    writeln;
    writeln('Fim das impressões');
end.
```

O programa acima imprime as seguintes linhas:

```
Valor de a = True. Vai para a próxima linha
Valor de b = 9. Vai para a próxima linha
Valor de c = Exemplo. Vai para a próxima linha
Valor de a = true. Valor de b = 9. Valor de c = Exemplo.
Fim das impressões.
```

Muitas vezes queremos formatar melhor a forma de impressão dos dados no comando *write* e *writeln*. Uma maneira simples de definir a quantidade de caracteres, que uma expressão será impressa, é colocar a expressão a ser impressa seguida de : (dois pontos) e a quantidade de caracteres desejado. Isto fará com que a expressão seja impressa com a quantidade de caracteres especificada. Caso o dado a ser impresso é um número real, podemos complementar esta formatação adicionando : (dois pontos) e a quantidade de casas decimais depois do ponto decimal. No exemplo a seguir apresentamos um programa com algumas formatações de impressões e sua respectiva tela de execução.

Exemplo 2.10 *Considere o programa a seguir:*

```
program ComandosDeEscrita;
var
    a: boolean;
    b: integer;
    c: string[50];
    d: real;
begin
    a := true;
    b := 4321;
    c :='Exemplo';
    d := 1234.5678;
    writeln('Valor de a = [',a:1,']');
    writeln('Valor de b = [',b:7,']');
    writeln('Valor de c = [',c:4,']');
    writeln('Valor de d = [',d:9:2,']');
end.
```

O programa acima imprime as seguintes linhas:

Valor de a = [T]
Valor de b = [4321]
Valor de c = [Exem]
Valor de d = [1234.57]

Obs.: Na formatação de valores reais, pode haver perda ou arredondamentos em algumas casas decimais.

2.8 Comandos de Leitura

Há dois comandos básicos de leitura: o comando **read** e o comando **readln**. Ambos os comandos tem como parâmetros variáveis, de tipos básicos diferentes de **boolean**, e permite ler do teclado os novos valores a serem atribuídos às variáveis. O comando **read**, após sua execução, mantém a posição da próxima impressão/leitura logo após a leitura das variáveis. O comando **readln** (de *read line*) atualiza a posição da próxima impressão/leitura para o início da próxima linha.

Exemplo 2.11 *Considere o programa a seguir:*

```
program ComandosDeLeitura;
var
     idade: integer;
     nome: string[50];
begin
     write('Entre com sua idade: ');
     readln(idade);
     write('Entre com seu nome: ');
     readln(nome);
     writeln('O Sr. ',nome,' tem ',idade,' anos.')
end.
```

Caso o usuário entre com os dados 29 *e* Fulano de Tal*, o programa deve ficar com a seguinte configuração na tela:*

Entre com sua idade: 29
Entre com seu nome: Fulano de Tal
O Sr. Fulano de Tal tem 29 anos.

Exercício 2.3 *Faça um programa que lê 7 números e imprima a média destes números. Obs.: Use no máximo duas variáveis numéricas.*

Exercício 2.4 *Sabendo que o valor numérico de uma letra na codificação ASCII é dada pela função* **ord***, faça um programa que leia uma letra e escreva a codificação em binário desta letra.*
Exemplo: Suponha que a letra a ser lida é a letra 'a'. A função **ord***('a') retorna o valor* 97 *e o programa deve imprimir:* **01100001**. *Note que* $0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 97$.

Exercício 2.5 *Faça uma programa que lê dois pontos* $P_1 = (x_1, y_1)$ *e* $P_2 = (x_2, y_2)$*; e devolve a distância entre estes dois pontos, cujo valor é impresso no programa principal. Obs.:* $dist = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.

Exercício 2.6 *Faça um programa que lê uma temperatura em graus Fahrenheit e retorne a temperatura em graus Centígrados. Obs.:* $(C = 5/9 \cdot (F - 32))$.

Exercício 2.7 *Faça um programa que lê uma temperatura em graus Centígrados e retorne a temperatura em graus Fahrenheit.*

2.9 Tipos Escalares

Os tipos escalares **integer**, **byte** e **char** são tipos que representam conjuntos ordenados com valores distintos. A linguagem Pascal também permite que possamos definir nossos próprios tipos de dados escalares ordenados. Isto pode ser feito definindo o conjunto ordenado especificando cada elemento do conjunto ou aproveitando os elementos de conjunto de dados já definidos anteriormente. A seguir apresentamos estas duas maneiras de se definir tipos escalares ordenados.

Com este tipo de dados, por serem ordenados, podemos fazer comparações do tipo: igualdade (=), menor (<), menor ou igual (<=), maior (>), maior ou igual (>=) e diferença (<>).

Especificando todos os elementos

Para definir um tipo escalar ordenado especificando todos os elementos, escrevemos seus elementos (como identificadores) entre parênteses e separados por vírgula.
Obs.: estes tipos não podem ser impressos diretamente com uso do comando write ou writeln.

(*Lista_de_identificadores*)

Como já vimos, a função **ord** sobre um caracter devolve o código ASCII do mesmo. A função **ord** também pode ser usada para encontrar a posição de outros elemento definidos por conjuntos escalares ordenados especificados elemento a elemento. A função **ord** retorna 0 caso seja o primeiro elemento, 1 se for o segundo elemento, 2 se for o terceiro elemento, assim por diante.

Exemplo 2.12 *A seguir apresentamos alguns exemplos de declarações e atribuições usando os conjuntos ordenados especificados elemento a elemento.*

```
type
    TipoSexo = (Masculino,Feminino);
    TipoCor = (Vermelho,Verde,Azul);
    TipoMes = (Janeiro,Fevereiro,Marco,Abril,Maio,Junho,Julho,
               Agosto,Setembro,Outubro,Novembro,Dezembro);
    TipoEstadoSul = (RS,SC,PR);
var
    Sexo: TipoSexo;
    Estado: TipoEstadoSul;
    Cor: TipoCor;
    Categoria: (Aluno,Funcionario,Professor);
    Mes: TipoMes;
begin
    Sexo := Masculino;
    Estado := PR;
    Categoria := Funcionario;
    Mes := Abril;
    writeln(Ord(Mes)); {Deve imprimir 3}
end.
```

Muitos compiladores fazem uma otimização de maneira que se o tipo de conjunto ordenado tiver até 256 elementos, então este tipo usa apenas um byte de memória.

Especificando uma faixa de elementos

Um tipo faixa é um tipo de dado escalar que define uma faixa de elementos. Esta faixa é especificada pelo primeiro elemento e o último elemento. Os elementos de um tipo de dado faixa devem fazer parte de um conjunto ordenado escalar. Para isso é possível usar tanto os tipos escalares já definidos (como os tipos **integer** e **char**) como também os novos tipos escalares ordenados como definidos elemento a elemento. Um tipo faixa é definido especificando o primeiro e o último elemento na seguinte sintaxe:

Primeiro_Elemento..Ultimo_Elemento

Isto significa que uma variável deste tipo poderá receber qualquer valor que esteja no intervalo *Primeiro_Elemento..Ultimo_Eleme*
Se uma variável é definida como sendo de um tipo faixa onde os extremos foram tomados a partir de um conjunto ordenado especificado elemento a elemento, então os elementos que são passíveis de serem atribuídos a esta variável ficam restritos ao intervalo definido no conjunto ordenado também.

Um tipo faixa também pode ser usado para especificar os índices de vetores, mesmo que não sejam números inteiros.

Obs.: Um tipo escalar só poderá ser impresso pelo comando write ou writeln se este for do tipo char ou integer.

Exemplo 2.13 *A seguir apresentamos alguns exemplos de declarações e atribuições usando uma faixa de elementos.*

```
type
    TipoMaiuscula = 'A'..'Z';
    TipoMinuscula = 'a'..'z';
    TipoDigitoCaracter = '0'..'9';
    TipoDigito = TipoDigitoCaracter;
    TipoDigitoNumero = 0..9;
    TipoMes = (Janeiro,Fevereiro,Marco,Abril,Maio,Junho,Julho,
               Agosto,Setembro,Outubro,Novembro,Dezembro);
    TipoMesPrimeiroSemestre = Janeiro..Junho; {Supondo a declaração de TipoMes do exemplo anterior}
    TipoString = string[100]; {Tipo auxiliar}
var
    DigC: TipoDigito;
    DigN: TipoDigitoNumero;
    IdadeAdolecencia: 12..18 ;
    Letra: TipoMaiuscula ;
    MesEstagio: TipoMesPrimeiroSemestre ;
begin
    DigC := '5';
    DigN := 5;
    Letra := 'X';
    MesEstagio := Abril;
end.
```

3 Estrutura Condicional

A estrutura condicional permite a execução de instruções quando uma condição representada por uma expressão lógica é satisfeita.

3.1 Estrutura Condicional Simples

if *(condição)* **then** Comando;

A seguir, exemplificamos o uso da estrutura condicional simples.

```
program maximo1;
var a,b: integer;
begin
    write('Entre com o primeiro número: ');
    readln(a);
    write('Entre com o segundo número: ');
    readln(b);
    if (a > b)
      then
          writeln('O maior valor é ',a);
    if (a <= b)
      then
          writeln('O maior valor é ',b)
end.
```

3.2 Estrutura Condicional Composta

Note que no exemplo do algoritmo *maximo1*, exatamente um dos dois comandos **if**'s é executado, i.e., ou $(a > b)$ ou $(a \leq b)$. Podemos considerar ambos os casos através de uma única condição:

if *(condição)*
    then Comando1_ou_Bloco_de_Comandos1
    else Comando2_ou_Bloco_de_Comandos2;

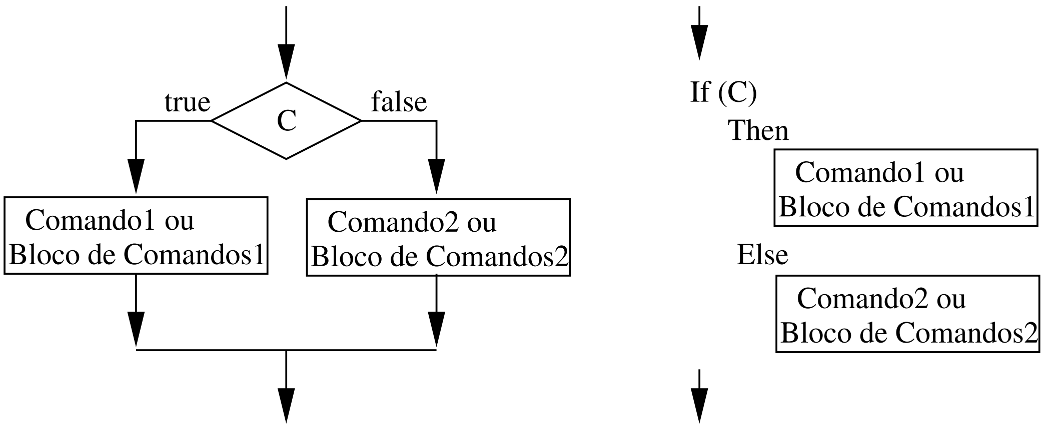


Figura 14: Fluxograma do comando If-Then-Else

Note que depois do *Comando1_ou_Bloco_de_Comandos1* não há ; (ponto e vírgula). Caso tivesse um ponto e vírgula, este estaria separando o comando **if** de outro comando e portanto a palavra **else** não seria continuação deste comando **if**. Na figura 14 apresentamos o fluxograma do comando **if**, e no programa seguinte exemplificamos o uso da estrutura condicional composta.

```
program maximo2;
var a,b: integer;
begin
    write('Entre com o primeiro número: ');    readln(a);
    write('Entre com o segundo número: ');    readln(b);
    if (a > b)
      then
          writeln('O maior é ',a)
      else
         writeln('O maior é ',b)
end.
```

3.3 Bloco de Comandos

Um bloco de comandos é um conjunto de instruções que vai ser considerado como sendo um comando único. Desta forma, é possível compor estruturas, como no caso do comando if, envolvendo mais comandos. Um bloco de comandos é definido pelas palavras **begin** e **end**.

```
begin
    Comando1;
    Comando2;
    Comando3;
      ⋮
end;
```

Aqui parece ser o momento para apresentar mais uma prática de programação estruturada. Sempre que um comando atuar sobre um bloco de comandos, dispomos este bloco de comandos deslocados mais a direita. Os comandos dentro de um bloco, entre **begin** e **end** estarão alinhados e também estarão deslocados mais a direita. Isto facilita muito a visualização da atuação de cada comando.

Exemplo 3.1 *No programa a seguir, exemplificamos o uso dos blocos de comandos.*

```
program maximo3;
var a,b: integer;
begin
    write('Entre com o primeiro número: ');    readln(a);
    write('Entre com o segundo número: ');    readln(b);
    if (a > b)
      then
          begin
              writeln('O maior está na variável a');
              writeln('O maior é ',a)
          end
      else
         begin
              writeln('O maior esta na variável b');
              writeln('O maior é ',b)
         end
end.
```

Exemplo 3.2 *No exemplo seguinte apresentamos um programa que lê os coeficientes de uma equação de segundo grau, a, b e c (equação $ax^2 + bx + c = 0$), e imprime as raizes desta equação (se existir).*

```
program equacaosegundograu;
var a, b, c: real;
    var x1, x2, delta: real;
    begin
        writeln('Programa para resolver uma equação de segundo grau.');
        writeln('Equação na forma: a*x*x + b*x+c = 0');
        write('Entre com o valor de a (diferente de zero): '); readln(a);
        write('Entre com o valor de b: '); readln(b);
        write('Entre com o valor de c: '); readln(c);
        delta := b * b - 4 * a * c;
        if (delta >= 0)
          then
            begin
                x1 := (-b + sqrt(delta))/(2 * a);
                x2 := (-b - sqrt(delta))/(2 * a);
                writeln('O valor de x1 = ',x1,' e o valor de x2 = ',x2);
            end
          else
            writeln('não é possível calcular raizes reais para esta equação');
    end.
```

3.4 Comando Case

O comando **case** é um comando que permite selecionar um conjunto de operações conforme uma expressão com resultado escalar. Neste comando o valor escalar é comparado com vários outros valores que dividem as opções de execução em casos (vamos chamar estes de valores de caso). No máximo um caso pode ser verdadeiro. Os valores de caso podem ser tanto valores escalares como faixas de valores de escalares. Além disso, todos os valores de caso devem ser disjuntos (sem interseções). Se o valor escalar é igual ao valor (ou está dentro de uma faixa) de um caso então os comandos associados a este caso são executados.

As sintaxes do Extended Pascal e Borland Pascal diferem levemente. Assim, daremos a sintaxe do Borland Pascal e posteriormente explicitaremos a diferença com a sintaxe do Extended Pascal.

Há duas formas para a sintaxe do comando **case**:

A seguir apresentamos a sintaxe do caso onde não definimos comandos para o caso de não haver correspondência com os valores escalares de cada caso.

```
case (Expressão_Escalar) of
      Lista_Constantes_Escalares1: Comando_ou_Bloco_de_Comandos1;
      Lista_Constantes_Escalares2: Comando_ou_Bloco_de_Comandos2;
                                  ⋮
      Lista_Constantes_EscalaresK: Comando_ou_Bloco_de_ComandosK;
end
```

A sintaxe para a situação onde definimos comandos para o caso onde não há correspondência com os valores escalares de cada caso é apresentada a seguir:

```
case (Expressão_Escalar) of
        Lista_Constantes_Escalares1: Comando_ou_Bloco_de_Comandos1;
        Lista_Constantes_Escalares2: Comando_ou_Bloco_de_Comandos2;
                                        ⋮
        Lista_Constantes_EscalaresK: Comando_ou_Bloco_de_ComandosK;
                        else
                                        ComandoE1;
                                        ComandoE2;
                                        ⋮
                                        ComandoEE;
end
```

Cada Lista_Constantes_Escalares pode ser uma seqüência de escalares ou faixas de escalares separados por vírgula.

A diferença com a sintaxe do padrão Extended Pascal é que em vez da palavra **else** na sintaxe anterior, é usada a palavra **otherwise**. No decorrer deste texto, usaremos a sintaxe do padrão Borland Pascal.

Exemplo 3.3 *O seguinte programa mostra um exemplo de uso do comando* **case***.*

```
program exemplo;
var c: char;
begin
        write('Entre com um caracter: ');
        readln(c);
        case c of
                'A'..'Z','a'..'z':
                                writeln('Caracter lido é letra.');
                '0'..'9': begin
                                writeln('Caracter lido é dígito.');
                                writeln('i.e., um caracter em [0,9].');
                          end;
                '+','-','*','/':
                                writeln('Caracter é um operador matemático.');
                '$': writeln('Caracter é o símbolo $.');
                else
                                writeln('O caracter lido não é letra, nem digito.');
                                writeln('nem operador matemático e nem é o símbolo $.');
        end;
end.
```

Exemplo 3.4 *O seguinte programa mostra um exemplo de menu implementado com o comando* **case**.

```
Program ExemploMenu;
var Opcao : char;
begin
  repeat
    { Escolha de uma opção do Menu }
    writeln('Entre com uma opção do menu abaixo: ');
    writeln('[1] - Inserir dados de aluno novo no cadastro.');
    writeln('[2] - Remover aluno do cadastro.');
    writeln('[3] - Alterar os dados de um aluno.');
    writeln('[4] - Sair do sistema de cadastro.');
    write('Opcao: '); readln(Opcao);   writeln;
    case Opcao of
     '1' : begin
           writeln('Inserir Funcionário.');
           writeln('Opção não implementada.');
         end;
     '2' : begin
           writeln('Remover Funcionário.');
           writeln('Opção não implementada.');
         end;
     '3' : begin
           writeln('Alterar Funcionário.');
           writeln('Opção não implementada.');
         end;
     '4' : writeln('Fim da execução do sistema.');
    else
      writeln('Opção inválida.');
    end; { case }
    writeln;
  until (Opcao = '4');
end.
```

3.5 Exercícios

1. Escreva um programa que determina a data cronologicamente maior de duas datas fornecidas pelo usuário. Cada data deve ser fornecida por três valores inteiros onde o primeiro representa um dia, o segundo um mês e o terceiro um ano.

2. Faça um programa que lê uma medida em *metros* e escreve esta medida em *polegadas, pés, jardas* e *milhas*. Obs.:

$$\begin{aligned} 1 \text{ polegada} &= 25.3995 \text{ milímetros} \\ 1 \text{ pé} &= 12 \text{ polegadas} \\ 1 \text{ jarda} &= 3 \text{ pés} \\ 1 \text{ milha} &= 1760 \text{ jardas} \end{aligned}$$

3. Sabendo que o valor numérico de uma letra na codificação ASCII é dada pela função **ord**,

   (a) faça um programa que leia uma letra e escreva a codificação em binário desta letra.
   Exemplo: Suponha que a letra a ser lida é a letra 'a'. A função **ord**('a') retorna o valor 97 e o programa deve imprimir: **01100001**.
   Note que $0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 97$.

(b) faça um programa que leia uma letra e escreva a codificação em hexadecimal desta letra.
Exemplo: Suponha que a letra a ser lida é a letra 'm'. A função **ord**('m') retorna o valor 109 e o programa deve imprimir: **6D**. Note que $6 \cdot 16^1 + D \cdot 16^0 = 109$, onde D em hexadecimal é igual a 13 em decimal.

Obs.: a função **ord** sobre um caracter devolve um número entre 0 e 255 (que pode ser representado em 8 dígitos binários ou 2 dígitos hexadecimais).

4 Estruturas de Repetição

A estrutura de repetição permite que um comando (ou bloco de comandos) seja executado repetidamente até que uma determinada condição de interrupção seja satisfeita.

4.1 Comando For

O comando **For** permite que um comando ou bloco de comandos seja repetido um número específico de vezes. Neste comando uma variável de controle é incrementada ou decrementada de um *valor inicial* em cada iteração até um *valor final*.

A sintaxe do comando **for** que incrementa a variável de controle é dada como:

for variável_de_controle := expressão_1 **to** expressão_2 **do** Comando_ou_bloco_de_comandos;

Para a forma que decrementa a variável de controle, temos a seguinte sintaxe:

for variável_de_controle := expressão_1 **downto** expressão_2 **do** Comando_ou_bloco_de_comandos;

Na figura 15 apresentamos o fluxograma de uma das formas do comando for.

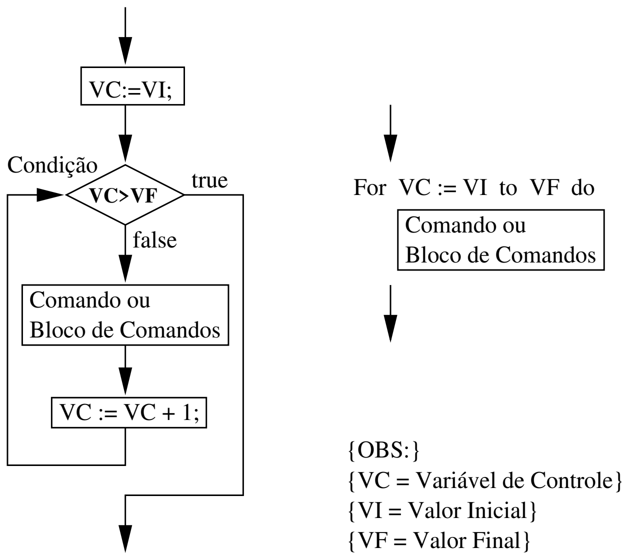


Figura 15: Fluxograma e sintaxe de uma forma do comando **for**.

Exemplo 4.1 *Faça um programa para calcular o fatorial de um determinado número n lido.*

```
program fat;
var n,i,fatorial :integer;
begin
  write('Entre com um número: ');
  readln(n);
  fatorial := 1;
  for i:=2 to n do
    fatorial:=fatorial * i;
  writeln('O fatorial de ',n,' é igual a ',fatorial);
end.
```

Exemplo 4.2 *Faça um programa que lê um valor inteiro positivo n e em seguida lê uma seqüência de n valores reais. O programa deve imprimir o maior valor real da seqüência.*

```
program Maximo;
var
  n,i       : integer;
  x         : real;
  maximo : real;
begin
  ReadLn(n);   {Supõe todos os dados não negativos.}
  maximo := 0.0
  for i:=1 to n do
  begin
    ReadLn(x);
    if x>maximo then maximo := x
  end;
  WriteLn(maximo)
end.
```

Exemplo 4.3 *(Tabuada) Faça um programa que imprima uma tabela, com 9 linhas e 9 colunas. Na interseção da linha i com a coluna j deve conter um valor que é a multiplicação do i com j. Isto é, o programa deve imprimir uma tabela da seguinte forma:*

1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27
4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45
6	12	18	24	30	36	42	48	54
7	14	21	28	35	42	49	56	63
8	16	24	32	40	48	56	64	72
9	18	27	36	45	54	63	72	81

```
program ProgramaTabuada;
var i,j     : integer;
begin
  for i:=1 to 9 do begin
    for j:=1 to 9 do
      write(i*j:3);
    writeln;
  end;
end.
```

Exercício 4.1 *(Tabela de potências) Faça um programa que lê dois inteiros positivos n e k e imprime uma tabela de tamanho $n \times k$ onde a posição (i, j) da tabela contém o número i^j.*

x	x^2	x^3	x^4	x^5
1	1	1	1	1
2	4	8	16	32
3	9	27	81	243
4	16	64	256	1024
5	25	125	625	3125
6	36	216	1296	7776

Exemplo 4.4 *(Triângulo de Floyd) O seguinte triângulo formado por* 6 *linhas de números consecutivos, cada linha contendo um número a mais que na linha anterior, é chamado de Triângulo de Floyd.*

1					
2	*3*				
4	*5*	*6*			
7	*8*	*9*	*10*		
11	*12*	*13*	*14*	*15*	
16	*17*	*18*	*19*	*20*	*21*

Faça um programa que imprime o Triângulo de Floyd com n *linhas (o valor de* n *é lido).*

```
program Floyd;
var i : integer; {índice da linha}
  j : integer; {índice da coluna}
  k : integer; {próximo número}
  m : integer; {número de linhas}
begin
  ReadLn(m);
  k := 0;
  for i:=1 to m do
  begin
    for j:=1 to i do
    begin
      k := k+1;
      Write(k:3)
    end;
    WriteLn
  end
end.
```

Exercício 4.2 *Faça um programa que lê um inteiro positivo* n *e imprime um triângulo constituído por números com o seguinte formato.*

6	*5*	*4*	*3*	*2*	*1*
5	*4*	*3*	*2*	*1*	
4	*3*	*2*	*1*		
3	*2*	*1*			
2	*1*				
1					

No caso a tabela foi impressa com valor de n *igual a 6.*

4.2 Comando While

O comando **while** é uma estrutura de repetição onde a condição de interrupção é testada antes de se executar os comandos a serem repetidos. Na figura 16, apresentamos o fluxograma e a sintaxe do comando While.

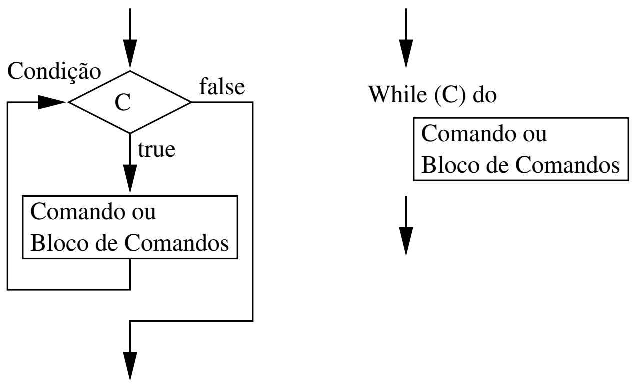


Figura 16: Fluxograma e sintaxe da rotina While.

Exemplo 4.5 *(Validação de entrada) Em determinado momento, um programa deve ler a partir do teclado um número que deve estar* <u>*necessariamente*</u> *no intervalo* $[10, 50]$. *Faça um programa que fique lendo números do teclado e pare quando o usuário entrar com o primeiro número entre* $[10, 50]$.

```
program Validacao;
var n:integer;
begin
  write('Entre com um número no intervalo [10,50]: ');  readln(n);
  while ((n<10) or (n>50)) do begin
    writeln('ERRO: Número inválido.');
    write('Entre com um número no intervalo [10,50]: ');   readln(n);
  end;
  writeln('O número positivo lido foi: ',n);
end.
```

Exemplo 4.6 *(Seqüência de números positivos) Faça um programa que lê uma seqüência de números positivos (pode ser vazia) e seguida pela leitura de um número negativo. O programa deve parar de ler números quando o usuário entrar com o número negativo. O programa deve imprimir a soma, média e quantidade dos números não negativos.*

```
program SequenciaPositivos;
var x,soma    : real;
  nelementos : integer;
begin
  write('Entre com um número: ');    readln(x);
  soma := 0;    nelementos := 0;
  while (x>=0) do begin
    soma := soma + x;        nelementos := nelementos + 1;
    write('Entre com um número: ');     readln(x);
  end;
  if (nelementos>0) then begin
    writeln('A soma dos elementos é: ',soma);
    writeln('A media dos elementos é: ',soma/nelementos);
    writeln('A quantidade de elementos é: ',nelementos);
  end
  else writeln('Não foi lido nenhum elemento positivo.');
end.
```

Exemplo 4.7 *Faça um programa que leia uma quantidade n, (vamos supor que $n \geq 0$) e em seguida o programa deve ler n idades inteiras e então deve imprimir a média das idades lidas.*

```
program MediaIdades;
var x,soma,lidos,n : integer;
begin
  write('Entre com a quantidade de idades a ler: ');
  readln(n);
  lidos:=0;
  soma := 0;
  while (lidos<n) do begin
    write('Entre com uma idade: ');
    readln(x);
    soma := soma + x;
    lidos := lidos + 1;
  end;
  if (lidos >0) then writeln('A média das idades é ',soma/lidos)
  else writeln('Não foi lido nenhuma idade.');
end.
```

Exemplo 4.8 *O cálculo da raiz quadrada de um número positivo n pode ser aproximado usando se a seguinte série:*

$$n^2 = 1 + 3 + 5 + \cdots + (2 \cdot n - 1) = \sum_{k=1}^{n} (2 \cdot k - 1)$$

Se n é um quadrado perfeito, então podemos calcular a raiz usando-se o seguinte código:

```
...
readln(n);
soma := 0;  i := 1;  raiz := 0;
while soma<>n do
  begin
  soma := soma+i;
  i := i+2;
  raiz := raiz+1
  end;
writeln(raiz);
...
```

Caso n não seja um quadrado perfeito podemos obter uma aproximação considerando a parte inteira da raiz. Seja $r = \lfloor\sqrt{n}\rfloor$. Então $r^2 \leq n < (r+1)^2$ Fazendo duas modificações no trecho acima:

```
...
readln(n);
soma := 0;  i := 1;  raiz := 0;
while soma<=n do  { <======= }
  begin
  soma := soma+i;
  i := i+2;
  raiz := raiz+1;
  end;
raiz := raiz−1;     { <======= }
writeln(raiz);
...
```

Note que a atribuição raiz:=raiz-1; *foi feita para voltar o valor da raiz de uma unidade, uma vez que a condição de parada do comando* **while** *é* soma<=n;. *Uma versão que atualiza a variável soma com atraso é apresentada a* *seguir:*

```
...
readln(n);
soma := 0;  i := 1;  raiz := −1;  {atraso na soma}
while soma<=n do
  begin
  soma := soma+i;
  i := i+2;
  raiz := raiz+1;
  end;
writeln(raiz);
...
```

Exercício 4.3 *Faça um programa para calcular a raiz aproximada, como no exemplo 4.8, mas usando o comando* **repeat** *em vez do comando* **while**.

Exercício 4.4 *Para calcular a raiz quadrada de n com uma casa decimal basta calcular a raiz de $100 \cdot n$ como no exemplo 4.8, e dividir o resultado por 10. Faça um programa que calcula a raiz de um valor n $(n \geq 0)$ pelo método do exemplo 4.8, com d casas decimais, (n e d são lidos).*

4.3 Comando Repeat

O comando **repeat** é uma estrutura de repetição onde a condição de interrupção da estrutura de repetição é testada no fim do bloco de repetição. Note que neste comando não é preciso usar **begin** e **end** para especificar os vários comandos a serem repetidos. Na figura 17, apresentamos o fluxograma e a sintaxe do comando Repeat.

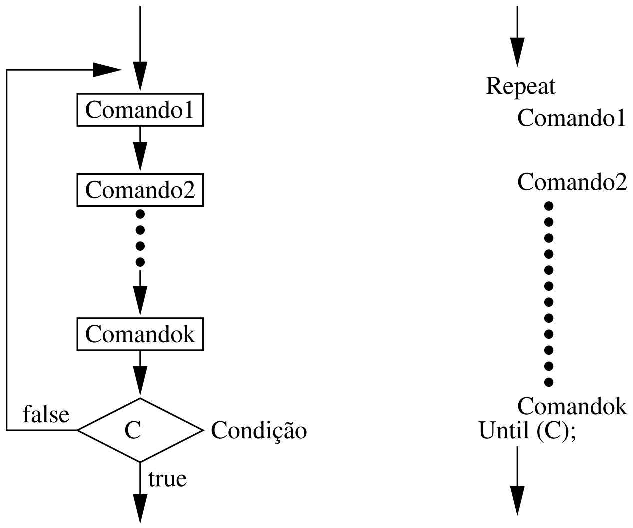


Figura 17: Fluxograma e sintaxe do comando Repeat.

Exemplo 4.9 *(Validação de entrada) Em determinado momento, um programa deve ler a partir do teclado um número que deve estar* *__necessariamente__* *no intervalo* $[10, 50]$. *Faça um programa que fique lendo números do teclado e pare quando o usuário entrar com o primeiro número entre* $[10, 50]$.

```
program Validacao;
var n:integer;
begin
  repeat
    write('Entre com um número no intervalo [10,50]: ');  readln(n);
    if ((n<10) or (n>50)) then writeln('ERRO: Número inválido.');
  until (n>=10) and (n<=50);
  writeln('O número positivo lido foi: ',n);
end.
```

Exemplo 4.10 *A seguir apresentamos a implementação do Algoritmo de Euclides usando o comando* **repeat**.

```
program Euclides;
var x,y,r,m,n : integer;
begin
  Readln(m,n);
  x := m;  y := n;
  repeat
    r := x mod y;
    x := y;  y := r
  until r=0;
  Writeln(x)
end.
```

Exemplo 4.11 *(Seqüência de números positivos) Faça um programa para ler uma seqüência de números positivos (pode ser vazia) e seguido pela leitura de um número negativo. O programa deve parar de ler números quando o usuário entrar com o número negativo. O programa deve imprimir a soma, média e quantidade dos números não negativos.*

```
program SequenciaPositivos2;
var x,soma    : real;
  nelementos : integer;
begin
  soma := 0;   nelementos := 0;
  repeat
    write('Entre com um número: ');    readln(x);
    if (x>=0) then begin
      soma := soma + x;
      nelementos := nelementos + 1;
    end;
  until (x<0);
  if (nelementos>0) then begin
    writeln('A soma dos elementos é: ',soma);
    writeln('A media dos elementos é: ',soma/nelementos);
    writeln('A quantidade de elementos é: ',nelementos);
  end
  else writeln('Não foi lido nenhum elemento positivo.');
end.
```

Exemplo 4.12 *A raiz quadrada de um número positivo $N = N_0$ pode ser calculada pelo método de aproximações sucessivas de Newton. As aproximações $N_1, N_2, \ldots$ são tais que $lim_{i\to\infty} N_i = \sqrt{N_0}$, onde*

$$N_i = \begin{cases} \frac{N_0}{2}, & \textit{se} \quad i = 1, \\ \frac{(N_{i-1})^2 + N_0}{2N_{i-1}}, & \textit{caso contrário}. \end{cases}$$

Faça um programa que calcula a raiz de um número calculando o valor da raiz pelo método de aproximações sucessivas de Newton parando as iterações quando a diferença entre o valor calculado em uma iteração com o valor calculado na iteração anterior seja menor que $0,00001$.

```
program raizquadrada;
var i          : integer;
  n,raiz,raizanterior : real;

begin
  write('Entre com um número: ');
  readln(n);
  raiz := n/2;
  repeat
    raizanterior := raiz;
    raiz := (raiz*raiz + n)/(2*raiz);
  until (abs(raiz−raizanterior) < 0.000001);
  writeln('A raiz de ',n:20:10,' é ',raiz:20:10);
end.
```

Exemplo 4.13 *Faça um programa que escreve individualmente os dígitos de um número inteiro positivo da direita para a esquerda.*

```
program Digitos1;
var n,d       : Integer;
begin
  write('Entre com um número inteiro positivo: ');
  Readln(n);
  repeat
    d := n mod 10;
    n := n div 10;
    Write(d:2)
  until n=0;
  writeln;
end.
```

Exercício 4.5 *Faça um programa que escreve individualmente os dígitos de um número inteiro positivo da esquerda para a direita.*

Exemplo 4.14 *O valor π pode ser calculado através da série $\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \cdots$. Cada termo desta série tem um formato $(-1)^{k+1} \cdot \frac{4}{2k-1}$, $k = 1, 2, \ldots$. A medida que k cresce, o termo vai se tornando cada vez menor, e sua contribuição para o valor de π se torna menor. Faça um programa que calcula o valor de π através da série acima, somando termo a termo, parando quando a diferença absoluta entre o valor de π calculado em uma iteração e o valor de π calculado na iteração anterior for menor que* $0,0001$.

```
program pi;
var pi,piant,termo,sinal : real ;
  i                  : integer;
begin
  pi := 0;
  i := 1;
  sinal := −1;
  termo := 4;
  repeat
    piant := pi;
    pi := pi + termo;
    i := i+2;
    termo := sinal∗4/i;
    sinal := sinal∗(−1);
  until abs(pi−piant) < 0.00001;
  writeln('pi = ',pi);
end.
```

Exercício 4.6 *Um programa deve ler um inteiro positivo n e em seguida ler mais n valores reais sendo que o programa deve imprimir a soma, a média, o menor valor e o maior valor dos n valores reais lidos. Faça três versões deste programa, usando os comandos* **while**, **repeat** *e* **for**.

4.4 Exercícios

1. Faça um programa que descubra um número entre 0 e 1000 imaginado pelo usuário. O programa deve fazer iterações com o usuário. A cada iteração, o programa deve tomar um número e perguntar para o usuário se este número é igual, menor ou maior do que o valor imaginado. O usuário deve responder de forma correta. A execução do programa deve terminar assim que o programa ”adivinhar”o valor imaginado pelo usuário. O programa deve imprimir o número imaginado e o número de perguntas feitas pelo programa. Seu programa não pode fazer mais que 10 perguntas.

2. Faça um programa que leia uma seqüência de números inteiros positivos e termine com um número negativo (este último não deve ser considerado, serve apenas para finalizar a seqüência). O programa deve verificar se os números positivos:

   (a) Estão em ordem crescente.
   (b) Estão em ordem decrescente.
   (c) Se a seqüência é uma progressão aritmética, neste caso dizer a razão.
   (d) Se a seqüência é uma progressão geométrica, neste caso dizer a razão.

3. O desvio padrão *dp* e a variância *var* dos números $x_1, \ldots, x_n$ podem ser calculados usando as seguintes fórmulas

$$\begin{aligned} dp(x_1,\ldots,x_n) &= \sqrt{\frac{1}{n-1}\left(\sum_{i=1}^{n} x_i^2 - \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right)^2\right)} \\ var(x_1,\ldots,x_n) &= (dp(x_1,\ldots,x_n))^2. \end{aligned}$$

   Faça um programa que lê o valor n e a seqüência dos n números reais e depois imprime a média, o desvio padrão e a variância dos n números lidos.
   Obs.: O desvio padrão também pode ser obtido através da fórmula $dp(x_1,\ldots,x_n) = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{X})^2}$, onde $\overline{X}$ é a média dos números $x_1,\ldots,x_n$. Mas note que o cálculo do desvio padrão, através desta fórmula, exige o cálculo prévio da média e portanto os números devem estar todos disponíveis ao mesmo tempo, na memória do computador, para o cálculo do desvio padrão.

4. Um banco faz empréstimos com uma taxa de juros mensal igual a t, $0 < t \leq 1$. Faça um programa que imprime quanto uma pessoa que toma emprestado q reais ficará devendo após m meses. Os valores de q, t e m são lidos.

5 Desenvolvendo Programas

5.1 Programação por Etapas

Quando um programa começa a ficar maior ou mais complicado, é comum se fazer primeiro um esboço, contendo operações em alto nível, mesmo que sejam complexas. Nos passos seguintes este esboço é detalhado em operações mais específicas até que se chegue na codificação do programa na linguagem de programação.

Exemplo 5.1 *Faça um programa que lê três números a, b e c e verifica se estes números formam os lados de um triângulo. Em caso afirmativo, o programa verifica se o triângulo formado é equilátero (três lados iguais), isóceles (dois lados iguais) ou escaleno (três lados diferentes).*

Vamos desenvolver este programa por etapas. Podemos considerar os seguintes passos principais:

Passo 1: *Leia a, b e c.*
Passo 2: *Se* $\boxed{(a, b, c) \text{ não formam um triângulo}}$ *então*
Passo 3: *Escreva: Os valores lidos não formam um triângulo.*
Passo 4: *Senão se* $\boxed{(a, b, c) \text{ formam um triângulo equilátero}}$ *então*
Passo 5: *Escreva: Os valores lidos formam um triângulo equilátero.*
Passo 6: *Senão se* $\boxed{(a, b, c) \text{ formam um triângulo isóceles}}$ *então*
Passo 7: *Escreva: Os valores lidos formam um triângulo isóceles.*
Passo 8: *Senão*
Passo 9: *Escreva: Os valores lidos formam um triângulo escaleno.*

Esta estrutura descreve uma estratégia para resolver o problema, sem se preocupar com cada um dos testes mais complicados (condições que estão destacadas). A tradução dos comandos não destacados para a linguagem Pascal é praticamente direta. Assim, vamos nos preocupar com cada um dos testes destacados.

1. *No passo 2, temos de verificar quando três números não formam os lados de um triângulo. Isto pode ser feito verificando se um dos valores é maior que a soma dos outros dois valores. Assim, podemos trocar a condição do passo 2 por:*
$$\boxed{(a > b + c) \text{ ou } (b > a + c) \text{ ou } (c > a + b)}$$
2. *No passo 4 devemos testar se os valores a, b e c formam um triângulo equilátero. Isto é verdadeiro se os três valores são iguais e pode ser feito usando duas relações de igualdade.*
$$\boxed{(a = b) \text{ e } (b = c)}$$
3. *No passo 6 devemos testar se os valores a, b e c formam um triângulo isóceles. Neste caso devemos verificar se qualquer dois dos três valores são iguais.*
$$\boxed{(a = b) \text{ ou } (a = c) \text{ ou } (b = c)}$$

Substituindo estas condições no algoritmo acima e traduzindo para a Linguagem Pascal, temos:

```
program triangulo;
var a,b,c: real;
begin
    write('Entre com o primeiro lado: ');   readln(a);
    write('Entre com o segundo lado: ');    readln(b);
    write('Entre com o terceiro lado: ');   readln(c);
    if ((a > b + c) or (b > a + c) or (c > a + b))
      then  {não existe triângulo}
         writeln('Os números não formam um triângulo.')
      else
         if (a = b) and (a = c)
           then  {é triângulo equilátero}
              writeln('O triângulo é equilátero')
           else
              if (a = b) or (a = c) or (b = c)
                then  {é triângulo isóceles}
                   writeln('O triângulo é isóceles)
                else  {é triângulo escaleno}
                   writeln('O triângulo é escaleno)
    end.
```

Exemplo 5.2 *Programa para ler 3 números n_1, n_2 e n_3 e imprimir (n_1, n_2, n_3) de maneira a ficarem em ordem não decrescente de valor.*

Um primeiro esboço, que apresenta uma estratégia para se resolver o problema, é dada a seguir:

1. *Ler os três números nas variáveis $n1, n2, n3$.*
2. *Trocar os valores das variáveis $n1, n2, n3$, de forma que $n3$ contém um maior valor entre $n1, n2, n3$. Neste momento, $n3$ contém um valor correto.*
3. *Trocar os valores das variáveis $n1, n2$, de forma que $n2$ contém um maior valor entre $n1, n2$.*
4. *Imprimir $(n1, n2, n3)$.*

Em um segundo passo, podemos descrever a estratégia acima de forma mais detalhada, omitindo alguns passos ainda desconhecidos.

```
program ordena3;
var n1, n2, n3: real;
begin
    write('Entre com o primeiro número: ');   readln(n1);
    write('Entre com o segundo número: ');    readln(n2);
    write('Entre com o terceiro número: ');   readln(n3);
    if (n3 < n1)
      then [troca valores de n3 e n1]
    if (n3 < n2)
      then [troca valores de n3 e n2]
    {Neste ponto, n3 contém um maior valor dos três valores lidos.}
    if (n2 < n1)
      then [troca valores de n2 e n1]
    {Neste ponto, n2 contém um segundo maior valor dos três valores lidos.}
    {Após acertar os valores em n3 e n2, observe que n1 contém um menor valor dos três lidos.}
    writeln(n1, n2, n3);
end.
```

Agora considere o problema de se trocar os valores contidos em apenas duas variáveis. Digamos que você queira trocar os valores das variáveis A e B. Não é possível passar o valor de A para B diretamente, já que isto faria com que perdêssemos o valor original de B. Da mesma forma, também não é possível passar o valor de B para A diretamente, sem perda do valor original de A. Isto nos induz a usar uma outra variável auxiliar, AUX, para primeiro guardar o valor de B, em seguida passar o valor de A para B, e depois passar o valor de AUX para A. Isto nos dá a seguinte seqüência de operações:

$$AUX \leftarrow B; \quad B \leftarrow A; \quad A \leftarrow AUX;$$

Substituindo este processo de troca dos valores em variáveis no pseudo programa acima, obtemos o seguinte programa final:

```
program ordena3;
var n1,n2,n3, AUX: real;
begin
  write('Entre com o primeiro número: '); readln(n1);
  write('Entre com o segundo número: '); readln(n2);
  write('Entre com o terceiro número: '); readln(n3);
  if (n3<n1) then begin
    AUX:= n3;   n3:= n1;   n1:= AUX;
  end;
  if (n3<n2) then  begin
    AUX:= n3;   n3:= n2;   n2:= AUX;
  end;
  if (n2<n1) then begin
    AUX:= n2;   n2:= n1;   n1:= AUX;
  end;
  writeln(n1,n2,n3);
end.
```

Exercício 5.1 *Descreva um programa usando a mesma estratégia do exemplo 5.2 para ordenar quatro números nas variáveis $n1$, $n2$, $n3$ e $n4$.*

Exercício 5.2 *A estratégia do exemplo 5.2 foi a de colocar o maior dos valores lidos na variável correta (acertar o valor em $n3$) e em seguida aplicar o mesmo método para acertar as variáveis restantes (variáveis $n1$ e $n2$). Faça um programa para ordenar três números como no exemplo 5.2, mas usando a estratégia de colocar o menor dos valores lidos na variável correta (variável $n1$) e reaplicar o mesmo método para as demais variáveis (variáveis $n2$ e $n3$).*

5.2 Simulação de programas

Uma simulação de um programa nada mais é que simular o comportamento de um programa em um computador. As simulações que faremos tem por objetivo verificar se o programa está realmente correto ou encontrar o erro de um programa incorreto.

É aconselhável que se faça sempre uma simulação do programa antes de implementá-lo no computador. Isto permitirá que o programador possa tratar possíveis erros no momento em que estiver projetando o programa e saber como consertá-lo mais facilmente uma vez que as idéias e estratégias usadas no desenvolvimento do programa estarão mais "frescas" em sua memória.

Muitos compiladores que integram ambientes de editoração possibilitam a execução de um programa passo a passo e a observação de todo o processamento dos dados existentes na memória. Com isso, as simulações podem ser feitas diretamente neste software integrado. Outros compiladores podem inserir informações dentro do código executável do programa contendo as informações necessárias para se fazer uma execução passo a passo no programa fonte.

Para fazer uma simulação em papel, precisamos ter o programa e uma "memória" em papel. Nesta simulação você estará simulando o computador acompanhando o programa e processando os dados que estão na memória de papel. Isto é, você fará o trabalho da CPU. A seguir iremos descrever como fazer esta simulação.

Primeiramente, descreva uma tabela, colocando na primeira linha os nomes de todas as variáveis declaradas no programa, como apresentado abaixo. Caso seja uma função ou procedimento (veja seção 7) não esqueça de colocar também os parâmetros declarados nestas rotinas.

Variável-1	Variável-2	···	Variável-K	Parâmetro-1	···	Parâmetro-M

Em seguida coloque os valores iniciais destas variáveis e parâmetros. Caso não tenha sido atribuído nenhum valor a elas, coloque a palavra *lixo* (já que é um valor que você desconhece). No caso de parâmetros, coloque os valores que foram passados como parâmetros (alguns podem ser lixo também). Qualquer operação que utilize o valor *lixo* também resulta em *lixo*.

Variável-1	Variável-2	Variável-3	Parâmetro-1	Parâmetro-2
lixo	*lixo*	*lixo*	*valor-param1*	*lixo*

Vamos fazer a simulação de um programa para ler uma seqüência de números inteiros não negativos terminada com um número inteiro negativo. O programa deve imprimir a soma e quantidade dos números não negativos.

```
program Sequencia;
var x,soma,n : integer;
begin
    write('Entre com um número: ');
    readln(x);
    soma := 0;
    n := 0;
    while (x>=0) do begin
        soma := soma + x;
        n := n + 1;
        write('Entre com um número: ');
        readln(x);
    end;
    writeln('Soma=',soma,' Quantidade=',n);
end.
```

Marcaremos o comando que acabou de ser executado com uma marca → (inicialmente marcado no **begin** do bloco de comandos do programa principal). Vamos supor que a entrada é a seqüência: 1,2,3,-1.

```
 1.  program Sequencia;
 2.  var x,soma,n : integer;
→    begin
 4.      write('Entre com um número:  ');
 5.      readln(x);
 6.      soma := 0;
 7.      n := 0;
 8.      while (x>=0) do begin
 9.          soma := soma + x;
10.          n := n + 1;
11.          write('Entre com um número:  ');
12.          readln(x);
13.      end;
14.      writeln('Soma=',soma,' Quantidade=',n);
15.  end.
```

x	**Soma**	**n**
lixo	*lixo*	*lixo*

Início da Simulação
Seqüência a testar:
1,2,3,-1

```
 1.  program Sequencia;
 2.  var x,soma,n : integer;
 3.  begin
 4.      write('Entre com um número:  ');
→        readln(x);
 6.      soma := 0;
 7.      n := 0;
 8.      while (x>=0) do begin
 9.          soma := soma + x;
10.          n := n + 1;
11.          write('Entre com um número:  ');
12.          readln(x);
13.      end;
14.      writeln('Soma=',soma,' Quantidade=',n);
15.  end.
```

x	**Soma**	**n**
~~*lixo*~~	*lixo*	*lixo*
1		

Execução dos
passos 4 e 5.
Leitura do valor 1.

```
 1.  program Sequencia;
 2.  var x,soma,n : integer;
 3.  begin
 4.      write('Entre com um número:  ');
 5.      readln(x);
 6.      soma := 0;
→        n := 0;
 8.      while (x>=0) do begin
 9.          soma := soma + x;
10.          n := n + 1;
11.          write('Entre com um número:  ');
12.          readln(x);
13.      end;
14.      writeln('Soma=',soma,' Quantidade=',n);
15.  end.
```

x	**Soma**	**n**
~~*lixo*~~	~~*lixo*~~	~~*lixo*~~
1	0	0

Execução dos
passos 6 e 7.
Inicialização de
Soma e *n*.

```
 1.  program Sequencia;
 2.  var x,soma,n : integer;
 3.  begin
 4.      write('Entre com um número:  ');
 5.      readln(x);
 6.      soma := 0;
 7.      n := 0;
→        while (x>=0) do begin
 9.          soma := soma + x;
10.          n := n + 1;
11.          write('Entre com um número:  ');
12.          readln(x);
13.      end;
14.      writeln('Soma=',soma,' Quantidade=',n);
15.  end.
```

x	**Soma**	**n**
~~*lixo*~~	~~*lixo*~~	~~*lixo*~~
1	0	0

Teste da condição
do **while**.
Elemento lido é $\geq$ 0 ?
$1 \geq 0$?
Sim. Entrar no *loop*.

```
 1.  program Sequencia;
 2.  var x,soma,n : integer;
 3.  begin
 4.      write('Entre com um número:  ');
 5.      readln(x);
 6.      soma := 0;
 7.      n := 0;
 8.      while (x>=0) do begin
 9.          soma := soma + x;
→            n := n + 1;
11.          write('Entre com um número:  ');
12.          readln(x);
13.      end;
14.      writeln('Soma=',soma,' Quantidade=',n);
15.  end.
```

x	**Soma**	**n**
~~*lixo*~~	~~*lixo*~~	~~*lixo*~~
1	~~0~~	~~0~~
	1	1

Execução dos passos 9 e 10. Acumular valor em Soma. Incrementar n.

```
 1.  program Sequencia;
 2.  var x,soma,n : integer;
 3.  begin
 4.      write('Entre com um número:  ');
 5.      readln(x);
 6.      soma := 0;
 7.      n := 0;
 8.      while (x>=0) do begin
 9.          soma := soma + x;
10.          n := n + 1;
11.          write('Entre com um número:  ');
→            readln(x);
13.      end;
14.      writeln('Soma=',soma,' Quantidade=',n);
15.  end.
```

x	**Soma**	**n**
~~*lixo*~~	~~*lixo*~~	~~*lixo*~~
~~1~~	~~0~~	~~0~~
2	1	1

Execução dos passos 11 e 12. Leitura do valor 2.

```
 1.  program Sequencia;
 2.  var x,soma,n : integer;
 3.  begin
 4.      write('Entre com um número:  ');
 5.      readln(x);
 6.      soma := 0;
 7.      n := 0;
→        while (x>=0) do begin
 9.          soma := soma + x;
10.          n := n + 1;
11.          write('Entre com um número:  ');
12.          readln(x);
13.      end;
14.      writeln('Soma=',soma,' Quantidade=',n);
15.  end.
```

x	**Soma**	**n**
~~*lixo*~~	~~*lixo*~~	~~*lixo*~~
~~1~~	~~0~~	~~0~~
2	1	1

Teste da condição do **while**. Elemento lido é ≥ 0 ? $2 \geq 0$? Sim. Entrar no *loop*.

```
 1.  program Sequencia;
 2.  var x,soma,n : integer;
 3.  begin
 4.      write('Entre com um número:  ');
 5.      readln(x);
 6.      soma := 0;
 7.      n := 0;
 8.      while (x>=0) do begin
 9.          soma := soma + x;
→            n := n + 1;
11.          write('Entre com um número:  ');
12.          readln(x);
13.      end;
14.      writeln('Soma=',soma,' Quantidade=',n);
15.  end.
```

x	**Soma**	**n**
~~*lixo*~~	~~*lixo*~~	~~*lixo*~~
~~1~~	~~0~~	~~0~~
2	~~1~~	~~1~~
	3	2

Execução dos passos 9 e 10. Acumular valor em Soma. Incrementar n.

```
 1.  program Sequencia;
 2.  var x,soma,n : integer;
 3.  begin
 4.      write('Entre com um número:  ');
 5.      readln(x);
 6.      soma := 0;
 7.      n := 0;
 8.      while (x>=0) do begin
 9.          soma := soma + x;
10.          n := n + 1;
11.          write('Entre com um número:  ');
→            readln(x);
13.      end;
14.      writeln('Soma=',soma,' Quantidade=',n);
15.  end.
```

x	**Soma**	**n**
~~*lixo*~~	~~*lixo*~~	~~*lixo*~~
~~1~~	~~0~~	~~0~~
~~2~~	~~1~~	~~1~~
3	3	2

Execução dos passos 11 e 12. Leitura do valor 3.

```
 1.  program Sequencia;
 2.  var x,soma,n : integer;
 3.  begin
 4.      write('Entre com um número:  ');
 5.      readln(x);
 6.      soma := 0;
 7.      n := 0;
 →       while (x>=0) do begin
 9.          soma := soma + x;
10.          n := n + 1;
11.          write('Entre com um número:  ');
12.          readln(x);
13.      end;
14.      writeln('Soma=',soma,' Quantidade=',n);
15.  end.
```

x	Soma	n
~~*lixo*~~	~~*lixo*~~	~~*lixo*~~
~~1~~	~~0~~	~~0~~
~~2~~	~~1~~	~~1~~
3	3	2

Teste da condição
do **while**.
Elemento lido é $\geq$ 0 ?
$3 \geq 0$?
Sim. Entrar no *loop*.

```
 1.  program Sequencia;
 2.  var x,soma,n : integer;
 3.  begin
 4.      write('Entre com um número:  ');
 5.      readln(x);
 6.      soma := 0;
 7.      n := 0;
 8.      while (x>=0) do begin
 9.          soma := soma + x;
 →           n := n + 1;
11.          write('Entre com um número:  ');
12.          readln(x);
13.      end;
14.      writeln('Soma=',soma,' Quantidade=',n);
15.  end.
```

x	Soma	n
~~*lixo*~~	~~*lixo*~~	~~*lixo*~~
~~1~~	~~0~~	~~0~~
~~2~~	~~1~~	~~1~~
3	~~3~~	~~2~~
	6	3

Execução dos
passos 9 e 10.
Acumular valor em Soma.
Incrementar n.

```
 1.  program Sequencia;
 2.  var x,soma,n : integer;
 3.  begin
 4.      write('Entre com um número:  ');
 5.      readln(x);
 6.      soma := 0;
 7.      n := 0;
 8.      while (x>=0) do begin
 9.          soma := soma + x;
10.          n := n + 1;
11.          write('Entre com um número:  ');
 →           readln(x);
13.      end;
14.      writeln('Soma=',soma,' Quantidade=',n);
15.  end.
```

x	Soma	n
~~*lixo*~~	~~*lixo*~~	~~*lixo*~~
~~1~~	~~0~~	~~0~~
~~2~~	~~1~~	~~1~~
~~3~~	~~3~~	~~2~~
-1	6	3

Execução dos
passos 11 e 12.
Leitura do valor -1.

```
 1.  program Sequencia;
 2.  var x,soma,n : integer;
 3.  begin
 4.      write('Entre com um número:  ');
 5.      readln(x);
 6.      soma := 0;
 7.      n := 0;
 →       while (x>=0) do begin
 9.          soma := soma + x;
10.          n := n + 1;
11.          write('Entre com um número:  ');
12.          readln(x);
13.      end;
14.      writeln('Soma=',soma,' Quantidade=',n);
15.  end.
```

x	Soma	n
~~*lixo*~~	~~*lixo*~~	~~*lixo*~~
~~1~~	~~0~~	~~0~~
~~2~~	~~1~~	~~1~~
~~3~~	~~3~~	~~2~~
-1	6	3

Teste da condição
do **while**.
Elemento lido é $\geq$ 0 ?
$-1 \geq 0$?
Não. Ir para o próximo
comando depois do *loop*.

```
 1.  program Sequencia;
 2.  var x,soma,n : integer;
 3.  begin
 4.      write('Entre com um número:  ');
 5.      readln(x);
 6.      soma := 0;
 7.      n := 0;
 8.      while (x>=0) do begin
 9.          soma := soma + x;
10.          n := n + 1;
11.          write('Entre com um número:  ');
12.          readln(x);
13.      end;
 →       writeln('Soma=',soma,' Quantidade=',n);
15.  end.
```

x	Soma	n
~~*lixo*~~	~~*lixo*~~	~~*lixo*~~
~~1~~	~~0~~	~~0~~
~~2~~	~~1~~	~~1~~
~~3~~	~~3~~	~~2~~
-1	6	3

Impressão de *Soma* e n.
Soma = 6
n = 3

5.3 Alinhamento de Comandos

Nesta seção apresentamos algumas formas de alinhamento que visam a melhor visualização dos comandos (ou declarações) que estão sob atuação de outros. Estas permitem que programas fiquem melhor apresentados e mais fáceis de se entender. A idéia é fazer com que os comandos que estão sob atuação de outros estejam alinhados e deslocados alguns espaços a direita.

A seguir apresentamos algumas exemplos de alinhamento para alguns comandos. O leitor não necessariamente precisa seguir esta regra, mas é importante que se use alguma regra razoável de alinhamento dos comandos.

Obs.: As barras verticais são apenas para visualizar o alinhamento dos comandos.

- Declarações: const, type e var.

const
  | $MAX = 101;$
  | $PI = 3.14158;$

type
  | $MeuInteiro = \textbf{integer};$
  | $TipoNome = \textbf{string}[\textbf{50}];$

var
  | $a, b, c: \ \textbf{integer};$
  | $x, y, z: \ \textbf{real};$

- Bloco de Comandos:

begin
  | *Comando1*;
  | *Comando2*;
  | ⋮
  | *ComandoN*;
end;

- Comando if-then, if-then-else.

if *(Condição)* **then**
  | *Comando*;

if *(Condição)* **then begin**
  | $Comando_1$;
  | ⋮
  | $Comando_N$;
end;

if *(Condição)* **then begin**
  | $Comando_1$;
  | ⋮
  | $Comando_N$;
end else begin
  | $Comando_{N+1}$;
  | ⋮
  | $Comando_{N+M}$;
end;

- Comando for

for *i:=(Expressão)* **to** *(Expressão)* **do**
  | *Comando*;

for *i:=(Expressão)* **to** *(Expressão)* **do begin**
  | $Comando_1$;
  | ⋮
  | $Comando_N$;
end;

- Comando while

while *(Condição)* **do**
  | *Comando*;

while *(Condição)* **do begin**
  | $Comando_1$;
  | ⋮
  | $Comando_N$;
end;

- Comando repeat

repeat
  | $Comando_1$;
  | ⋮
  | $Comando_N$;
until *(Condição);*

Alinhamento de comandos encaixados

Quando um comando é colocado internamente a outro comando, o comando interno e todos sobre sua atuação são deslocados adicionalmente ao alinhamento do comando externo.

Exemplo 5.3 *No exemplo a seguir apresentamos o esboço de um programa com alguns comandos encaixados em outros e o alinhamento em cada um deles.*

```
program EsboçoAlinhamento;
var
    | a, b, c : integer;
    | x, y, z : real;
begin
  | if (Condição) then begin
  |   | Comando_1;
  |   | for i:=(Expressão) to (Expressão) do begin
  |   |   | ComandoF_1;
  |   |   |   ⋮
  |   |   | ComandoF_N;
  |   | end;
  |   | if(Condição) then begin
  |   |   | ComandoI_1;
  |   |   | for i:=(Expressão) to (Expressão) do begin
  |   |   |   | ComandoIF_1;
  |   |   |   |   ⋮
  |   |   |   | ComandoIF_N;
  |   |   | end;
  |   |   | while (Expressão) do begin
  |   |   |   | if (Condição) then begin
  |   |   |   |   | ComandoIFT_1;
  |   |   |   |   |   ⋮
  |   |   |   |   | ComandoIFT_N;
  |   |   |   | end else begin
  |   |   |   |   | ComandoIFE_{N+1};
  |   |   |   |   |   ⋮
  |   |   |   |   | ComandoIFE_{N+M};
  |   |   |   | end;
  |   |   |   | ComandoIW_1;
  |   |   |   |   ⋮
  |   |   |   | ComandoIW_N;
  |   |   | end;
  |   | end;
  | end;
  | repeat
  |   | ComandoR_1;
  |   |   ⋮
  |   | ComandoR_K;
  | until (Condição);
end.
```

5.4 Desenvolvimento de Algoritmos Eficientes

Em geral teremos inúmeros algoritmos para se resolver o mesmo problema. Alguns deles podem ser rápidos, enquanto outros podem exigir mais processamento e conseqüêntemente podem resultar em programas lentos ou até mesmo inviáveis de serem executados. Outros algoritmos podem ser rápidos mas podem consumir muita memória, deixando o programa igualmente inviável.

Ao desenvolver um algoritmo, precisamos considerar os propósitos do algoritmo juntamente com os recursos computacionais usados por ele. Dois dos principais recursos computacionais são o *tempo de processamento* e a *memória*.

Outra consideração importante é o grau de facilidade de desenvolvimento e manutenção do programa. Programas escritos de maneira simples são em geral mais fáceis de se dar manutenção do que aqueles que usam estruturas complexas e códigos otimizados.

No exemplo seguinte consideramos algumas versões de algoritmos para o problema de se verificar se um número é primo ou não.

Exemplo 5.4 *Faça um programa para verificar se um número positivo, lido pelo programa, é primo ou não. Obs.: Um número positivo maior que 1 é primo se é divisível somente por ele mesmo e pela unidade.*

```
Program Primo1;
var i,n       : integer;
  EhPrimo : Boolean;
begin
  writeln('Programa para verificar se um número positivo é primo ou não.');
  write('Entre com o número a testar: ');
  readln(n);
  EhPrimo:= True;
  for i:=2 to n−1 do
    if (n mod i = 0) then EhPrimo := false;
  if (EhPrimo) and (n>1)
    then writeln('O número ',n,' é primo.')
    else writeln('O número ',n,' não é primo.')
end.
```

O programa acima é bem simples e compacto. É fácil ver que ele está correto, observando apenas a definição de número primo, não ser divisível por nenhum número entre $\{2, \ldots, n-1\}$, e do comando de repetição que percorre estes números testando se n é divisível por um deles. Além disso, este programa usa pouca memória, uma vez que usamos um pequeno número de variáveis. Por outro lado, se n é um número grande, a quantidade de iterações é no pior casos igual a $n-2$.

Um outro extremo é quando temos já armazenados todos os números representáveis de nosso interesse (por exemplo: os números que podem ser armazenados em um tipo inteiro de 2 bytes), dizendo se cada um é primo ou não. Note que isto já exigiu um grande pré-processamento para todos os números, mas uma vez feito isso, todas as consultas poderão ser feitas em tempo constante e portanto extremamente rápido. É fácil ver que este tipo de solução pode ser inviável devido a grande memória usada para armazenar todos estes números. Uma solução menos drástica seria armazenar todos os primos representáveis (em vez de todos os números). Mas mesmo com esta redução a quantidade de primos existentes é grande, além disso agora não podemos mais fazer o teste em tempo constante (é possível fazer em tempo proporcional a $\log(m)$, onde m é a quantidade de primos).

Outra estratégia interessante é testar primeiro para um subconjunto fixo de primos (quantidade constante). Por exemplo, vamos testar inicialmente se n é divisível por 2 (i.e., n é par); em caso positivo, n não é primo, caso contrário testamos apenas para os elementos ímpares dos números em $\{3, \ldots, n-1\}$. Isto já nos dá um algoritmo que faz, no pior caso, em torno da metade do processamento do primeiro algoritmo.

Uma estratégia interessante é estudarmos mais sobre as características do problema para se tentar otimizar o programa. Note que se um número n não é primo, então deve ser possível escrever n como $n = a \cdot b$. Sem perda de generalidade, considere $a \leq b$. Para verificar se n não é primo, basta achar um dos divisores. Então se encontramos a, não é mais necessário testar para os outros números maiores que a. Além disso, podemos usar o fato de a ser *limitado por $\sqrt{n}$. Este fato é importante uma vez que para valores grandes de n, $\sqrt{n}$ é bem menor que n, o que reduz bastante a quantidade de testes feito pelo programa. O seguinte programa apresenta a implementação desta idéia.*

```
Program Primo2;
var i,n      : integer;
  Max        : real;
  EhPrimo : Boolean;
begin
  writeln('Programa para verificar se um número positivo é primo ou não.');
  write('Entre com o número a testar: ');
  readln(n);
  if (n = 2) then EhPrimo := true
  else if (n mod 2 = 0) or (n<=1) then EhPrimo:= False
  else begin
    EhPrimo:= True;
    i:=3;
    Max := sqrt(n);
    while (i<= Max) and (n mod i <>0) do i := i+2;
    if (i<=Max) and (n mod i = 0) then EhPrimo := false;
  end;
  if EhPrimo
    then writeln('O número ',n,' é primo.')
    else writeln('O número ',n,' não é primo.');
end.
```

Note que se $n = 1031$ (n é primo), o primeiro programa faz 1029 iterações, no comando de repetição, e o programa acima, que também faz uso de uma quantidade constante de memória, faz apenas 15 iterações.

Exercício 5.3 *Faça um programa que verifica se um número é primo, como no programa Primo2, mas além de pular os múltiplos de 2, também pula os múltiplos de 3.*

5.5 Precisão Numérica e Erros de Precisão

A linguagem Pascal nos oferece o tipo real para trabalharmos com números. Em geral este tipo é implementado internamente com uma quantidade fixa de bytes, o que torna impossível a representação exata de todos os números. Isto é razoável uma vez que há infinitos números reais, mesmo em intervalos pequenos como $[0, 1]$.

Apesar disso, a maioria das implementações representa o número $\frac{1}{3}$ internamente com uma boa quantidade de casas decimais corretas. Por exemplo, se $\frac{1}{3}$ for representado como $0,33333333333333315$ teremos o valor correto até a 16ª casa decimal. Na maioria das aplicações este erro na precisão pode ser desprezível. Um dos principais problemas com erros de precisão é o cálculo entre valores que já contêm erros de precisão, que podem gerar resultados com erros maiores. O seguinte programa mostra como o erro de precisão em apenas uma conta faz com que tenhamos um resultado totalmente inesperado.

```
program Terco;
var i      : integer;
  terco : real;
begin
  terco := 1/3;
  for i:=1 to 30 do begin
    terco := 4 * terco - 1;
    writeln('Iteração: ',i:2,', Terco = ',terco:20:18);
  end;
end.
```

Se não houvesse erros de precisão, a variável *Terco* começaria com o valor $\frac{1}{3}$ e em cada iteração o valor de *Terco* é atualizado como $\boxed{4 \cdot Terco - 1}$. Se *Terco* tivesse o valor $\frac{1}{3}$, o novo valor de *Terco* seria $4 \cdot \frac{1}{3} - 1$ que é igual a $\frac{1}{3}$. Isto é, em cada iteração *Terco* recebe novamente o valor $\frac{1}{3}$.

No entanto, a maioria dos programas executáveis construídos a partir deste programa fonte deve imprimir, na última iteração, um valor que é totalmente diferente de $\frac{1}{3}$.

Vamos analisar este programa com mais detalhes. Note que $\frac{1}{3}$ não deve ser armazenado de forma exata, sendo armazenado correto até certa casa decimal. Portanto, há um pequeno erro de precisão na primeira atribuição de *Terco*. Vamos supor que o valor realmente atribuído seja igual a $\frac{1}{3} - \epsilon$, onde $\epsilon > 0$ é este pequeno erro de precisão. A seguir, vamos computar os valores de *Terco* em cada iteração:

Iteração	*Terco*
Início	$= \frac{1}{3} - \epsilon$
1	$= 4 \cdot Terco - 1$ $= 4 \cdot \left(\frac{1}{3} - \epsilon\right) - 1$ $= \frac{1}{3} - 4\epsilon$
2	$= 4 \cdot Terco - 1$ $= 4 \cdot \left(\frac{1}{3} - 4 \cdot \epsilon\right) - 1$ $= \frac{1}{3} - 4^2\epsilon$
3	$= 4 \cdot Terco - 1$ $= 4 \cdot \left(\frac{1}{3} - 4^2 \cdot \epsilon\right) - 1$ $= \frac{1}{3} - 4^3\epsilon$
$\vdots$	$\vdots$
30	$= 4 \cdot Terco - 1$ $= 4 \cdot \left(\frac{1}{3} - 4^{29} \cdot \epsilon\right) - 1$ $= \frac{1}{3} - 4^{30}\epsilon$

Note que 4^{30} é um número muito grande, e mesmo que ϵ seja bem pequeno, o valor $4^{30}\epsilon$ chega a ser maior que $\frac{1}{3}$, resultando em um valor negativo para *Terco*. A função $f(n) = 4^n$ é uma função exponencial e tem um crescimento extremamente rápido. A medida que n cresce a função $g(n) := \frac{1}{3} - 4^n\epsilon$ é dominada pelo termo $4^n\epsilon$. Assim, o valor $g(n)$ rapidamente se torna negativo.

A seguinte tabela apresenta a impressão gerada pela execução deste programa (o programa executável foi gerado em um computador Sun-Sparc 4 com o compilador *gpc – Gnu Pascal Compiler*).

Iteração:	1,	Terco = 0,3333333333333325931847
Iteração:	2,	Terco = 0,3333333333333303727386
Iteração:	3,	Terco = 0,3333333333333214909544
Iteração:	4,	Terco = 0,3333333333332859638176
Iteração:	5,	Terco = 0,3333333333331438552705
Iteração:	6,	Terco = 0,3333333333325754210819
Iteração:	7,	Terco = 0,3333333333303016843274
Iteração:	8,	Terco = 0,3333333333212067373097
Iteração:	9,	Terco = 0,3333333332848269492388
Iteração:	10,	Terco = 0,3333333331393077969551
Iteração:	11,	Terco = 0,3333333325572311878204
Iteração:	12,	Terco = 0,3333333302289247512817
Iteração:	13,	Terco = 0,3333333209156990051270
Iteração:	14,	Terco = 0,3333332836627960205078
Iteração:	15,	Terco = 0,3333331346511840820312
Iteração:	16,	Terco = 0,3333325386047363281250
Iteração:	17,	Terco = 0,3333301544189453125000
Iteração:	18,	Terco = 0,3333206176757812500000
Iteração:	19,	Terco = 0,3332824707031250000000
Iteração:	20,	Terco = 0,3331298828125000000000
Iteração:	21,	Terco = 0,3325195312500000000000
Iteração:	22,	Terco = 0,3300781250000000000000
Iteração:	23,	Terco = 0,3203125000000000000000
Iteração:	24,	Terco = 0,2812500000000000000000
Iteração:	25,	Terco = 0,3125000000000000000000
Iteração:	26,	Terco = 0,2500000000000000000000
Iteração:	27,	Terco = 0,0000000000000000000000
Iteração:	28,	Terco = -1,0000000000000000000000
Iteração:	29,	Terco = -5,0000000000000000000000
Iteração:	30,	Terco = -21,0000000000000000000000

Observações:

1. Caso a primeira atribuição de *Terco* seja algo como $\frac{1}{3}+\epsilon$, $\epsilon > 0$, na última iteração teríamos algo como $\frac{1}{3}+4^{30}\epsilon$, que igualmente daria um valor inesperado para *Terco*, mas desta vez positivo.
2. Note que o único lugar onde consideramos um erro de precisão foi na primeira atribuição de *Terco* e todas as outras atribuições envolveram cálculos exatos.

5.6 Tipos definidos pelo programador

Além dos tipos já definidos na linguagem Pascal, é possível se construir novos tipos associados a apenas um identificador. Assim, poderemos declarar tipos complicados e associar este tipo com um identificador. A definição destes novos tipos deve ser feita na área de declarações, usando a palavra reservada **type** seguida das definições de cada tipo. Cada tipo novo é definido como:

IdentificadorDeTipo = especificação_do_tipo;

Exemplo 5.5 *Declare tipos chamados MeuInteiro, TipoNome e TipoNumero, onde MeuInteiro é igual ao tipo* **integer**, *TipoNome é igual ao tipo* **string[50]** *e TipoNumero é igual ao* **real**.

```
type
    MeuInteiro  = integer;
    TipoNome    = string[50];
    TipoNumero  = real;
```

Exemplo 5.6 *Os seguintes programas são equivalentes:*

```
program ExemploTipos;
type
    TipoNome     = string[50];
    MeuInteiro = integer;
var
    Idade : MeuInteiro;
    Nome  : TipoNome;
begin
  write('Entre com seu nome: ');
  readln(Nome);
  write('Entre com sua idade: ');
  readln(Idade);
  writeln(Nome,' tem ',idade,' anos.');
end.
```

```
program ExemploTipos;
var
    Idade : integer;
    Nome  : string[50];
begin
  write('Entre com seu nome: ');
  readln(Nome);
  write('Entre com sua idade: ');
  readln(Idade);
  writeln(Nome,' tem ',idade,' anos.');
end.
```

Exemplo 5.7 *Outros exemplos de declarações de tipos e variáveis.*

```
program Exemplo;
type
    TipoNome    = string[50];
    TipoRG      = string[15];
    TipoIdade   = integer;
    TipoSalario = real;
var
    NomePedro,NomePaulo       : TipoNome;
    IdadePedro,IdadePaulo     : TipoIdade;
    RgPedro,RgPaulo           : TipoRG;
    SalarioPedro,SalarioPaulo : TipoSalario;
```

Algumas vantagens e necessidades de se usar tipos:

- Basta se lembrar do nome do tipo, sem precisarmos escrever toda a especificação de um novo objeto a ser declarado. Um exemplo disto é o caso do *TipoRG* usado no exemplo acima, a cada vez que formos declarar um RG, não precisaremos nos lembrar se um RG é declarado com 15 ou 16 ou mais caracteres; esta preocupação já foi considerada no momento da especificação do tipo *TipoRG*.

- As re-especificações de tipos ficam mais fáceis. Usando **string[15]** em todos os lugares onde declaramos um RG e se quisermos mudar para **string[20]**, devemos percorrer todo o programa procurando pelas declarações de RG's e fazendo as devidas modificações (note que isto não pode ser feito de qualquer forma, já que nem todo lugar onde aparece **string[15]** é uma declaração de RG. Usando tipos, basta mudar apenas uma vez, i.e., na especificação de *TipoRG*.
- Além disso há tipos de objetos da linguagem Pascal que não admitem declarações usando mais que uma palavra. Alguns tipos de objetos deste caso são os parâmetros e as funções, que veremos na próxima seção.

6 Variáveis Compostas Homogêneas

6.1 Vetores Unidimensionais

Até agora, vimos que uma variável está associada a uma posição de memória e qualquer referência a ela significa um acesso ao conteúdo de um pedaço de memória cujo tamanho depende de seu tipo. Nesta seção, iremos ver um dos tipos mais simples de estrutura de dados que nos possibilitará associar um identificador a um conjunto de elementos de um mesmo tipo. Naturalmente precisaremos de uma sintaxe apropriada para acessar cada elemento deste conjunto de forma precisa.

Antes de apresentar este tipo de estrutura de dados, considere o seguinte exemplo:

Exemplo 6.1 *Lêr 10 notas de alunos, calcular a média destas notas e imprimir as notas acima da média.*

Note que neste exemplo as notas devem ser lidas primeiro para depois se calcular a média das notas. Logo em seguida, cada nota deve ser comparada com a média, sendo que as maiores que a média são impressas.

Portanto, um programa para isso deveria conter pelo menos 10 variáveis apenas para guardar cada nota. O programa relativo a este exemplo se encontra na figura 18

```
program notas;
var nota1,nota2,nota3,nota4,nota5,
   nota6,nota7,nota8,nota9,nota10 : real;
   media                          : real;

begin
  write('Entre com a nota 1: '); readln(nota1);
  write('Entre com a nota 2: '); readln(nota2);
  write('Entre com a nota 3: '); readln(nota3);
  write('Entre com a nota 4: '); readln(nota4);
  write('Entre com a nota 5: '); readln(nota5);
  write('Entre com a nota 6: '); readln(nota6);
  write('Entre com a nota 7: '); readln(nota7);
  write('Entre com a nota 8: '); readln(nota8);
  write('Entre com a nota 9: '); readln(nota9);
  write('Entre com a nota 10: '); readln(nota10);
  media := (nota1+nota2+nota3+nota4+nota5+nota6+nota7+nota8+nota9+nota10)/10;
  writeln('A média das notas é ',media);
  if (nota1 > media) then writeln(nota1);
  if (nota2 > media) then writeln(nota2);
  if (nota3 > media) then writeln(nota3);
  if (nota4 > media) then writeln(nota4);
  if (nota5 > media) then writeln(nota5);
  if (nota6 > media) then writeln(nota6);
  if (nota7 > media) then writeln(nota7);
  if (nota8 > media) then writeln(nota8);
  if (nota9 > media) then writeln(nota9);
  if (nota10 > media) then writeln(nota10);
end.
```

Figura 18: Leitura de 10 notas, cálculo da média e impressão das maiores notas, usando variáveis simples.

Note que o programa da figura 18 está cheio de duplicações e cálculos idênticos para cada nota. Agora imagine este programa feito para uma turma de 100 alunos. Certamente não seria nada agradável ler, escrever ou programar desta maneira.

Variáveis compostas homogêneas correspondem a posições de memória, identificadas por um mesmo nome, individualizadas por índices e cujo conteúdo é de mesmo tipo.

Assim, o conjunto de 10 notas pode ser associado a apenas um identificador, digamos NOTA, que passará a identificar não apenas uma única posição de memória, mas 10. A referência ao conteúdo do n-ésimo elemento do conjunto será indicada pela notação NOTA$[n]$, onde n é um valor inteiro (também podendo ser uma expressão cujo resultado é inteiro).

85	100	50	40	70	80	95	65	75	90
1	2	3	4	5	6	7	8	9	10

Figura 19: Vetor de 10 notas representado pelo identificador NOTA.

Na figura 19, a nota de valor 70, que está na quinta posição da seqüência de notas é obtida como NOTA$[5]$.

A declaração de um vetor é feita usando a seguinte sintaxe:

Identificador : **array**[*Faixa_Escalar*] **of** Tipo_de_Cada_Elemento;

Onde *Faixa_Escalar* indica uma faixa de valores escalares. Naturalmente, se desejamos trabalhar com K elementos, então o vetor deve ser declarado com pelo menos K posições na faixa. Os elementos do vetor devem ser referenciados com valores de índice que pertencem a faixa de valores escalares usada na declaração do vetor.

Exemplo 6.2 *Exemplo de declaração de vetores:*

V : **array**$[1..1000]$ **of integer**;
VetReais : **array**$[-10..10]$ **of real**;
VetIndLetras : **array**$['a'..'z']$ **of real**;

A declaração do vetor NOTA, no exemplo acima, é feita da seguinte forma.

NOTA : **array**$[1..10]$ **of real**;

Na figura 20, apresentamos o programa alternativo ao da figura 19, agora usando vetor.

```
program notas2;
const quant=10;
var nota      : array[1..quant] of real;
  soma, media : real;
  i           : integer;
begin
  soma:=0;
  for i:=1 to quant do begin
    write('Entre com a nota ',i,': '); readln(nota[i]);
    soma := soma+nota[i];
  end;
  media := soma/quant;
  writeln('A média das notas é ',media);
  for i:=1 to quant do
    if (nota[i] > media) then writeln(nota[i]);
end.
```

Figura 20: Leitura de 10 notas, cálculo da média e impressão das maiores notas. Versão usando vetores.

Note que agora é fácil fazer uma mudança no programa da figura 20 para que o programa faça o mesmo processamento para 100 alunos em vez de 10. Basta mudar a quantidade de alunos definida na constante *quant* de 10 para 100. Outra alternativa é declarar o vetor com quantidade suficientemente grande de elementos, usar uma variável para armazenar a quantidade de alunos e ler pelo teclado esta quantidade.

Exemplo 6.3 *O seguinte programa lê um número positivo n e n valores reais em um vetor. O programa imprime os elementos do vetor que estão acima da média. Vamos projetar nosso programa por etapas, usando a seqüência dos seguintes passos:*

1. *Leia n, $n > 0$.*
2. *Leia os n valores reais no vetor.*
3. *Calcule a média M.*
4. *Imprima os valores maiores que M.*

```
program LeImprimeVetorReal;
const MAX       = 100;
type TipoVetorReal = array[1..MAX] of real;
var n,i       : integer;
  v        : TipoVetorReal;
  M,Soma : real;
begin
  { Passo 1: Leitura do valor n }
  write('Entre com a quantidade de elementos a ler: ');
  readln(n);
  while (n<1) or (n>MAX) do begin
    writeln('Quantidade fora dos limites possíveis [1,',MAX,'].');
    readln(n);
  end;
  { Passo 2: Leitura dos n valores reais no vetor }
  for i:=1 to n do begin
    write('Entre com o ',i,'-esimo elemento: ');
    readln(v[i]);
  end;
  { Passo 3: Cálculo da Média dos n valores lidos }
  Soma := 0;
  for i:=1 to n do Soma := Soma + v[i];
  M := Soma/n;
  { Passo 4: Impressão dos valores maiores que M }
  writeln('Valores maiores que a média ',M,':');
  for i:=1 to n do
    if (v[i]>M) then writeln(v[i]);
end.
```

Exemplo 6.4 *Faça um programa para imprimir uma seqüência de números lidos em ordem inversa.*

```
program Inverte;
const TamMax = 100;
var   i,n : Integer;
  v       : array [1..TamMax] of Integer;
begin
  Readln(n);  {0 <= n <= TamMax}
  for  i:=1  to  n  do  Read(v[i]);
  for  i:=n  downto  1  do  Writeln(v[i]);
end.
```

Exemplo 6.5 *O seguinte programa imprime o índice do maior elemento de um vetor lido. Neste exemplo, guardamos o índice onde aparece o maior elemento, inicialmente atribuído com 1. Em seguida, percorremos os demais elementos,* *atualizando o valor deste índice sempre que encontramos um elemento maior.*

```
program maximo;
const MAX      = 100;
type TipoVetorReal = array[1..MAX] of real;
var i, n, Ind : integer;
   V       : TipoVetorReal;
begin
  { Leitura dos elementos do vetor }
  write('Entre com a quantidade de elementos a ler: ');  readln(n);
  for i:=1 to n do begin
    write('Entre com o ',i,'-ésimo elemento: ');    readln(V[i]);
  end;
  { Busca da posição do maior elemento no vetor, guardando na variável Ind }
  if (n>0) then begin
    Ind := 1;
    for i:=2 to n do
      if V[Ind] < V[i] then Ind:=i; {Ao encontrar um valor maior, atualiza Ind}
  end;
  { Impressão do maior valor e seu índice. }
  if (n<=0) then writeln('Não há elementos no vetor.')
  else writeln('O maior valor do vetor é ',V[Ind]:7:2,' e aparece no índice ',ind);
end.
```

Exemplo 6.6 *O seguinte programa lê um vetor contendo n números reais imprime o desvio padrão, dp, dos n elementos. Obs.:* $dp = \sqrt{\frac{1}{n-1}\left(\sum_{i=1}^{n} x_i^2 - \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right)^2\right)}$

```
program ProgramaDesvioPadrao;
const MAX      = 100;
type TipoVetorReal = array[1..MAX] of real;
var i,n                          : integer;
  Soma,SomaQuadrado,DesvioPadrao : real;
  v                              : TipoVetorReal;
begin
  { Leitura do vetor }
  write('Entre com a quantidade de elementos a ler: ');  readln(n);
  for i:=1 to n do begin
    write('Entre com o ',i,'-ésimo elemento: ');    readln(V[i]);
  end;
  { Cálculo do desvio padrão }
  if (n<=1) then DesvioPadrao := 0
  else begin
    Soma := 0.0;    SomaQuadrado := 0.0;
    for i:=1 to n do begin
      Soma := Soma + v[i];
      SomaQuadrado := SomaQuadrado + Sqr(v[i]);
    end;
    DesvioPadrao := Sqrt( (1/(n-1)) * (SomaQuadrado-(1/n)*Sqr(Soma)));
  end;
  { Impressão do desvio padrão }
  writeln('O desvio padrão dos números lidos é: ',DesvioPadrao);
end.
```

Exemplo 6.7 *(Busca Seqüêncial) O seguinte programa lê o nome e a idade de vários alunos (no máximo 100). E em seguida o programa lê repetidamente um nome de aluno e imprime a idade deste aluno. O programa deve parar de ler* *quando for dado o nome 'fim', que não deve ter idade lida.*

```
program LeImprimeVetorReal;
const MAX         = 100;
type
  TipoNome         = string[100];
  TipoVetorIdade = array[1..MAX] of integer;
  TipoVetorNome  = array[1..MAX] of TipoNome;
var
  Vnome  : TipoVetorNome;
  Vidade : TipoVetorIdade;
  n,i        : integer;
  Nome        : TipoNome;
  Achou  : boolean;
begin
  { Leitura dos nomes e idades dos alunos }
  n:=0;
  writeln('Leitura dos nomes e idades dos alunos : ');
  write('Entre com o nome de um aluno: '); readln(nome);
  while (nome<>'fim') and (n<MAX) do begin
    n:=n+1;
    Vnome[n] := nome;
    write(' Entre com a idade do aluno ',nome,': ');    readln(Vidade[n]);
    write('Entre com o nome de um aluno: ');    readln(nome);
  end;
  { Leitura de nome e pesquisa (busca seqüêncial) da sua idade }
  writeln('Pesquisa das idades dos alunos : ');
  write('Entre com o nome de um aluno para pesquisar: '); readln(nome);
  while (nome<>'fim') do begin
    i:=1;
    Achou := false;
    while (not achou) and (i<=n) do
      if (Vnome[i]=nome) then Achou:=true
      else i:=i+1;
    if (Achou) then writeln('A idade do aluno ',nome,' é: ',Vidade[i])
    else writeln('O nome do aluno não foi encontrado');
    write('Entre com o nome de um para pesquisar aluno: ');    readln(nome);
  end;
end.
```

Exemplo 6.8 *A matriz abaixo representa o Triângulo de Pascal de ordem 6 (6 linhas).*

1					
1	1				
1	2	1			
1	3	3	1		
1	4	6	4	1	
1	5	10	10	5	1

Os elementos extremos de cada linha são iguais a 1. Os outros elementos são obtidos somando-se os dois elementos que aparecem imediatamente acima e à esquerda na linha anterior (i.e., 10=4+6). O seguinte programa lê um inteiro positivo n *e imprime as* n *linhas do triângulo de Pascal.*

```
program Pascal;
const TamMax = 100;
var n,i,j    : Integer;
  prev,corr : array[0..TamMax] of Integer;
begin
  Readln(n);   {0 < = n <= TamMax}
  prev[0] := 1;  corr[0] := 1;
  for i:=1 to n do prev[i] := 0;  {truque para evitar caso especial na diagonal}
  for i:=1 to n do begin
    for j:=1 to i do corr[j] :=  prev[j−1]+prev[j];
    for j:=0 to i do begin
      Writeln(corr[j]);
      prev[j] := corr[j];
    end
  end
end.
```

Obs.: Poderíamos usar "prev:=corr", mas seriam copiados todos os valores, mesmo os que não são usados.

Exemplo 6.9 *(Ordenação por Seleção) O seguinte programa lê um vetor de reais n números reais e o imprime em ordem não decrescente (menores primeiro). A estratégia usada é a mesma do algoritmo ordena3 apresentado no exemplo 7.4 na página 69. Neste caso, o algoritmo realiza $n-1$ iterações. Na primeira iteração, o programa encontra um maior elemento do vetor, e troca este elemento com o elemento que está na n-ésima posição. Na segunda iteração, o algoritmo encontra um segundo maior elemento (tirando o maior elemento, o segundo maior é um maior entre os demais) e troca com o elemento que está na $(n-1)$-ésima posição. Este processo se repete até que se tenha colocado o segundo elemento na posição correta. Naturalmente após estas $n-1$ iterações, um menor elemento deve estar necessariamente na primeira posição. Para encontrar um maior elemento, podemos usar a estratégia de busca apresentada no exemplo 6.5 (pg. 57).*

```
program SelectionSort;
const MAX        =  100;
type TipoVetorReal = array[1..MAX] of real;
var n,m,i,imax : integer;
  v             : TipoVetorReal;
  aux           : real;
begin
  { Passo 1: Leitura do valor n }
  write('Entre com a quantidade de elementos a ler: ');  readln(n);
  { Passo 2: Leitura dos n valores reais no vetor }
  for i:=1 to n do begin
    write('Entre com o ',i,'-esimo elemento: ');    readln(v[i]);
  end;
  { Passo 3: Ordenação do vetor }
  for m:=n downto 2 do begin
    { Encontra o m−ésimo maior elemento }
    imax := 1;
    for i:=2 to m do
      if (v[i] > v[imax]) then imax:=i;
    { Coloca o m−ésimo maior elemento na posição m, trocando valores de v[m] e v[imax] }
    aux := v[m];     v[m] := v[imax];     v[imax] := aux;
  end;
  { Passo 4: Impressão do vetor ordenado }
  for i:=1 to n do     writeln(v[i]:10:3);
end.
```

Uma das vantagens de se usar este método é que ele faz no máximo $n-1$ trocas entre elementos. Isto é vantajoso quando o tamanho de cada elemento (memória usada pelo elemento) é grande.

Exemplo 6.10 *(Intercalação de vetores) Faça um programa que lê dois vetores ordenados, $v1$ e $v2$, com $n1$ e $n2$ elementos, respectivamente e intercala os dois vetores gerando um terceiro vetor ordenado $v3$, com $n3 := n1 + n2$ elementos da intercalação de $v1$ e $v2$.*

```
program Intercala;
const TamMax = 100;
  TamMaxRes = 200;
var n1,n2,i1,i2,k : Integer;
  v1,v2         : array [1..TamMax] of Integer;
  w              : array [1..TamMaxRes] of Integer;
begin
  Readln(n1,n2);   {0< = n1,n2 <=TamMax; seqüências ordenadas}
  for i1:=1 to n1 do  Readln(v1[i1]);
  for i2:=1 to n2 do  Readln(v2[i2]);
  { Os dois vetores devem ser lidos ordenados }
  i1 := 1;  i2 := 1;  k := 0;
  while  (i1<=n1) and (i2<=n2)  do begin
    k := k+1;
    if  v1[i1]<=v2[i2] then  begin
      w[k] := v1[i1];
      i1 := i1+1
    end else begin
      w[k] := v2[i2];
      i2 := i2+1
    end
  end;
  for i1:=i1 to n1 do begin w[k] := v1[i1]; k := k+1 end;
  for i2:=i2 to n2 do begin w[k] := v2[i2]; k := k+1 end;
  for i1:=1 to k do Writeln(w[i1])
end.
```

Exemplo 6.11 *(Busca Binária) Note que no exemplo da busca seqüencial, descrita no exemplo 6.7 podemos ver que no pior caso, a rotina faz n comparações com elementos do vetor. Uma vez que temos um vetor já ordenado, é possível se fazer um algoritmo de busca mais eficiente.*

A estratégia é a mesma que usamos para encontrar um determinado nome em uma lista telefônica (uma lista telefônica de uma grande cidade pode ter milhões de assinantes). Neste caso, partimos a lista telefônica em uma posição, verificamos se um nome desta posição é menor, maior ou igual ao elemento que estamos procurando. Se for menor ou maior, podemos descartar uma das partes da lista.

Assim, a estratégia que podemos usar é a de sempre pegar um elemento do meio do vetor (ou um dos elementos do meio) e comparar, se este elemento é menor, maior ou igual ao elemento que estamos procurando. Se for menor ou maior, podemos restringir a busca em apenas uma das partes do vetor.

```
program ProgBuscaBinaria;
const MAX      = 100;
type  TipoVetorReal = array[1..MAX] of real;
var
  v                    : TipoVetorReal;
  n,pos,i,esq,dir,meio : integer;
  x                    : real;
begin
  writeln('Entre com o número de elementos a inserir: '); readln(n);
  { A seqüência lida deve estar ordenada. }
  for i:=1 to n do begin write('Entre com o ',i,'-ésimo elemento: '); readln(v[i]); end;
  { Pesquisa de um elemento }
  write('Entre com um elemento a pesquisar: ');  readln(x);
  pos := 0; { Uma vez que o elemento tenha sido encontrado, pos terá o índice do elemento }
  esq:=1;
  dir :=n;
  while (esq<=dir) and (pos=0) do begin
    meio := (esq+dir) div 2;
    if (x<v[meio]) then dir := meio-1 { descarta parte da direita }
    else if (x>v[meio]) then esq := meio+1 {descarta parte da esquerda }
    else pos:=meio; { encontrou o elemento }
  end;
  if pos>0 then Writeln('O elemento ',x,' está na posição ',pos)
  else Writeln('Elemento não encontrado.');
end.
```

A busca binária nos introduz a uma questão interessante. Quando usar a busca binária e quando usar a busca seqüêncial ?

Considere primeiro o algoritmo que usa busca seqüêncial. Este algoritmo realiza no pior caso, no máximo n iterações por busca. Assim, se necessitamos de m buscas, faremos em média uma quantidade de operações proporcional a $n \cdot m$ operações.

Para o caso da busca binária, note que é necessário se fazer inicialmente a ordenação do vetor. Vamos supor que usamos a estratégia do algoritmo *SelectionSort* para ordenar o vetor. Fazendo uma análise em alto nível, podemos ver que a rotina SelectionSort faz $n-1$ iterações, mas em cada uma delas há uma chamada da rotina *IndMaximo* que faz no máximo $n-1$ iterações. Assim, a grosso modo, a quantidade de operações total para ordenar não é maior que uma proporção de n^2. Agora vamos analisar, também em alto nível, a busca binária no pior caso, que faz digamos k iterações. Na primeira iteração são considerados os n elementos. Na segunda iteração são considerados $n/2$ elementos. Na terceira iteração são considerados $n/4$ elementos, assim, por diante. Com isso a quantidade de operações realizadas não é maior que uma proporção de $\log_2(n)$. Considerando o tempo de ordenação mais o tempo de se fazer m buscas, temos um tempo que é delimitado por uma proporção de $n^2 + m \cdot \log_2(n)$ operações.

Assim, podemos dizer que os tempos de processamento total destas duas estratégias são próximos dos comportamentos de duas funções: $s(n,m) := c_s \cdot (n \cdot m)$ para a busca seqüêncial e a função $b(n,m) := c_b \cdot (n^2 + m \cdot \log_2(n))$ para a busca binária (c_s e c_b são constantes adequadas). Note que para valores pequenos de m, a busca seqüêncial faz muito menos processamento que a estratégia que usa busca binária. Por outro lado, para valores grandes de m (e.g., valores bem maiores que n), temos que a busca binária faz menos processamento que a busca seqüêncial. Na seção 11 iremos estudar métodos mais eficientes para ordenação, o que faz com que métodos de busca usando a estratégia da busca binária fiquem bem atraentes.

6.2 Vetores Multidimensionais

A linguagem Pascal também permite a declaração de vetores de vetores. I.e., é um vetor onde cada elemento também é um vetor. Além disso, você pode declarar vetores de vetores de vetores ...

Vetores de vetores são comumente chamadas de matrizes, e correspondem a forma de matrizes que costumamos aprender no ensino básico. Matrizes podem ser declaradas como nas seguintes formas equivalentes:

mat : **array**$[I1..F1]$ **of array**$[I2..F2]$ **of** Tipo_de_Cada_Elemento;

ou

mat : **array**$[I1..F1, I2..F2]$ **of** Tipo_de_Cada_Elemento;

Na figura 21 apresentamos uma representação gráfica da matriz, conforme a declaração acima. O elemento X pode ser acessado usando se a sintaxe $Mat[i, j]$.

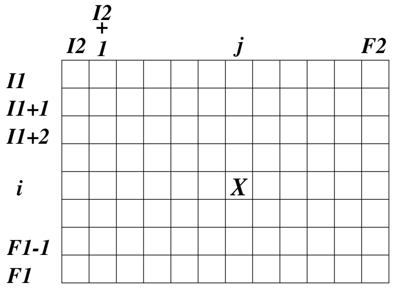


Figura 21: Mat : **array**$[I1..F1, I2..F2]$ **of** Tipo_de_Cada_Elemento

Exemplo 6.12 *O seguinte programa faz a leitura e a impressão de uma matriz de forma tabular. Para cada uma destas operações, o programa percorre todos os elementos a serem lidos/impressos com dois comandos* **for** *encadeados.*

```
program matriz;
const MaximaDimensao = 100;
type TipoMatrizReal = array[1..MaximaDimensao,1..MaximaDimensao] of real;
  TipoMsg        = string[100];
var n,m,i,j : integer;
  mat        : TipoMatrizReal;
begin
  { Leitura das dimensões da matriz }
  write('Entre com o número de linhas da matriz: ');  readln(n);
  write('Entre com o número de colunas da matriz: ');  readln(m);
  { Leitura da matriz }
  writeln('Leitura dos elementos da matriz');
  for i:=1 to n do
    for j:=1 to m do begin
      write('Entre com o elemento (',i:3,',',j:3,'): ');  readln(mat[i,j]);
    end;
  { Impressão da matriz }
  for i:=1 to n do begin
    for j:=1 to m do      write(mat[i,j]:5:1,' ');
    writeln;
  end;
end.
```

Exemplo 6.13 *O seguinte programa lê três inteiros, p, q e r; lê duas matrizes A e B, de dimensões $p \times q$ e $q \times r$, respectivamente; e imprime a multiplicação da matriz A com a matriz B. Obs.: Vamos supor que temos descrito o código para ler e imprimir as matrizes, como descritos no exemplo 6.12.*

```
program multmat;
const MAX = 10;
type
  tipomatrizreal = array[1..MAX,1..MAX] of real;
var
  Soma,A,B,C      : TipoMatrizReal;
  p,q,r,i,j,k : integer;

begin
  { Passo 1: Fazer leitura das dimensões p,q,r }
  { Passo 2: Fazer leitura das Matrizes A e B }
  { Passo 3: Multiplicação das matrizes A e B em C }
  for i:=1 to p do
    for j:=1 to r do begin
      Soma := 0;
      for k:=1 to q do
        Soma := Soma + A[i,k]*B[k,j];
      C[i,j] := Soma;
    end;
  { Passo 4: Fazer impressão da matriz C }
end.
```

6.3 Exercícios

1. Faça um programa que lê as notas de n alunos, cada nota é um inteiro entre 0 e 100, e imprima a quantidade de vezes com que apareceu cada nota.

2. Faça um programa para ordenar um vetor de números reais de forma não crescente (i.e., os maiores primeiro).

3. Faça um programa lê o nome, idade e salário de n pessoas (n lido). O programa deve ordenar e imprimir os dados destas pessoas ordenados pelo nome, pelo salário e pela idade (de maneira não decrescente).

4. Faça um programa que lê uma matriz 3×3 e imprime o determinante da matriz.

5. Escreva um programa que imprime um Triângulo de Pascal de ordem n (n lido). O programa deve usar apenas um vetor e apenas um comando de repetição **for**.

6. Uma matriz quadrada inteira é chamada de *quadrado mágico* se a soma dos elementos de cada linha, a soma dos elementos de cada coluna e as soma dos elementos das diagonais principal e secundária são todos iguais. Exemplo: As matrizes abaixo são quadrados mágicos:

$$\begin{bmatrix} 3 & 4 & 8 \\ 10 & 5 & 0 \\ 2 & 6 & 7 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

   Faça um programa que verifica se uma dada matriz quadrada lida é ou não um quadrado mágico.

7. Um número é dito ser palíndrome se a seqüência dos dígitos do número lidos da esquerda para a direita é igual a seqüência de dígitos lidos da direita para a esquerda. Por exemplo: Os seguintes números são palíndromes: 123454321, 54445, 789987, 121.
   Faça duas versões de um programa que verifica se um número <u>inteiro</u> é ou não um número palíndrome, uma usando vetor e outra sem usar vetor.

8. Leia dois vetores v_1 e v_2, com n_1 e n_2 elementos, respectivamente, onde cada elemento do vetor é um dígito. Faça um programa que realiza a multiplicação dos números representados por estes dois vetores colocando o resultado em um terceiro vetor. Obs.: Cada vetor de entrada tem tamanho máximo de 100 digitos.

9. Um mapa apresenta n cidades definidas pelos números $1, 2, \ldots, n$. Cada duas cidades i e j podem estar ligadas por uma estrada. Chamamos de componente um grupo maximal de cidades que estão ligadas por estradas (i.e., é possível ir de uma cidade para outra que esteja no mesmo grupo andando por estradas). Faça um programa que lê um valor n, quantidade de cidades, um valor m, quantidade de estradas, e m pares $(i_1, j_1), (i_2, j_2), \ldots, (i_m, j_m)$. Cada par (i, j) indica que há uma estrada ligando as cidades i e j. O programa deve imprimir k linhas, onde k é a quantidade de componentes no mapa. Cada linha contém uma lista das cidades que estão em uma mesma componente.

7 Procedimentos e Funções

A linguagem pascal permite que possamos declarar trechos de código fora do programa principal e associados a um identificador que podem ser executadas sempre que invocados. Chamaremos estes trechos de código por *módulos* ou *rotinas*.

Os principais motivos para se usar rotinas são:

1. Evitar codificação: trocar certos trechos de programas que se repetem por chamadas de apenas uma rotina que será codificada apenas uma vez.
2. Modularizar o programa, dividindo-o em módulos (rotinas) lógicamente coerentes, cada uma com função bem definida. Isto facilita a organização do programa, bem como o entendimento dele.
3. Facilitar a busca e correção de possíveis erros do programa, uma vez que podemos isolá-los mais facilmente.

Na linguagem pascal, podemos declarar dois tipos de rotinas, que são os procedimentos e as funções.

7.1 Procedimentos

Um procedimento é sempre declarado em uma área de declarações, podendo ser tanto na área de declarações do programa principal como nas áreas de declarações de outras rotinas. Neste último caso, a *visibilidade* deste procedimento segue as regras de escopo (veja seção 7.4).

A forma geral de um procedimento segue o seguinte formato:

```
procedure Identificador_Procedimento(Lista_de_Parâmetros);
    Declarações   {Área de Declarações}
begin
    Comandos   {Corpo de Execução do Procedimento}
end;
```

O procedimento tem o mesmo formato do programa principal, sendo que o corpo do procedimento está delimitado por **begin** e **end**, este último terminado com um ponto e vírgula (Na rotina principal, o **end** é terminado com um ponto final). Além disso, a lista de parâmetros é opcional.

Na área de declarações, é possível definir constantes, tipos, declarar variáveis, funções e outros procedimentos, da mesma forma como declarados na área de declarações da rotina principal. É importante observar que os objetos declarados na área de declarações de uma rotina, são válidos apenas no escopo desta rotina, e não são válidos fora dela.

Exemplo 7.1 *Um determinado programa imprime o seguinte cabeçalho diversas vezes no meio de sua execução:*

```
Aluno: Fulano de Tal
Data: 01/01/01
Programa: Exemplo de Procedimento
```

Desta maneira, é melhor construir um procedimento que escreve este cabeçalho sempre que necessário. O pseudo-programa seguinte declara este procedimento e faz algumas chamadas no meio de sua execução.

```
program Exemplo;

procedure Cabecalho;
begin
    writeln('Aluno: Fulano de Tal.');
    writeln('Data: 01/01/01.');
    writeln('Programa: Exemplo de Procedimento.');
end;

begin
    ⋮
    Cabecalho;   {(*)}
    ⋮
    Cabecalho;   {(*)}
    ⋮
    Cabecalho;   {(*)}
    ⋮
end.
```

O cabeçalho será impresso nas chamadas da rotina Cabecalho, linhas comentadas com (*).

7.2 Passagem de Parâmetros

Como estratégia de programação, é desejável que cada rotina não faça uso de dados externos, pelo menos de forma direta, e toda a comunicação com o resto do programa seja feito através de parâmetros ou se for o caso, como retorno de função.

No programa seguinte, descrevemos um procedimento chamado *ImprimeMaximoDeTres* com três parâmetros reais: *x, y* e *z*. Os parâmetros declarados neste procedimento são declarados com *passagem de valor*. Isto significa que o procedimento ImprimeMaximoDeTres é chamado com três parâmetros:

ImprimeMaximoDeTres(<u>Expressão 1</u>, <u>Expressão 2</u>, <u>Expressão 3</u>);

onde <u>Expressão 1</u>, <u>Expressão 2</u> e <u>Expressão 3</u> são expressões numéricas que serão primeiro avaliadas para valores numéricos e só então transferidas para o procedimento. Internamente ao procedimento estes valores são acessados pelos parâmetros x, y e z.

```
program Maximos;
var a,b,c           : Real;
  i,n               : Integer;
procedure ImprimeMaximoDeTres(x,y,z : Real);
var t : Real;
begin
  t := x;
  if t<y then t := y;
  if t<z then t := z;
  writeln('O maior entre ',x,',',y,',',z,' é ',t);
end; {MaximoDeTres}
begin
  Write('Entre com três números: ');
  Readln(a,b,c);
  ImprimeMaximoDeTres(a,b,c);
end.
```

Os parâmetros são declarados da seguinte forma:

$$\begin{array}{ccc}
[\textbf{var}] & \text{ListaDeParâmetrosDoTipo1} & \text{Tipo1;} \\
[\textbf{var}] & \text{ListaDeParâmetrosDoTipo2} & \text{Tipo2;} \\
\vdots & \vdots & \vdots \\
[\textbf{var}] & \text{ListaDeParâmetrosDoTipoK} & \text{TipoK;}
\end{array}$$

onde [**var**] indica que a palavra **var** pode ou não ser colocada antes da lista de parâmetros de um tipo.

As palavras *Tipo1, Tipo2,..., TipoK* são palavras que identificam o tipo do parâmetro. Assim o seguinte cabeçalho de procedimento é inválido.

```
procedure Imprime(msg : string[50]);
```

Uma construção correta seria primeiro definir um tipo associado a **string[50]** e só então usar este tipo na declaração de *msg*. I.e.,

```
type TipoMSG = string[50];
procedure Imprime(msg : TipoMSG);
```

A inserção da palavra **var** antes da declaração de uma lista de parâmetros indica que estes parâmetros são declarados com Passagem por Referência e a ausência da palavra **var** indica que a lista de parâmetros seguinte é feita com Passagem por Valor.

A seguir, descrevemos os dois tipos de passagem de parâmetros.

Passagem por Valor Nesta forma, a expressão correspondênte ao parâmetro é avaliada e *apenas seu valor* é passado para o variável correspondente ao parâmetro dentro da rotina.

Passagem por Referência Nesta forma, o parâmetro que vai ser passado na chamada da rotina deve ser necessariamente uma va Isto porque não é o valor da variável que é passada no parâmetro, mas sim a sua referência. Qualquer alteração de valor no parâmetro correspondente refletirá em mudanças na variável correspondente, externa ao procedimento.

Exemplo 7.2 *No exemplo da figura 22 apresentamos um programa com uma rotina chamada Conta, contendo dois parâmetros, um por valor e outro por referência. Na linha (7), o comando de escrita imprime os valores de* $a = 20$

```
( 1)  Program exemplo;
( 2)  var x: integer;  y:real;
( 3)  Procedure Conta(a: integer; var b: real);
( 4)  begin
( 5)       a := 2 * a;
( 6)       b := 3 * b;
( 7)       writeln('O valor de a = ',a,' e o valor de b = ',b);
( 8)  end;
( 9)  begin
(10)       x := 10;
(11)       y := 30.0;
(12)       Conta(x, y);
(13)       writeln('O valor de x = ',x,' e o valor de y = ',y);
(14)  end.
```

Figura 22: Parâmetros por valor (a: integer) e por referência (var b: real).

e $b = 90.0$. *Na linha (13), depois da chamada da rotina Conta, são impressos os valores de* $x = 10$ *e* $y = 90.0$, *já que a passagem do parâmetro correspondente a* x *é por valor e a passagem do parâmetro correspondente a* y *é por referência (*x *tem o mesmo valor antes da chamada, e* y *tem o valor atualizado pela rotina Conta).*

Exemplo 7.3 *A seguir, apresentamos alguns exemplos do cabeçalho correspondente a declaração de rotinas com parâmetros:*

- *Procedure Conta(i, j:integer;var x, y:integer;var a, b:real);*
  Neste exemplo, os parâmetros correspondentes a i e j são declarações de parâmetros com passagem por valor; e x, y, a e b são declarações de parâmetros com passagem por referência.
- **type** *MeuTipoString = string[50];*
  Procedure processa_string(var nome1:MeuTipoString; nome2:MeuTipoString;var nome3:MeuTipoString; var a:integer; b:real);
  Neste exemplo, os parâmetros correspondentes a nome1, nome3 e a são declarações de parâmetros com passagem por referência; e nome2 e b são declarações de parâmetros com passagem por valor.

Exemplo 7.4 *O programa para ordenar três números, descrito na página 42, usa a estratégia de trocar os valores de variáveis. Note que naquele programa, a troca de valores é feita em três pontos do programa. Assim, nada melhor que fazer um procedimento para isso. O seguinte programa descreve esta alteração no programa ordena3. Note que os dois parâmetros da rotina trocaReal devem ser necessariamente declarados como parâmetros passados por referência.*

```
program ordena3;

procedure TrocaReal(var A, B : real);  {Troca os valores das duas variáveis}
var AUX: real;
begin
    AUX := A;
    A := B;
    B := AUX;
end;

var n1, n2, n3: real;
begin
    write('Entre com o primeiro número: ');
    readln(n1);
    write('Entre com o segundo número: ');
    readln(n2);
    write('Entre com o terceiro número: ');
    readln(n3);
    if (n1 > n2) then TrocaReal(n1, n2);
    if (n1 > n3) then TrocaReal(n1, n3);
    {Neste ponto, n1 contém o menor dos três valores}
    if (n2 > n3) then TrocaReal(n2, n3);
    { Neste ponto, n1 contém o menor dos três valores e n2 é menor ou igual a n3.}
    writeln(n1, n2, n3);
end.
```

Note que este programa ficou mais enxuto e mais fácil de entender.

Exemplo 7.5 *Um certo programa precisa ter, em vários pontos do seu código, leituras de números inteiros positivos e em cada um destes lugares, é necessário se fazer a validação do número lido. Uma maneira de se validar o número a ser lido é usar uma estrutura de repetição, como no exemplo 4.5 da página 34, em um procedimento que lê uma variável inteira e já faz sua validação. Desta maneira não precisaremos repetir o código para cada leitura.*

```
program ProgramaValidacao;
type messagetype = string[50];
procedure LeInteiroPositivo(var m : integer; msg:messagetype);
begin
  write(msg);
  readln(m);
  while (m<=0) do begin
    writeln('ERRO: Número inválido.');
    write(msg);
    readln(m);
  end;
end;

var n1,n2,n3:integer;
begin
  LeInteiroPositivo(n1,'Entre com o primeiro número inteiro positivo: ');
  LeInteiroPositivo(n2,'Entre com o segundo  número inteiro positivo: ');
  LeInteiroPositivo(n3,'Entre com o terceiro número inteiro positivo: ');
  writeln('Os três números positivos foram: ',n1,',',n2,',',n3);
end.
```

Consideração sobre a passagem de estruturas grandes como parâmetros

Muitas vezes, quando temos passagem de estruturas grandes (como vetores e matrizes) como parâmetros por valor, é preferível recodificar a rotina para que esta seja feita como parâmetro por referência. O motivo disto é justamente o fato destas estruturas serem duplicadas na passagem por valor. Na passagem de parâmetros por referência apenas a referência do objeto é transferida, gastando uma quantidade de memória constante para isso. Naturalmente esta codificação não deve mudar os valores da estrutura, caso contrário estas se manterão após o término da rotina.

Exemplo 7.6 *Considere os dois procedimentos seguintes, nas figuras 23 e 24, para imprimir o maior valor em um vetor V de n elementos. Note que nenhuma das duas rotinas faz alterações no vetor. A única diferença destas duas implementações é a passagem do parâmetro V por valor (figura 23) e por referência (figura 24). Assim, a implementação da figura 24 é mais eficiente que a da figura 23, caso o compilador em uso não faça nenhuma otimização para mudar o tipo da passagem de parâmetro.*

```
type TipoVetorReal = array[1..1000] of real;
procedure ImpMaximo(V : TipoVetorReal;
              n : integer);
var M : real; i:integer;
begin
  if (n>0) then begin
    M := V[1];
    for i:=2 to n do
      if (V[i]>M) then M:=V[i];
    writeln('O máximo é ',M);
  end
end.
```

Figura 23:

```
type TipoVetorReal = array[1..1000] of real;
procedure ImpMaximo(var V : TipoVetorReal;
                  n : integer);
var M : real; i:integer;
begin
  if (n>0) then begin
    M := V[1];
    for i:=2 to n do
      if (V[i]>M) then M:=V[i];
    writeln('O máximo é ',M);
  end
end.
```

Figura 24:

7.3 Funções

A linguagem *Pascal* também permite desenvolver novas funções além das já existentes. Funções são rotinas parecidas com procedimentos, com a diferença que funções retornam um valor. Uma função é declarada com o seguinte cabeçalho:

function Identificador(Lista de Parâmetros): Tipo_da_Função;

O valor a ser retornado pela função é calculado dentro do *corpo* da função e para retorná-lo, é usado um identificador com o mesmo nome da função. O tipo de valor retornado é do tipo *Tipo_da_Função* que deve ser uma palavra identificadora do tipo. Assim, uma função não deve ser declarada como no exemplo a seguir

function Maiuscula(str: MeuTipoString):string[50];

Neste caso, **string[50]** não é apenas uma palavra. Assim, uma possível declaração seria:

function Maiuscula(str: MeuTipoString):MeuTipoString;

Exemplo 7.7 *No programa a seguir, temos a declaração de uma função chamada* **Cubo**, *que dado um valor real (como parâmetro), a função devolve o cubo deste valor.*

```
program exemplo;
var a: real;
function Cubo(x:real): real;
begin
   Cubo:= x * x * x;
end;
begin
   write('Entre com o valor de a: ');
   readln(a);
   write('O cubo de ',a,' é ',Cubo(a));
end.
```

Exemplo 7.8 *No programa a seguir, apresentamos um programa com a declaração de uma função que devolve o maior valor entre dois valores, dados como parâmetros.*

```
program exemplo;
var a, b, c: real;
function Maximo(x,y: real): real;
begin
   if (x > y)
     then
       Maximo := x
     else
       Maximo := y;
end;
begin
   write('Entre com o valor de a: ');
   readln(a);
   write('Entre com o valor de b: ');
   readln(b);
   c := Maximo(a, b);
   write('O máximo entre ',a,' e ',b,' é: ',c);
end.
```

Exemplo 7.9 *Faça um programa contendo uma função que calcula o fatorial de um número passado como parâmetro inteiro.*

```
program ProgramaFatorial;
function fatorial(n:integer):integer;
var F,i:integer;
begin
  F := 1;
  for i:=2 to n do  F:=F * i;
  fatorial := F;
end;
var n:integer;
begin
  write('Entre com um número: ');  readln(n);
  writeln('O fatorial de ',n,' é igual a ',fatorial(n));
end.
```

Exemplo 7.10 *Implemente o algoritmo de Euclides para calcular o máximo divisor comum de dois números.*

```
program ProgramaMDC;
function mdc(a,b : integer):integer;
var aux,maior,menor : integer;
begin
  maior := a;  menor := b;
  while (menor <> 0) do begin
    aux := menor;
    menor := maior mod menor;
    maior := aux;
  end;
  mdc := maior;
end;
var a,b:integer;
begin
  write('Entre com dois numeros positivos: '); readln(a,b);
  writeln('O mdc dos dois é ',mdc(a,b));
end.
```

Exemplo 7.11 *Descreva uma função para testar se um número é primo usando a estratégia do programa Primo2 apresentada na página 50.*

```
function Primo(n :integer ):boolean;
var i,Max    : integer;
begin
  if (n = 2) then Primo := true
  else if (n mod 2 = 0) or (n<=1) then Primo:= False
  else begin
    Primo := True;
    i:=3;
    Max := trunc(sqrt(n));
    while (i<= Max) and (n mod i <>0) do i := i+2;
    if (i<=Max) and (n mod i = 0) then Primo := false
  end
end
```

7.4 Escopo

Todos os objetos declarados em um programa (subprogramas, variáveis, constantes, tipos, etc) possuem um *escopo* de atuação. Por escopo de um objeto, entendemos como as regiões de um programa onde o objeto é válido.

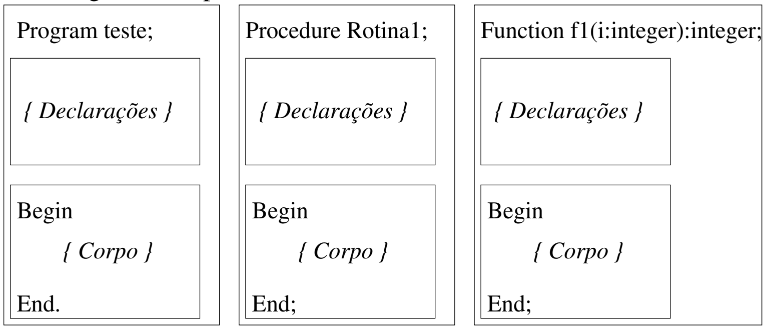
Primeiramente, vamos definir alguns termos:

Cabeçalho de uma Rotina É o texto que indica o tipo da rotina (se programa principal, procedimento ou função), seu nome e seus parâmetros.

Área de Declaração da Rotina Área onde se declaram as variáveis, tipos, procedimentos e funções da rotina.

Corpo de uma Rotina Definiremos o Corpo de uma Rotina (pode ser o Programa Principal, um Procedimento ou uma Função) como o trecho da rotina contendo as instruções a serem executadas.

Na figura a seguir, temos alguns exemplos de rotinas.



Note também que na área de declaração de uma rotina podemos declarar novas rotinas. Na figura 25, temos uma estrutura de um programa com várias rotinas, declaradas uma dentro da área de declarações da outra.

Se em algum lugar é feito alguma referência a um objeto do programa, este já deve ter sido declarado em alguma posição acima desta referência.

Os objetos declarados em uma rotina **R** são visíveis no corpo de **R** e em todas as subrotinas dentro da área de declarações de **R**. Por visualizar, queremos dizer que podemos usar a variável/procedimento/função no local de visualização.

Se dois objetos têm o mesmo nome, a referência através deste nome é feita para o objeto que estiver na área de declarações visível mais interna. Exemplo: considere duas rotinas, **R** e **R1**, onde **R1** está declarado na área de declarações de **R** (**R1** está contido em **R**). Um identificador declarado dentro de **R1** pode ser declarado com o mesmo nome de um outro identificador em **R** (externo a R1). Neste caso, a variável declarada em **R** não será visualizada em R1.

Na figura 25 apresentamos o esboço de um programa com diversos níveis de encaixamento de rotinas. A seguir, descrevemos quais objetos podem ser visualizados em cada região:

1. Na região 1, podemos *visualizar* os objetos: **C, B, A** (primeiro A), **Rotina1** e **Rotina2**.
2. Na região 2, podemos *visualizar* os objetos: **D, B, A** (segundo A), **Função1, Rotina1** e **Rotina2**.
3. Na região 3, podemos *visualizar* os objetos: **A** (primeiro A), **B, Função1, Rotina1** e **Rotina2**.
4. Na região 4, podemos *visualizar* os objetos: **E, A** (primeiro A) e **Rotina3** e **Rotina1**.
5. Na região 5, podemos *visualizar* os objetos: **A** (primeiro A) e **Rotina1** e **Rotina3**.

Exemplo 7.12 *O programa seguinte apresenta um procedimento que encontra as raizes reais de uma equação do segundo grau. O procedimento se chama Equacao, tem três parâmetros e chama uma função particular chamado CalculaDelta. Por ser particular a este procedimento, a função CalculaDelta pode estar dentro da área de declarações*

```
Program escopo;
        var A: integer;    {Primeiro A}
        Procedure rotina1;
            var B: integer;
            Procedure rotina2;
            var C: integer;
            begin
                {Região 1}
             end;
            Function funcao1(x:integer): real;
            var A: integer;   {Segundo A}
                 D: real;
            begin
                {Região 2}
            end;
        begin
              {Região 3}
        end;
        Procedure rotina3;
            var E: integer;
        begin
                {Região 4}
        end.
begin
        {Região 5}
end.
```

Figura 25: Exemplo de objetos com escopos diferentes.

de Equacao.

```
program EquacaoSegundoGrau;
var a_ext, b_ext, c_ext : real;
procedure Equacao(a,b,c: real);  { imprime as soluções reais de a*x*x + b*x+c = 0 }
var x1, x2,delta              : real;  { Variáveis locais ao procedimento.}
function CalculaDelta(c1,c2,c3: real): real;
begin
  CalculaDelta := sqr(c2) − 4*c1*c3;
end;
begin
  delta:=CalculaDelta(a,b,c);
  if (delta>=0)  then  begin
    x1:=(−b+sqrt(delta)) / (2*a);      x2:=(−b−sqrt(delta)) / (2*a);
    writeln('O valor de x1 = ',x1,' e o valor de x2 = ',x2);
  end else writeln('não é possivel calcular raizes reais para esta equação');
end;
begin
  writeln('Encontrar raizes reais de equação na forma: a*x*x + b*x+c = 0');
  write('Entre com o valor de a (diferente de zero), b e c: '); readln(a_ext,b_ext,c_ext);
  Equacao(a_ext,b_ext,c_ext);
end.
```

Exemplo 7.13 *O seguinte programa apresenta um procedimento para ordenação usando a estratégia do algoritmo SelectionSort, descrita com duas subrotinas encaixadas.*

```
program SelectionSort;
const MAX       = 100;
type TipoVetorReal = array[1..MAX] of real;
  TipoMsg        = string[100];
procedure LeVetorReal(var v : TipoVetorReal; n:integer;msg:TipoMsg);
var i : integer;
begin
  writeln(msg);
  for i:=1 to n do begin writeln('Entre com o ',i,'-ésimo elemento: '); readln(V[i]); end
end; { LeVetorReal }
procedure ImprimeVetorReal(var v : TipoVetorReal; n:integer;msg:TipoMsg);
var i : integer;
begin
  writeln(msg);  for i:=1 to n do begin writeln(V[i]); end
end; { ImprimeVetorReal }
procedure SelectionSort(var v : TipoVetorReal; n:integer);
var m,imax : integer;
  procedure TrocaReal(var a,b : real);
  var aux : real;
  begin   aux := a;    a:=b;    b:=aux;  end;
  function IndMaximo(var v : TipoVetorReal; n : integer) : integer;
  var i, Ind : integer;
  begin
    if (n <=0) then Ind := 0
    else begin
      Ind := 1; {O maior elemento começa com o primeiro elemento do vetor}
      for i:=2 to n do if v[Ind] < v[i] then Ind:=i;
    end;
    IndMaximo := Ind;
  end;
begin
  for m:=n downto 2 do begin
    imax := IndMaximo(v,m);
    TrocaReal(v[m],v[imax]);
  end;
end; { SelectionSort }
var i, n, Ind : integer;
   V       : TipoVetorReal;
begin
  write('Entre com a quantidade de elementos a ler: '); readln(n);
  LeVetorReal(v,n,'Leitura do vetor a ordenar');
  SelectionSort(v,n);
  ImprimeVetorReal(v,n,'Vetor ordenado');
end.
```

7.5 Cuidados na Modularização de Programas

A modularização dos programas pode levar ao desenvolvimento de programas mais indepententes e mais fáceis de se entender. Mas há casos onde devemos tomar algum cuidado, se quisermos evitar processamentos desnecessários. Dois casos onde podemos ter processamento desnecessário são:

1. Rotinas distintas fazendo os mesmos cálculos, i.e., alguns dados vão ser recalculados.
2. Chamadas de uma mesma rotina várias vezes, onde uma chamada pode ter feito cálculos já realizados pela chamada anterior.

Muitas vezes priorizamos a independência e funcionalidade das rotinas e toleramos que algumas computações sejam duplicados. Isto é razoável quando estas duplicações não são críticas no tempo de processamento total. Quando o tempo de processamento deve ser priorizado, devemos reconsiderar o programa e evitar estas computações desnecessárias. O seguinte exemplo mostra duas versões de um programa para calcular a exponencial (e^x).

Exemplo 7.14 *A exponencial* (e^x) *pode ser calculada pela seguinte série:*

$$e^x = x^0 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

O programa seguinte mostra uma implementação da função exponencial, usando duas outras funções, uma para calcular o fatorial, e outra para calcular a potência. O programa pára quando a diferença entre o valor da série calculado até uma iteração com o valor calculado na iteração anterior é menor que 0,00001.

```
program exprog1;
var   y : real;
function fatorial(n : integer):integer;
var i,f      : integer;
begin
  i:=1;  f:=1;
  while (i<=n) do begin     f:=f*i;  i:=i+1;  end;
  fatorial:=f;
end;
function potencia(x : real; n:integer):real;
var i : integer;
   p : real;
begin
  p:=1;
  for i:=1 to n do p:=p*x;
  potencia := p;
end;
function expo1(x : real):real;
var termo,ex,exant: real;  i:integer;
begin
  ex := 0;  termo:=1;  i := 0;
  repeat
    exant := ex;
    termo := potencia(x,i)/fatorial(i);
    ex := ex + termo;
    inc(i);
  until abs(ex−exant)<0.0001;
  expo1 := ex;
end;
begin
  write('Entre com um número real: ');  readln(y);
  writeln('O valor de e elevado a  ',y,' é igual a ',expo1(y));
end.
```

O programa exprog1 é fácil de entender, uma vez que cada termo da série é facilmente calculado atráves da função fatorial e da função potência. Mas note que ao se calcular um termo genérico desta série, em uma iteração, digamos o termo $\frac{x^k}{k!}$, o programa calcula o valor x^k, sendo que em iterações anteriores, já foram computados os valores de x^1, x^2,..., x^{k-1}. Cada um destes cálculos poderia ter aproveitado o cálculo da potência feita na iteração anterior. O mesmo acontece com o fatorial, $k!$, que poderia ter aproveitado o cálculo de $(k-1)!$ feito na iteração anterior. Portanto, o termo calculado em uma iteração é igual ao termo anterior, $\frac{x^{k-1}}{(k-1)!}$, multiplicado por $\frac{x}{k}$. O seguinte programa apresenta a versão modificada, sem repetição de processamento.

```
program exprog2;
var    y : real;
function expo2(x : real):real;
var termo,ex,exant : real;   i:integer;
begin
  termo:=1;   i := 1;   ex := 0;
  repeat
    exant := ex;
    ex := ex + termo;
    termo := termo * x/i;
    inc(i);
  until abs(ex−exant)<0.0001;
  expo2 := ex;
end;
begin
  write('Entre com um número real: ');
  readln(y);
  writeln('O valor de e elevado a  ',y,' é igual a ',expo2(y));
end.
```

Exercício 7.1 *O valor do seno(x) pode ser dado pela seguinte série:*

$$\text{seno}(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots$$

Implemente uma função chamada seno(x), para calcular o valor do seno de x. Note que x^{k+2} é igual a $x^k * x^2$ e $(n+2)!$ é igual a $n! \cdot (n+1) \cdot (n+2)$.
OBS.: *A função deve usar no máximo um loop. I.e, não se pode usar loops encaixados.*

Exercício 7.2 *O valor do co-seno de x pode ser calculado pela serie*

$$co\text{−}seno(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \ldots$$

Descreva uma função com o cabeçalho: *function coseno(x:real):real;*
e que devolve o valor do coseno de x calculado com os 100 primeiros termos dada pela série acima.
OBS.: *A função deve usar no máximo um loop. I.e, não se pode usar loops encaixados.*

7.6 Exercícios

1. Faça três versões da função fatorial:
   function fatorial(n:**integer**):**integer**;
   usando as estruturas de repetição: **for**, **while** e **repeat**.
2. Os números de fibonacci $n_1, n_2, \ldots$ são definidos da seguinte forma:

$$\begin{aligned} n_0 &= 0, \\ n_1 &= 1, \\ n_i &= n_{i-1} + n_{i-2}, \quad i \geq 2. \end{aligned}$$

Faça um programa contendo uma função

function fibonacci(n:**integer**):**integer**;

que retorna o n-ésimo número de fibonacci.

3. Faça um procedimento com um parâmetro inteiro n e que ao ser chamado, o procedimento imprime uma figura da seguinte forma:

```
....*....
...***...
..*****..
.*******.
*********
.*******.
..*****..
...***...
....*....
```

No caso, o procedimento foi chamado com parâmetro 5. A quantidade de linhas impressas é $2n - 1$.

4. O valor de π também pode ser calculado usando a série $S = \frac{1}{1^3} - \frac{1}{3^3} + \frac{1}{5^3} - \frac{1}{7^3} + \frac{1}{9^3} - \frac{1}{11^3} + \cdots$, sendo que o valor de π é calculado como $\pi = \sqrt[3]{32 \cdot S}$. Faça uma função para calcular o valor de π usando esta série e parando quando a diferença do valor de π calculado em uma iteração e a iteração anterior for menor que 0,0001.
5. Escreva um programa contendo uma função

function binario(n:integer):TipoString;

que retorna um tipo chamado *TipoString*, declarado como

type TipoString = string[50];

string contendo o número n na base binária.

6. Faça duas funções que tem um parâmetro inteiro e retornam verdadeiro se o parâmetro for primo e falso caso contrário, uma função usando o método ingênuo apresentado na página 49 e o método otimizado apresentado na página 50. Verifique quanto tempo estas duas funções gastam se usadas para contar todos os primos no intervalo $[1000, 5000]$. Experimente para outros intervalos maiores.
7. Implemente conjuntos através de vetores e faça os seguintes procedimentos:

   (a) Procedimento chamado conjunto, com dois parâmetros, um vetor e um inteiro positivo $n \geq 0$, indicando a quantidade de elementos no vetor. O procedimento deve remover os elementos duplicados do vetor e atualizar o valor de n.

   (b) Procedimento chamado *Intersecao* com seis parâmetros, com o seguinte cabeçalho:
   procedure *intersecao*(**var** v_1:tipovetor;n_1:integer;
   var v_2:tipovetor;n_2:integer;
   var v_3:tipovetor; **var** n_3:integer);
   O vetor v_1 (v_2) contém n_1 (n_2) elementos. O vetor v_3 receberá a interseção dos elementos de v_1 e v_2 e n_3 deve retornar com a quantidade de elementos em v_3 (vamos supor que cada vetor contém elementos distintos).
   Você deve usar a seguinte estratégia para gerar a interseção de v_1 e v_2:
   (1) Ordene o vetor v_1.
   (2) Para cada elemento de v_2, faça uma busca binária do elemento no vetor v_1.
   (2.1) Se o elemento se encontrar no vetor, insira o elemento no vetor v_3.

   (c) Procedimento chamado *Uniao* com seis parâmetros, com o seguinte cabeçalho:
   procedure *uniao*(**var** v_1:tipovetor;n_1:integer;
   var v_2:tipovetor;n_2:integer;
   var v_3:tipovetor; **var** n_3:integer);
   O vetor v_1 (v_2) contém n_1 (n_2) elementos. O vetor v_3 receberá a união dos elementos de v_1 e v_2 e n_3 deve retornar com a quantidade de elementos em v_3 (vamos supor que cada vetor contém elementos distintos).
   Você deve usar a seguinte estratégia para gerar a união de v_1 e v_2:
   (1) Copie o vetor v_1 no vetor v_3 (atualizando n_3).
   (2) Para cada elemento de v_2, faça uma busca binária do elemento no vetor v_1.
   (2.1) Se o elemento não se encontrar no vetor, insira o elemento no vetor v_3.

8. Tem-se um conjunto de dados contendo a altura e o sexo (M ou F) de 50 pessoas. Fazer um algoritmo que calcule e escreva:
   (a) A maior e a menor altura do grupo.
   (b) A média de altura das mulheres.
   (c) O número de homens.
9. Descreva uma função que tenha como parâmetros uma razão, r, um valor inicial, v_0, e um número n e devolva a soma dos n primeiros elementos de uma progressão aritmética começando com v_0 e com razão r.
10. Descreva um procedimento, com parâmetro inteiro positivo n e dois outros parâmetros b e k que devem retornar valores inteiros positivos. Os valores a serem retornados em b e k são tais que b seja o menor valor inteiro tal que $b^k = n$.
11. Faça uma função booleana com parâmetro n e que retorna verdadeiro se n é primo, falso caso contrário.
12. Faça uma função que tenha como parâmetro uma temperatura em graus Fahrenheit e retorne a temperatura em graus Celsius. Obs.: $(C = 5/9 \cdot (F - 32))$.
13. O imposto que um trabalhador paga depende da sua faixa salarial. Existem até k faixas salariais, cada uma com uma correspondente taxa. Exemplo de um sistema com até 4 faixas salariais:
   (a) Para salários entre 0 e 100 reais, é livre de imposto.
   (b) Para salários maiores que 100 e até 500 reais, é 10 % de imposto.
   (c) Para salários maiores que 500 e até 2000 reais, é 20 % de imposto.
   (d) Para salários maiores que 2000 é 30 % de imposto.

   Faça um programa que leia estas k faixas salariais e leia uma seqüência de salários e imprima para cada um, o imposto a pagar. O programa deve parar quando for dado um salário de valor negativo.
14. Faça um programa de lotérica, que lê o nome de n jogadores e os números que eles apostaram (um número entre 0 e 100). Use a função RANDOM(N) para sortear um número. Se houver ganhador, imprima o nome dele e o número que ele apostou, caso contrário, avise que ninguém ganhou.

8 Processamento de Cadeias de Caracteres

Em praticamente todos os padrões de codifiação, a representação interna das letras está agrupada e ordenada. Este é um dos motivos dos computadores poderem fazer comparações com letras, como 'A'<'B' rapidamente, internamente é comparada sua codificação interna. Além disso, a codificação das letras e números é em geral seqüêncial. Um exemplo disto é a codificação ASCII (veja tabela 1).

Como vimos, uma cadeia de caracteres pode ser definida com o tipo **string**. Além disso, podemos fazer operações com os caracteres que estão nesta string (cadeia). Considere as seguintes declarações:

```
type TipoTexto = string[1000];
var Texto : TipoTexto;
```

Com estas declarações, podemos fazer a seguinte atribuição para a variável *Texto*: `Texto := 'Algoritmo';`
Uma função que devolve o comprimento da cadeia de caracteres é a função **length**. Após a atribuição acima, **length***(Texto)* deve retornar o inteiro 9. Além disso, podemos trabalhar com cada caracter deste texto, como se *Texto* fosse um vetor de 9 posições, i.e., podemos acessar e atualizar um elemento da cadeia. Assim, após a atribuição da variável *Texto* feita acima, temos em *Texto[1]* a letra 'A', *Texto[2]* a letra 'l', *Texto[3]* a letra 'g', ..., *Texto[9]* a letra 'o'.

Observações:

1. Em Turbo Pascal há uma limitação do tamanho máximo de uma String, de 255 caracteres. A declaração de variáveis usando apenas a palavra reservada "String" denota o tipo string com 255 caracteres. No Turbo Pascal a passagem de tipos é feita de maneira rígida, de tal forma que uma variável declarada como String[255] não pode ser enviada como um parâmetro declarado apenas como o tipo string.

2. Em Delphi, "String" denota normalmente o tipo "long string" ou "AnsiString" cujo limite de comprimento máximo é 2 GB! Com a diretiva de compilação $H-$, "String" denota o tipo "String[255]" como em Turbo Pascal. Outra alternativa equivalente é usar o tipo pré-definido "ShortString".

3. Em Gnu Pascal não é possível declarar variáveis usando apenas a palavra "String", mas na passagem de parâmetros e ponteiros sim. Desta maneira poderemos ter variáveis declaradas como strings de tamanhos diferentes mas que são recebidas como parâmetros do tipo String (sem especificação do tamanho).

4. Em Extended Pascal também não é possível se usar apenas a palavra String para declarar variáveis, além disso o tamanho da string é declarado usando se parenteses em vez de colchetes. Por exemplo, a seguinte declaração apresenta uma string em Extended Pascal com 100 caracteres.
   var nome:string(100);

Nos exemplos deste texto usaremos a seguinte sintaxe:

- Na declaração de variáveis e tipos usaremos colchetes com o devido tamanho da string.
- Na passagem de parâmetros daremos prioridade para a sintaxe usando a palavra string. No caso onde a rotina tem funcionalidade fixa para um determinado tipo usaremos o nome do tipo no lugar da palavra string.

A seguir apresentamos algumas rotinas para manipulação de strings.

Função	*Resultado*
Length(s)	Retorna a quantidade de caracteres da string s
SetLength(s, n)	Redefine a quantidade de caracteres em uso como n
Pos(s_1, s_2)	Retorna a posição da cadeia s_1 na cadeia s_2 (0 se não ocorre)
Concat$(s_1, s_2, \ldots)$	Retorna a concatenação de $s_1, s_2, \ldots$ (equivale a $s_1 + s_2 + \ldots$)
Copy(s, p, n)	Retorna a cadeia formada pelos n caracteres a partir da posição p
Insert(s_1, s_2, p)	Insere s_1 em s_2 a partir da posição p
Delete(s, p, n)	Remove os n caracteres a partir da posição p

Exemplo 8.1 *Descreva uma implementação em pascal da função Pos.*

```
function Pos(var s1,s2: String): Integer;
var n1,n2,d,i,j       : Integer;  achou, cont: Boolean;
begin
  n1 := Length(s1); n2 := Length(s2); d := n2−n1+1;
  achou := False;
  i :=1;
  while (not achou) and (i<=d) do
  begin
    j := 1;  cont := True;
    while  cont and (j<=n1) do
    begin
      cont := (s1[j]=s2[i+j−1]);
      inc(j)
    end;
    achou := cont;
    inc(i);
  end;
  if achou then Pos := i−1
  else Pos := 0
end; {Pos}
```

Exemplo 8.2 *Descreva uma implementação em pascal da função Insert.*

```
function Insert(var s1,s2: String; p: Integer);
{Insere os caracteres de s1 que couberem em s2}
const ComprMax = 255;
var n1,n2,d,i : Integer;
begin
  n1 := Length(s1); n2 := Length(s2);
  if (n1+n2)>ComprMax
    then d := ComprMax−n2
    else d := n1;
  i := 1;
  for i:= n2 downto n2−d+1 do
    s2[i+d] := s[i];
  for i:=1 to d do
    s2[p+i−1] := s1[i];
  SetLength(s2,n2+d)
end; {Insere}
```

Exemplo 8.3 *Uma cadeia de caracteres é dita ser palíndrome se a seqüência dos caracteres da cadeia da esquerda para a direita é igual a seqüência de caracteres da direita para a esquerda. Por exemplo: As seguintes cadeias de caracteres são palíndromes: ABC12321CBA, ACCA, XYZ6.6ZYX.*
Faça uma função que retorna verdadeiro se a cadeia de caracteres enviada como parâmetro é palíndrome.

```
function Palindrome(var s: String): Boolean;
var i,m,n : Integer; p: Boolean;
begin
  p := True;    n := Length(s);
  m := n div 2;    i := 1;
  while p and (i<=m) do begin
    p := (s[i]=s[n−i+1]);
    inc(i);
  end;
  Palindrome := p
end; {Palindrome}
```

8.1 Letras com Acêntos

Em muitas situações temos cadeias de caracteres seguindo uma codificação para acentuação das letras (eg. textos escritos na língua portuguesa). As codificações de acentuação não são consideradas na codificação ASCII e na maioria das vezes não seguem uma padronização envolvendo diferentes plataformas (i.e., uma letra acentuada em um ambiente Unix pode ter código diferente da mesma letra acentuada em ambiente MS-DOS).

Uma tarefa comum que temos de fazer, quando consideramos acêntos, é a remoção destes do texto. A seguir apresentamos duas aplicações onde precisamos fazer um pré-processamento para remover os acentos:

1. *Envio de mensagens pela internet por correio eletrônico.*
   A internet é uma rede de computadores que não necessariamente seguem a mesma configuração. Os textos como as mensagens enviadas por correio eletrônico são enviadas considerando a configuração ASCII. Por outro lado, letras acentuadas em um computador podem não ser consideradas realmente letras em outros computadores. Assim, estes podem ser interpretados como códigos de ação totalmente imprevisível. Uma solução para isso é eliminar os acentos, ou trocá-los por uma seqüência de letras ASCII que representem a letra acentuada.

2. *Ordenação de cadeias de caracteres.*
   Para comparar duas cadeias de caracteres, a maioria dos programas considera as representações binárias caracter a caracter. Considerando a codificação ASCII, a ordenação de letras em ASCII se torna fácil, uma vez que temos apenas que comparar letra a letra nesta codificação. Por outro lado, teremos uma ordenação errada caso tenhamos acentos.

   Por exemplo, considere as palavras *macro, macaco* e *maço*. Se a codificação de 'ç' vier antes de 'c', então teremos a seguinte ordem para estas palavras: *maço, macaco* e *macro*, que não é a ordem que desejamos. Por outro lado, se a codificação de 'c' vier antes de 'ç', então teremos a ordem: *macaco, macro* e *maço*, que também não é a desejada. Assim, uma solução para isto é comparar as palavras trocando as letras acentuadas pelas correspondentes letras sem acento.

Exemplo 8.4 *O seguinte programa contém uma função que dado uma letra, retorna a letra sem acentuação.*
Obs.: Esta implementação é dependente da codifição das letras usada no programa fonte.

```
program acentos;
function SemAcento(caracter : char):char;
begin
  case caracter of
   'á' : SemAcento:='a';     'Á' : SemAcento:='A';     'é' : SemAcento:='e';
   'É' : SemAcento:='E';     'í' : SemAcento:='i';     'Í' : SemAcento:='I';
   'ó' : SemAcento:='o';     'Ó' : SemAcento:='O';     'ú' : SemAcento:='u';
   'Ú' : SemAcento:='U';     'à' : SemAcento:='a';     'À' : SemAcento:='A';
   'â' : SemAcento:='a';     'Â' : SemAcento:='A';     'ê' : SemAcento:='e';
   'Ê' : SemAcento:='E';     'ô' : SemAcento:='o';     'Ô' : SemAcento:='O';
   'ã' : SemAcento:='a';     'Ã' : SemAcento:='A';     'õ' : SemAcento:='o';
   'Õ' : SemAcento:='O';     'ü' : SemAcento:='u';     'Ü' : SemAcento:='U';
   'ç' : SemAcento:='c';     'Ç' : SemAcento:='C';
   else SemAcento:=caracter;
  end; { case }
end;
var  c : char;
begin
  c:='á'; writeln('A letra ',c,' sem acento fica ',SemAcento(c));
  c:='é'; writeln('A letra ',c,' sem acento fica ',SemAcento(c));
  c:='ã'; writeln('A letra ',c,' sem acento fica ',SemAcento(c));
  c:='õ'; writeln('A letra ',c,' sem acento fica ',SemAcento(c));
end.
```

Exercício 8.1 *Faça uma função que tem como parâmetro uma cadeia de caracteres e retorna a mesma cadeia sem acentos.*

8.2 Transformação entre Maiúsculas e Minúsculas

Uma maneira de se tranformar uma cadeia de caracteres trocando cada letra por sua correspondente em maiúscula é percorrer toda a cadeia e para cada caracter colocar uma seqüência de testes, um para cada letra minúscula, e trocá-la pela correspondente letra maiúscula. Um algoritmo deste tipo iria requerer uma estrutura com pelo menos 25 condições, tornando o programa longo e lento.

Uma maneira mais eficiente de se implementar tal procedimento, é considerar a representação interna de cada caracter. Considerando que internamente cada letra é representada em um byte, e a representação das letras é seqüencial A idéia é mudar apenas aqueles caracteres que estiverem no intervalo *['a',...,'z']*. Quando ocorrer um caracter c neste intervalo, obtemos o valor inteiro em ASCII de c e seu deslocamento d a partir do caracter 'a'. Em seguida, reatribuímos o caracter que estiver na posição de *'A'* mais o deslocamento d.

```
program ProgramMaiusculas;
type TipoTexto = string[1000];
procedure Maiuscula(var texto : TipoTexto);
var i,tam : integer;
begin
  tam := length(texto);
  for i:=1 to tam do
    if ('a'<=texto[i]) and (texto[i]<='z') then
      texto[i] := chr( ord('A') + ord(texto[i]) − ord('a'));
end;
var Cadeia : TipoTexto;
begin
  write('Entre com um texto: ');
  readln(cadeia);
  Maiuscula(cadeia);
  writeln('O texto em maiúsculo é :',cadeia);
end.
```

Exercício 8.2 *Faça um programa análogo ao apresentado acima, mas para tranformar uma cadeia em minúsculas.*

8.3 Casamento de Padrões

Nesta seção vamos construir uma função para que dados duas cadeia de caracteres, uma chamada *texto*, e a outra chamada *padrão*, verifica se existe uma ocorrência de *padrão* no *texto*. Caso ocorra uma ocorrência, a função retorna a posição no *texto* onde ocorre o *padrão*, caso contrário, a função retorna 0.

Uma idéia simples de implementação é ir percorrendo todas as posições possíveis de *texto*, de se começar o *padrão* (i.e., posições $1, \ldots, length(texto) - length(padrao) + 1$). Para cada uma destas posições, há uma outra estrutura de repetição que verifica se o padrão está começando naquela posição. A função para assim que encontrar o primeiro padrão, ou até que todas as possibilidades tenham sido testadas. O seguinte programa implementa esta idéia.

```
program ProgramaBuscaPadrao;

type TipoString      = string[1000];
  TipoPadrao      = string[50];

{Retorna a posição do índice onde começa o padrao, 0 caso não exista. }
function BuscaPadrao(var texto : TipoString; var Padrao:TipoPadrao): integer;
var
  i,j,TamTexto,TamPadrao  : integer;
  achou,SubsequenciaIgual : boolean;
begin
  TamTexto := length(texto);
  TamPadrao := length(padrao);
  i := 0;
  achou := false;
  while (not achou) and (i<=TamTexto−TamPadrao) do begin
    i:=i+1;
    j:=1;
    SubSequenciaIgual := true;
    while (j<=TamPadrao) and (SubsequenciaIgual=true) do
      if (Padrao[j]=Texto[i+j−1]) then j:=j+1
      else SubSequenciaIgual := false;
    achou := SubSequenciaIgual;
  end;
  if (achou) then BuscaPadrao := i
  else BuscaPadrao :=0;
end; { BuscaPadrao }

var texto : TipoString;
  padrao : TipoPadrao;
  pos        : integer;
begin
  write('Entre com um texto: ');
  readln(texto);
  write('Entre com um padrao: ');
  readln(padrao);
  pos := BuscaPadrao(texto,padrao);
  if (pos=0) then writeln('Padrao não encontrado.')
  else writeln('Padrao encontrado na posição ',pos);
end.
```

Obs.: Existem outros algoritmos para fazer busca de padrões que são computacionalmente mais eficientes, como o algoritmo de Knuth, Morris e Pratt e o algoritmo de Boyer e Moore.

Exercício 8.3 *Faça uma rotina para fazer busca em uma cadeia de caracteres, mas com um padrão contendo símbolos da seguinte maneira:*

$\backslash\backslash$ *Estas duas barras indicam apenas um caracter que é o caracter barra* ($\backslash$).

$\backslash\star$ *Indica o caracter asterisco* ($\star$)

$\backslash?$ *Indica o caracter de interrogação (?).*

$\star$ *Indica uma quantidade qualquer de caracteres (pode ser vazio).*

? *Indica um caracter qualquer (não pode ser vazio).*

Assim, se tivermos duas cadeias C_1 *e* C_2*, digamos*

C_1=*Algoritmos e Programação de Computadores*

C_2=*Produção de Programas de Computador*

A seguinte tabela mostra a pertinência de alguns padrões nestas duas cadeias:

Padrão	C_1	C_2
Pro$\star$*ção*$\star$*de*$\star$*Computador*	$\times$	$\times$
ritmo$\star$*de*$\star$*dor*	$\times$	
Alg$\star$*Programa??o*	$\times$	
r$\star$*e*$\star$*ma*$\star$*dor*	$\times$	$\times$

A rotina deve retornar a posição do primeiro caracter na cadeia que foi associado ao padrão, caso não exista, retorna 0.

8.4 Criptografia por Substituições - Cifra de César

Criptografia é a ciência e o estudo de escrita secreta. Um *sistema criptográfico* é um método secreto de escrita pelo qual um texto legível é transformado em texto cifrado. O processo de transformação é conhecido como *ciframento* e a transformação inversa é conhecida como *deciframento*.

A idéia aqui não é a de apresentar métodos seguros de criptografia, mas sim de trabalhar mais com as cadeias de caracteres.

Provavelmente os primeiros métodos de criptografia usavam métodos de substituição. Dado dois alfabetos $\mathcal{A}$ e $\mathcal{A}'$, uma função de criptografia por substituição troca um caracter de um texto $\mathcal{A}$ por outra em $\mathcal{A}'$. Naturalmente deve haver uma função inversa para que o receptor da mensagem criptografada possa recuperar o texto original.

Cifra de César

Um caso particular do método de substituição é o seguinte. Considere um alfabeto $\mathcal{A} = (c_0, c_1, \ldots, c_{n-1})$ com n símbolos. Considere um inteiro k, $0 \leq k \leq n-1$ (chamado de chave). Uma função de criptografia $f_k : \mathcal{A} \rightarrow \mathcal{A}$ e sua inversa f_k^{-1} (para decriptografar) são dadas a seguir:

$$f_k(c_i) = c_{(i+k) \bmod n} \quad f_k^{-1}(c_i) = c_{(i-k+n) \bmod n}$$

Assim, se o alfabeto é $\mathcal{A} = (A, B, C, D, E, F, G, H, \ldots, S, T, U, V, W, X, Y, Z)$, e $k = 3$ a função transforma A em D, B em E, C em F, ... I.e.,

A	B	C	D	E	F	G	H	$\ldots$	V	W	X	Y	Z
$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\ldots$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
D	E	F	G	H	I	J	K	$\ldots$	Y	Z	A	B	C

Este tipo de criptografia também é chamado de cifra de César, porque Júlio César usou com $k = 3$. A seguir apresentamos um programa que criptografa um texto, com o alfabeto das letras minúsculas, usando este tipo de criptografia, transformando apenas as letras e deixando os demais símbolos intactos.

```
program progcifracesar;
type tipostring     = string[100];
procedure criptocesar(var texto,cifra : tipostring; k:integer);
var i,tam : integer;
begin
  cifra := '';
  tam := length(texto);
  for i:=1 to tam do begin
    if (texto[i]>='a') and (texto[i]<='z') then
      cifra := cifra + chr((ord(texto[i]) - ord('a') + k) mod 26 + ord('a'))
    else cifra := cifra + texto[i];
  end;
end; { criptocesar }
var Texto,Cripto : tipostring;
begin
  write('Entre com uma cadeia de caracteres: ');
  readln(texto);
  criptocesar(texto,cripto,3);
  writeln('A texto criptografado é: ', cripto);
end.
```

Exercício 8.4 *Faça um procedimento para decriptografar tendo como dados de entrada a cadeia de caracteres criptografada pelo programa acima e o valor usado de k.*

Os métodos de substituição são rápidos, mas pouco seguros. Em geral, cada língua possui freqüências diferentes para as ocorrências de cada letra.

Exemplo 8.5 *Em um texto em inglês, temos em geral as seguintes freqüências esperadas para as letras:*

$$\begin{array}{lll} A = 8,0\% & J = 0,5\% & S = 6,0\% \\ B = 1,5\% & K = 0,5\% & T = 9,0\% \\ C = 3,0\% & L = 3,5\% & U = 3,0\% \\ D = 4,0\% & M = 3,0\% & V = 1,0\% \\ E = 13,0\% & N = 7,0\% & W = 1,5\% \\ F = 2,0\% & O = 8,0\% & X = 0,5\% \\ G = 1,5\% & P = 2,0\% & Y = 2,0\% \\ H = 6,0\% & Q = 0,2\% & Z = 0,2\% \\ I = 6,5\% & R = 6,5\% & \end{array}$$

Assim, para decriptografar um texto, onde o original é em inglês, calculamos as freqüências dos símbolos no texto criptografado e em seguida relacionamos com as freqüências acima. Isto permite que possamos relacionar os símbolos com os caracteres originais mais prováveis.

Exercício 8.5 *Faça um programa que lê um número inteiro n, $0 \leq n \leq 1000000$ e escreve seu valor por extenso. Exemplos:*

- *1 = um*
- *19 = dezenove*
- *20 = vinte*
- *21 = vinte e um*
- *100 = cem*
- *111 = cento e onze*
- *1000 = mil*
- *2345 = dois mil trezentos e quarenta e cinco*
- *9012 = nove mil e doze*
- *12900 = doze mil e novecentos*
- *100101 = cem mil cento e um*
- *110600 = cento e dez mil e seiscentos*
- *999999 = novecentos e noventa e nove mil novecentos e noventa e nove*
- *1000000 = um milhão*

Sugestão: Escreva funções ou procedimentos convenientes que tratam de números:

- *até 9*
- *até 19*
- *até 99*
- *até 999*
- *até um milhão*

Note que as funções para intervalos maiores podem usar as funções para intervalos menores. Pense com cuidado na lógica do uso da conjunção 'e'.

Exercício 8.6 *(Cálculo de dígito verificador) Muitas vezes, quando queremos representar objetos de maneira única, definimos um código para ele. Por exemplo, o sistema da Receita Federal usa o CPF para representar uma pessoa física (note que o nome não é uma boa maneira de representar um objeto, uma vez que podem ocorrer várias pessoas com o mesmo nome). Certamente o banco de dados de pessoas físicas é bem grande e uma busca de uma pessoa através de um código incorreto resultaria em um grande desperdício de tempo de processamento. Assim, para diminuir as chances de uma busca com um código errado, usamos dígitos verificadores nestes códigos. É o caso do CPF, onde os dois últimos dígitos são dígitos verificadores. O CPF é formado por onze dígitos $d_1, d_2, \ldots, d_{11}$ e seus dígitos verificadores são calculados da seguinte maneira:*

- *O dígito d_{10} é calculado da seguinte maneira*

$$d_{10} \leftarrow 11 - \left(\sum_{i=1}^{9} (10 - i) \cdot d_i \right) \mathbf{mod}\ 11.$$

  Caso este cálculo leve a um valor maior que 9, então d_{10} recebe o valor 0.

- *O dígito d_{11} é calculado da seguinte maneira*

$$d_{11} \leftarrow 11 - \left(\sum_{i=1}^{9} (11 - i) \cdot d_i \ + \ 2d_{10} \right) \mathbf{mod}\ 11.$$

  Caso este cálculo leve a um valor maior que 9, então d_{11} recebe o valor 0.

Faça um programa que leia um código, possívelmente com caracteres '.', '-', '/' e ' ' e verifique se este código representa um CPF correto.

8.5 Exercícios

1. Faça um programa para converter um número romano para número no sistema decimal. Ex.: O número romano CMXCIX é o número 999 no sistema decimal. Obs.: No sistema romano para o decimal, M=1000, D=500, C=100, L=50, X=10, V=5 e I=1.

2. Faça um programa que converte um número decimal em um número romano.

3. Um programador está implementando um processador de textos e quer adicionar uma facilidade para a composição de cartas, colocando um comando que com apenas os dados da data (dia, mês e ano), apresenta o seguinte cabeçalho:

   <Dia da semana>, <dia> de <mês> de <ano>

   Ex.: Colocando a data 01/01/1901, temos o seguinte cabeçalho:

   Terça-feira, 1 de janeiro de 1901

   Para implementar esta facilidade, ajude este programador construindo uma função que tem como parâmetros o *dia*, o *mês* e o *ano* e retorna uma string contendo o cabeçalho como no exemplo acima. Considere que as datas estão no intervalo de 01/01/1901 a 01/01/2099.

4. Pelo calendário gregoriano, intituído em 1582 pelo Papa Gregório XIII, os anos bisextos são aqueles cujo ano são divisíveis por 4, exceto os anos que são divisíveis por 100 e não por 400. Por exemplo, os anos 1600, 2000 são anos bisextos enquanto os anos 1700, 1800 e 1900 não são. Com isto, em mente, resolva o exercício anterior para considerar anos a partir de 1600.
   Obs.: O dia 01/01/1600 é um sábado.

5. O programa apresentado na seção 8.4 restringe a criptografia ao alfabeto $\{a, \ldots, z\}$. Faça um programa para criptografar e decriptografar considerando o alfabeto formado pelo tipo **byte**.

6. Faça um programa contendo funções com os seguintes cabeçalhos:

   - **function** criptografa(str : string; k:integer):tipostring;
   - **function** decriptografa(str : string; k:integer):tipostring;

- **function** maiuscula(str : string):tipostring;
- **function** minuscula(str : string):tipostring;
- **function** semacentos(str : string):tipostring;
- **function** cadeialivre(str : string):tipostring;

onde *tipostring* é definido como:

type tipostring=string[255];

As funções criptografa e decriptografa são para criptografar e decriptografar usando a cifra de César, com deslocamento k e o alfabeto das letras minúsculas. As funções maiúscula e minúscula são para retornar a função dada em maiúsculo e minúsculo. A função semacentos retorna a cadeia de caracteres str sem as letras dos acentos (ela preserva maiúsculas e minúsculas). A função cadeia livre transforma a cadeia de caracteres str em uma cadeia sem acentos e com todos as letras em minúsculo.

O programa deve ler uma cadeia de caracteres e uma opção para uma das seis operações acima. Caso a opção seja para criptografar ou decriptografar, o parâmetro k também deve ser lido. Após a execução de uma opção, a cadeia de caracteres resultante deve ser impressa.

9 Variáveis Compostas Heterogêneas - Registros

A linguagem Pascal nos permite especificar objetos formados por diversos dados associados a eles, possivelmente de tipos diferentes. Assim, em apenas um objeto (variável) poderemos ter vários outros dados como por exemplo do tipo string, integer, real, ... Vamos chamar este tipo de objeto por *registro*. Poderemos usar dois tipos de registros: um onde os diferentes tipos de dados estão armazenados em memória distinta e outro quando parte dos dados estão armazenados em um mesmo endereço de memória.

9.1 Registros Fixos

Para se especificar um registro onde cada campo está definido em uma determinada memória (sem interseção com os demais campos), usamos a seguinte sintaxe (que pode ser usada para definir tanto variáveis como tipos):

```
record
    Lista_de_Identificadores_do_Tipo_1 : Tipo_1;
    Lista_de_Identificadores_do_Tipo_2 : Tipo_2;
                    ⋮
    Lista_de_Identificadores_do_Tipo_K : Tipo_K;
end
```

Na sintaxe acima, cada identificador é chamado de *Campo* do registro. Além disso, cada um dos tipos (Tipo_i) também pode ser a especificação de outro registro.

Para acessar um campo chamado **Campo1** de um registro chamado **Reg1**, usamos a seguinte sintaxe:

Reg1.Campo1

Caso Campo1 seja também um registro e contenha o campo **SubCampo11**, acessamos este último da seguinte maneira:

Reg1.Campo1.SubCampo11

Números Complexos

Como tipos numéricos a linguagem Pascal oferece os tipos **integer** e **real**, mas a maioria dos compiladores Pascal não oferece um tipo especial para tratar números complexos. Como um número complexo é dividido em duas partes na forma $n = a + b \cdot i$, onde a e b são números reais, precisamos de um tipo que contemple estes dois números em apenas um tipo. Com isso, podemos definir um tipo chamado *complexo* que contém as duas partes de um número complexo. No seguinte quadro, definimos um tipo complexo usando **record** e apresentamos funções para fazer soma, subtração e multiplicação de números complexos:

É importante observar que a sintaxe da linguagem Pascal não permite acessar os campos da própria variável de retorno de função. Devemos usar uma variável para receber um valor de retorno e em seguida atribuir esta variável para o retorno de função, sendo este o motivo de usarmos a variável z nas rotinas *SumComplex, SubComplex* e *MulComplex*.

```
program ProgComplex;
type complex = record
        a,b : real; {Número na forma (a+b.i) }
      end;
procedure LeComplex(var x : complex); { Le um número complexo }
begin
  writeln('Entre com um número complex (a+b.i) entrando com a e b: ');
  readln(x.a,x.b);
end; { LeComplex }
procedure ImpComplex(x : complex);  { Imprime um número complexo }
begin
  writeln('(',x.a:5:2,'+',x.b:5:2,'i)');
end; { ImpComplex }
function SomComplex(x,y:complex):complex; { Retorna a soma de dois números complexos }
var z : complex;
begin
  z.a := x.a+y.a;    z.b:=x.b+y.b;
  SomComplex := z;
end; { SomComplex }
function SubComplex(x,y:complex):complex; { Retorna a subtração de dois números complexos }
var z : complex;
begin
  z.a := x.a−y.a;    z.b:=x.b−y.b;
  SubComplex := z;
end; { SubComplex }
function MulComplex(x,y:complex):complex; { Retorna a multiplicação de dois números complexos }
var z : complex;
begin
  z.a := x.a*y.a−x.b*y.b;    z.b:=x.a*y.b+x.b*y.a;
  MulComplex := z;
end; { MulComplex }
var x,y      : complex;
begin
  LeComplex(x); LeComplex(y);
  write('A soma dos números complexos lidos é '); ImpComplex(SomComplex(x,y));
  write('A subtração dos números complexos lidos é '); ImpComplex(SubComplex(x,y));
  write('A multiplicação dos números complexos lidos é '); ImpComplex(MulComplex(x,y));
end.
```

Exercício 9.1 *Um número racional é definido por duas partes inteiras, o numerador e o denominador. Defina um tipo chamado* **racional** *usando a estrutura* **record** *para contemplar estas duas partes. Além disso, faça funções para ler, somar, subtrair, dividir e simplificar números racionais. Por simplificar, queremos dizer que o número racional $\frac{a}{b}$ tem seu númerador e denominador divididos pelo máximo divisor comum entre eles (veja programa do exemplo 7.10).*

Cadastro de Alunos

Suponha que você tenha um cadastro de alunos onde cada aluno contém as seguintes características: Nome, Data de Nascimento (dia, mês e ano), RG, Sexo, Endereço (Rua, Cidade, Estado, CEP), RA (Registro do Aluno) e CR (Coeficiente de Rendimento: número real no intervalo $[0, 1]$).

Note que um aluno deve ter as informações da data de nascimento que podem ser divididas em três partes (dia, mês e ano). Sendo assim, definiremos um tipo chamado TipoData que representará uma data. Note que o mesmo ocorre com endereço assim, usaremos um tipo chamado TipoEndereco que representará um endereço. Obs.: Não necessariamente precisávamos definir tipos separados para data e endereço, mas sempre que temos informações com um tipo independente e com certa "vida própria", é razoável se definir um tipo particular a ele.

O seguinte quadro apresenta a declaração do tipo aluno e os demais tipos necessários para definí-lo.

```
type
  TipoNome      = string[50];
  TipoRG        = string[10];
  TipoDia       = 1..31;
  TipoMes       = 1..12;
  TipoAno       = integer;
  TipoRua       = string[50];
  TipoEstado    = string[2];
  TipoCep       = string[9];
  TipoRA        = string[6];
  TipoCR        = real;
  TipoData      = record
             Dia : TipoDia;
             Mes : TipoMes;
             Ano : TipoAno;
           end;
  TipoEndereco  = record
             Rua    : TipoRua;
             Cidade : TipoCidade;
             Estado : TipoEstado;
             CEP    : TipoCep;
           end;
  TipoAluno     = record
             Nome           : TipoNome;
             RG             : TipoRG;
             DataNascimento : TipoData;
             Endereco       : TipoEndereco;
             RA             : TipoRA;
             CR             : TipoCR;
           end;
```

Usando as declarações dos tipos acima, exemplificamos, nos dois quadros seguinte, o acesso e uso destes tipos com variáveis. Os dois programas apresentados são equivalentes.

```
{ Supondo feita as declarações }
{ dos tipos do quadro acima }
var
  Aluno    : TipoAluno;
  Data          : TipoData;
  Endereco : TipoEndereco;
begin
  Aluno.Nome := 'Fulano de Tal';
  Aluno.RG  := '9999999999';
  Data.Dia   := 1;
  Data.Mes   := 1;
  Data.Ano   := 2000;
  Aluno.DataNascimento := Data;
  Endereco.Rua := 'R. Xxx Yyy, 999';
  Endereco.Cidade := 'Campinas';
  Endereco.Estado := 'SP';
  Aluno.Endereco := Endereco;
  Aluno.RA   := '999999';
  Aluno.CR   := 0.99;
  Writeln('O aluno ',Aluno.Nome,' mora em ',
       Aluno.Endereco.Cidade,'-',
       Aluno.Endereco.Estado,'.');
end.
```

```
{ Supondo feita as declarações }
{ dos tipos do quadro acima }
var
  Aluno    : TipoAluno;


begin
  Aluno.Nome := 'Fulano de Tal';
  Aluno.RG   := '9999999999';
  Aluno.DataNascimento.Dia   := 1;
  Aluno.DataNascimento.Mes   := 1;
  Aluno.DataNascimento.Ano   := 2000;

  Aluno.Endereco.Rua := 'R. Xxx Yyy, 999';
  Aluno.Endereco.Cidade := 'Campinas';
  Aluno.Endereco.Estado := 'SP';

  Aluno.RA   := '999999';
  Aluno.CR   := 0.99;
  Writeln('O aluno ',Aluno.Nome,' mora em ',
       Aluno.Endereco.Cidade,'-',
       Aluno.Endereco.Estado,'.');
end.
```

Exercício 9.2 *Usando as declarações de tipo apresentadas no quadro da página 9.1, faça um cadastro de alunos com um vetor com 100 posições, cada posição do tipo TipoAluno, i.e.,*

```
type TipoCadastro = record
        Quant: integer;
        Aluno: array [1..100] of TipoAluno;
end
```

O campo Quant representa a quantidade de alunos efetivamente inseridos no vetor. O programa deve manipular este cadastro com as seguintes opções:

1. *Iniciar cadastro vazio (inicialmente sem elementos).*
2. *Inserir um novo aluno no cadastro (se o cadastro estiver cheio, avise que não há memória disponível).*
3. *Ordenar o cadastro por nome em ordem alfabética.*
4. *Ordenar o cadastro por CR, maiores primeiro.*
5. *Ler o valor de um RA e imprimir os dados do aluno no cadastro com mesmo RA.*
6. *Imprimir o cadastro na ordem atual.*
7. *Sair do programa.*

9.2 Registros Variantes

A linguagem Pascal permite que possamos definir um registro com uma parte variante. Nesta parte variante, podemos ter diferentes conjuntos de campos. A idéia aqui envolve a situação onde o valor da característica de um objeto pode implicar em conjuntos de características diferentes.

Por exemplo, suponha que temos registros com informações dos móveis de uma casa. Vamos supor que temos apenas três *tipos* de móveis: (*Armário, Mesa, Sofá)*. Todos os móveis considerados têm atributos em comum, que são: *(cor, material, comprimento, largura, altura)*. Mas se o móvel for um armário, então ele tem um atributo particular que é o *número de gavetas*. Se o móvel for uma mesa, ela tem outro atributo que é a forma, se *circular* ou *retangular* e o *número de cadeiras* que usa. Se o móvel for um sofá, então ele tem um atributo especial que é a *quantidade de lugares* para se sentar. Além disso, um atributo particular de um tipo de móvel não faz sentido para outro tipo de móvel.

Se formos usar o tipo de registro fixo, então deveriamos colocar todos os atributos no registro que especifica o móvel, cada um usando uma parte da memória. Mas note que se o móvel for um Armário, então o campo que indica a quantidade de cadeiras de uma mesa é um desperdício de memória e não será usado. A idéia de usar registros variantes é que cada conjunto de atributos particulares esteja começando na mesma posição de memória.

Uma restrição da linguagem Pascal é que podemos ter apenas uma parte que é definida como parte variante. Sua sintaxe é definida da seguinte maneira:

record
    Lista_de_Identificadores_do_Tipo_1 : Tipo_1;
    ⋮
    Lista_de_Identificadores_do_Tipo_K : Tipo_K;
case *[CampoSelecionador:] Tipo_Selecionador* **of**
    Escalar_1 : (Lista_de_Declaração_de_Campos_1);
    ⋮
    Escalar_P : (Lista_de_Declaração_de_Campos_P);
end;

A lista de identificadores dos tipos *Tipo_1...Tipo_K* formam a parte fixa do registro. A parte variante começa depois da palavra **case**. O *CampoSelecionador* é opcional. A idéia é que caso o *CampoSelecionador* seja igual a *Escalar_1*, então os campos válidos serão os que estão na *Lista_de_Declaração_de_Campos_1*. Caso o *CampoSelecionador* seja igual a *Escalar_2*, então os campos válidos serão os que estão na *Lista_de_Declaração_de_Campos_2*, e assim por diante. O tipo do CampoSelecionador deve ser necessariamente um escalar. Note também que temos apenas uma palavra **end**, i.e., a palavra **end** termina a definição de todo o registro.

No programa do quadro seguinte apresentamos as declarações e algumas atribuições usando registro variante no exemplo dos móveis, apresentado acima.

```
program ProgMobilia;
type
  TipoMaterial: (Madeira, Metal, Couro, Sintetico);
  TipoCor:      (Branco, Preto, Azul, Amarelo, Vermelho, Verde, Cinza);
  TipoMovel = record
          Cor          : TipoCor;
          Altura,Comprimento,Largura : real;
          Material     : TipoMaterial;
          case Tipo  : (Mesa,Sofa,Armario) of
            Armario    : (NoGavetas:integer);
            Mesa       : (Forma:(Circular,Retangular); NoCadeiras:integer);
            Sofa       : (NoLugares:integer);
        end;
var M1,M2 : TipoMovel;
begin
  M1.Tipo := Sofa; {Definindo o atributo selecionador}
  M1.Cor := Amarelo;   M1.Material := Couro;
  M1.Largura:=1.0;     M1.Altura:=0.9;   M1.Comprimento :=3.0;
  M1.NoLugares := 3; {Atributo particular de sofá}

  M2.Tipo := Mesa; {Definindo o atributo selecionador}
  M2.Cor := Azul;      M2.Material := Madeira;
  M2.Largura:=1.5;     M2.Altura:=1.2;   M2.Comprimento :=2.0;
  M2.NoCadeiras := 6;      {Atributo particular de mesa}
  M2.Forma := Retangular; {Atributo particular de mesa}
  {...}
end.
```

O acesso aos campos do registro é feita da mesma forma como nos registros fixos. A seguinte figura apresenta a configuração da memória para o registro do tipo *TipoMovel* apresentado no quadro acima.

<table>
<tr><td colspan="6">$\overbrace{\hspace{20em}}^{\text{Parte Fixa}}$</td><td colspan="2">Parte Variante</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="2">NoGavetas</td></tr>
<tr><td>Cor</td><td>Altura</td><td>Comprimento</td><td>Largura</td><td>Material</td><td>Tipo</td><td>Forma</td><td>NoCadeiras</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="2">NoLugares</td></tr>
</table>

Mesmo que um campo selecionador esteja indicado com um determinado tipo, todos os campos da parte variante do registro são passíveis de serem usados. I.e., a responsabilidade de manter a coerência dos campos da parte variante é do programador.

Outro uso interessante deste tipo de estrutura é a não necessidade do campo selecionador. Neste caso, podemos ter campos que nos permitem acessar uma mesma memória de diferentes maneiras, sem precisar gastar memória com um campo selecionador.

```
type    Tipo32bits = record
                B1,B2,B3,B4 : byte;
              end;
        TipoConjuntoElementos = record
                case integer of
                1     : VetorByte: array[1..4*1000] of byte;
                2     : Vetor32Bits: array[1..1000] of Tipo32Bits;
              end;
```

Com o tipo *TipoConjuntoElementos* podemos estar usando no programa o campo *VetorByte*, mas em situações onde precisamos copiar um vetor deste tipo para outro será em muitos computadores mais rápido fazer 1000 atribuições do tipo *Tipo32Bits* do que 4000 atribuições do tipo *byte*. Nestes casos, usaríamos o campo *Vetor32Bits* em vez do vetor *VetorByte*, já que o resultado será o mesmo. Isto se deve a arquitetura do computador, que caso seja de 32 bits, pode fazer transferência de 32 bits por vez.

Exemplo 9.1 *Uma universidade mantém registros de todos os seus alunos, que inclui os alunos de graduação e de pós-graduação. Muitos tipos de dados são iguais para estas duas classes de alunos, mas alguns dados são diferentes de um para outro. Por exemplo: Todos os alunos possuem nome, data de nascimento, RG, e RA. Entretanto existem algumas diferenças para cada tipo de aluno. As disciplinas feitas pelos alunos de graduação são numéricas. Para as disciplinas de pós-graduação temos conceitos. Além disso, alunos de pós-graduação fazem tese, possuem orientador e fazem exame de qualificação. Assim, podemos ter a seguinte definição de tipo para um aluno desta universidade:*

```
type
  NivelAluno     = (Grad, Pos);
  TipoData       = record
                dia : 1..31;  mes:1..12;  ano:1900..2100;
              end;
  TipoAlunoUni     = record
                nome          : string[50];
                RG            : string[15];
                DataNascimento : TipoData;
                {... demais campos ...}
              case nivel: NivelAluno of
               Grad  :
                  (Notas: ListaNotas);
               Pos :
                  (Conceitos: ListaConceitos;
                   ExamQualif, Defesa : TipoData;
                   Titulo: string[100],
                   Orientador: string[50]);
              end;
```

A seguir apresentamos um esboço de programa que usa esta estrutura para ler os dados de um aluno desta universidade.

```
var Aluno :  TipoAlunoUni;
begin
  { ... }
  write('Entre com o nome do aluno: ');  readln(Aluno.nome);
  LeDataNascimento(Aluno.DataNascimento);
  { ... leitura dos demais dados iguais tanto para aluno de graduação como de pos ...}
  case Aluno.Nivel of
   Grad : begin
          LeNotasAlunoGraduacao(Aluno.NotasGrad);
        end;
   Pos  : begin
          LeConceitosAlunoPos(Aluno.Notas);
          LeDataQualificacao(Aluno.ExamQualif);
          LeDataDefesa(Aluno.Defesa);
          writeln('Entre com o título da tese/dissertação: '); readln(Aluno.Titulo);
          writeln('Entre com o nome do orientador: '); readln(Aluno.Orientador);
        end;
  end; { case }
  { ... }
end.
```

9.3 Comando With

A linguagem Pascal oferece uma maneira fácil para acessar os campos de um registro. Isto é feito usando-se o comando **with** que permite referenciar, no escopo do comando, os campos dos registros sem referência aos identificadores dos registros. A sintaxe do comando with é a seguinte:

with Lista_de_Variáveis_de_Registro do Comando_ou_Bloco_de_Comandos;

Obs.: na lista de variáveis não deve haver variáveis de mesmo tipo, ou variáveis que tenham campos com o mesmo nome.

Exemplo 9.2 *Vamos escrever a função MulComplex apresentada no quadro anterior usando o comando with.*

```
function MulComplex(x,y:complex):complex; { Retorna a multiplicação de dois números complexos }
var z : complex;
begin
  with z do begin
    a := x.a*y.a−x.b*y.b;
    b := x.a*y.b+x.b*y.a;
  end;
  MulComplex := z;
end; { MulComplex }
```

9.4 Exercícios

1. Declare um tipo chamado *tiporeg*, definido como um tipo de registro contendo os seguintes campos: *Nome, RG, Salario, Idade, Sexo, DataNascimento*; onde Nome e RG são strings, Salario é real, Idade é inteiro, sexo é char e DataNascimento é um registro contendo três inteiros, dia, mes e ano. Declare um tipo de registro chamado TipoCadastro que contém dois campos: Um campo, Funcionario, contendo um vetor com 100 posições do tipo *tiporeg* e outro campo inteiro, Quant, que indica a quantidade de funcionários no cadastro.
   Todos os exercícios seguintes fazem uso do tipo TipoCadastro.
2. Faça uma rotina, *InicializaCadastro*, que inicializa uma variável do tipo TipoCadastro. A rotina atribui a quantidade de funcionários como zero.
3. Faça um procedimento, *LeFuncionarios*, com parâmetro uma variável do tipo TipoCadastro. A rotina deve ler os dados de vários funcionários e colocar no vetor do cadastro, atualizando a quantidade de elementos não nulos. Caso o nome de um funcionário seja vazio, a rotina deve parar de ler novos funcionários. A rotina deve retornar com o cadastro atualizado. Lembre que o cadastro não suporta mais funcionários que os definidos no vetor de funcionários.
4. Faça uma rotina, chamada ListaFuncionários, que imprime os dados de todos os funcionários.
5. Faça duas rotinas para ordenar os funcionários no cadastro. Uma que ordena pelo nome, OrdenaNome, e outra que ordena pelo salário, OrdenaSalario.
6. Faça uma rotina, SalarioIntervalo, que tem como parâmetros: um parâmetro do tipo TipoCadastro e dois valores reais v_1 e v_2, $v_1 \leq v_2$. A rotina lista os funcionários com salário entre v_1 e v_2. Depois de imprimir os funcionários, imprime a média dos salários dos funcionários listados.
7. Faça uma rotina que dado um cadastro, imprime o nome do funcionário e o imposto que é retido na fonte. Um funcionário que recebe até R$1000,00 é isento de imposto. Para quem recebe mais que R$1000,00 e até R$2000,00 tem 10% do salário retido na fonte. Para quem recebe mais que R$2000,00 e até R$3500,00 tem 15% do salário retido na fonte. Para quem recebe mais que R$3500,00 tem 25% do salário retido na fonte.
8. Faça uma função, BuscaNome, que tem como entrada o cadastro e mais um parâmetro que é o um nome de um funcionário. O procedimento deve retornar um registro (tipo tiporeg) contendo todas as informações do funcionário que tem o mesmo nome. Caso a função não encontre um elemento no vetor contendo o mesmo nome que o dado como parâmetro, o registro deve ser retornado com nome igual a vazio.
9. Faça uma rotina, AtualizaSalario, que tem como parâmetros o cadastro de funcionários. A rotina deve ler do teclado o RG do funcionário a atualizar. Em seguida a rotina lê o novo salário do funcionário. Por fim, a rotina atualiza no cadastro o salário do funcionário com o RG especificado.
10. Faça uma função, chamada ListaMaraja, que tem como parâmetro o cadastro e devolve um registro contendo os dados de um funcionário que tem o maior salário.
11. Faça uma rotina que tem como parâmetros o cadastro e o RG de um funcionário. A rotina deve remover do cadastro o funcionário que contém o RG especificado. Lembre-se que os elementos não nulos no vetor do cadastro devem estar contíguos. Além disso, caso um elemento seja removido, a variável que indica a quantidade de elementos deve ser decrementada de uma unidade. Caso não exista nenhum elemento no vetor com o RG fornecido, a rotina não modifica nem os dados do vetor nem sua quantidade.

12. Faça uma rotina, ListaAniversarioSexo, que tem como entrada o cadastro, três inteiros: dia, mes e ano, que correspondem a uma data e um caracter (sexo) com valor 'F' ou 'M'. A rotina deve imprimir o nome dos funcionários que nasceram nesta data e com sexo igual ao definido pelo parâmetro.

10 Recursividade

Dizemos que um objeto é dito ser *recursivo* se ele for definido em termos de si próprio.

Este tipo de definição é muito usado na matemática. Um exemplo disto é a função fatorial, que pode ser definido como:

$$n! = \begin{cases} 1 & \text{se } n = 0 \text{ e} \\ n \cdot (n-1)! & \text{se } n > 0. \end{cases}$$

Existem muitos objetos que podem ser formulados de maneira recursiva. Este tipo de definição permite que possamos definir infinitos objetos de maneira simples e compacta. Podemos inclusive definir algoritmos que são recursivos, i.e., que são definidos em termos do próprio algoritmo. Nesta seção veremos como poderemos usar esta poderosa técnica como estratégia para o desenvolvimento de algoritmos.

10.1 Projeto por Indução

Os algoritmos recursivos são principalmente usados quando a estratégia de se resolver um problema pode ser feita de maneira recursiva ou quando os próprios dados já são definidos de maneira recursiva. Naturalmente exitem problemas que apresentam estas duas condições mas que não se é aconselhado usar algoritmos recursivos. Um problema que pode ser resolvido de maneira recursiva também pode ser resolvido de maneira iterativa. Algumas das principais vantagens de usar recursão são a possibilidade de se gerar programas mais compactos, programas fáceis de se entender e o uso de uma estratégia para se resolver o problema. Esta estratégia é a de atacar um problema (projetando um algoritmo) sabendo (supondo) se resolver problemas menores: *Projeto por Indução*. Note que já usamos esta idéia para desenvolver o algoritmo de busca binária.

Os objetos de programação que iremos usar para trabalhar com a recursão serão as funções e os procedimentos. Assim, uma rotina (função ou procedimento) é dita ser recursiva se ela chama a si mesma. Uma rotina recursiva pode ser de dois tipos: se uma rotina $\mathcal{R}$ faz uma chamada de si mesmo no meio de sua descrição então $\mathcal{R}$ é uma rotina recursiva direta; caso $\mathcal{R}$ não faça uma chamada a ela mesma, mas a outras rotinas que porventura podem levar a chamar a rotina $\mathcal{R}$ novamente, então $\mathcal{R}$ é uma rotina recursiva indireta.

Quando uma rotina recursiva está sendo executada, podemos visualizar uma seqüência de execuções, uma chamando a outra. Veja a figura 26.

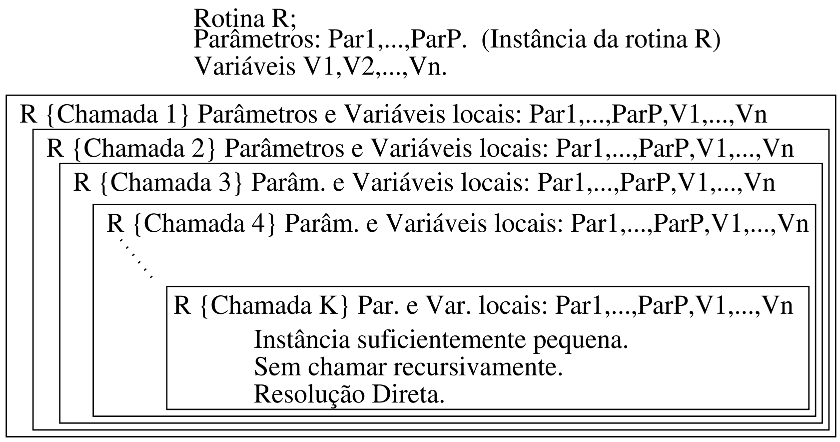
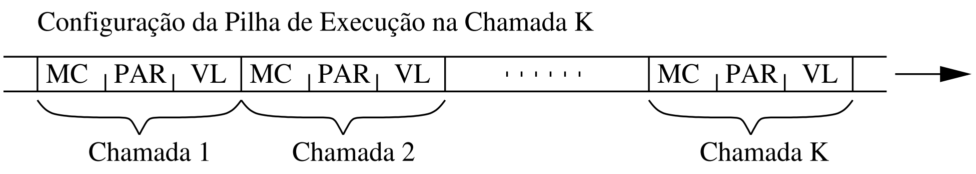


Figura 26: Seqüência das chamadas recursivas de uma rotina R até sua última chamada recursiva.

Quando em uma determinada chamada recursiva a rotina faz referência a uma variável ou parâmetro, esta está condicionada ao escopo daquela variável. Note que em cada uma das chamadas recursivas, as variáveis e parâmetros (de mesmo nome) podem ter valores diferentes. Se a rotina faz referência à variável $V1$ na terceira chamada recursiva, esta irá obter e atualizar o valor desta variável naquela chamada. A figura 27 apresenta a configuração da memória usada para a execução do programa, chamada pilha de execução, no momento em que foi feita a k-ésima chamada recursiva. A seta indica o sentido do crescimento da pilha de execução.



MC = Memória de Controle (para retorno da chamada)
PAR = Memória relativa aos parâmetros da chamada
VL = Memória relativa às variáveis locais da chamada

Figura 27: Configuração da pilha de execução após k-ésima chamada recursiva.

10.2 Garantindo número finito de chamadas recursivas

Note que na figura 26, da primeira chamada até a última, fizemos k chamadas recursivas. Em um programa recursivo é muito importante se garantir que este processo seja finito. Caso contrário, você terá um programa que fica fazendo "infinitas" chamadas recursivas (no caso até que não haja memória suficiente para a próxima chamada).

Para garantir que este processo seja finito, tenha em mente que sua rotina recursiva trabalha sobre uma instância e para cada chamada recursiva que ela irá fazer, garanta que a instância que será passada a ele seja sempre mais restrita que a da chamada superior. Além disso, garanta que esta seqüência de restrições nos leve a uma instância suficientemente simples e que permita fazer o cálculo deste caso de maneira direta, sem uso de recursão. É isto que garantirá que seu processo recursivo tem fim. Vamos chamar esta instância suficientemente simples (que a rotina recursiva resolve de maneira direta) de *base da recursão*.

Nas duas figuras a seguir apresentamos um exemplo de programa usando recursão direta e indireta.

```
program ProgramaFatorialRecursivo;

function fatorial(n : integer):integer;
begin
  if (n=0)
    then fatorial := 1
    else fatorial := n * fatorial(n−1);
end; { fatorial }

var n : integer;
begin
  write('Entre com um número: ');
  readln(n);
  writeln('O fatorial de ',n,
       ' é igual a ',fatorial(n));
end.
```

Figura 28: Fatorial recursivo (recursão direta).

```
program ProgramaParesImpares;
function impar(n : integer):boolean; forward;
function par(n      : integer):boolean;
begin
  if n=0 then par:=true
  else if n=1 then par:=false
  else par := impar(n−1);
end; { par }
function impar(n : integer):boolean;
begin
  if n=0 then impar := false
  else if n=1 then impar:=true
  else impar := par(n−1);
end; { impar }
var n : integer;
begin
  write('Entre com um inteiro positivo: ');
  readln(n);
  if (par(n)) then writeln('Número ',n,' é par.')
  else writeln('Número ',n,' é ímpar.');
end.
```

Figura 29: Funções par e impar (recursão indireta).

Estes dois exemplos são exemplos "fabricados" de rotinas recursivas, mas que tem o propósito de apresentar este tipo de rotina como uma primeira instância. Naturalmente existem soluções melhores para se implementar as funções acima. Discutiremos mais sobre isso posteriormente.

Agora vamos analisar melhor a função fatorial recursiva. Note que o parâmetro da função fatorial é um número inteiro positivo n. A cada chamada recursiva, o parâmetro que é passado é diminuido de uma unidade (o valor que é passado na próxima chamada recursiva é $(n-1)$). Assim, a instância do problema a ser resolvido (o cálculo de fatorial) vai diminuindo (ficando mais restrito) a cada chamada recursiva. E este processo deve ser finito porque existe uma condição que faz com que ele pare quando a instância ficar suficientemente pequena: caso $n = 0$.

Considere o programa da figura 29. Neste programa apresentamos um exemplo de recursão indireta. A função *par* não chama a si mesma, mas chama a função *impar* que em seguida pode chamar a função *par* novamente. A função *par* (*impar*) retorna true caso o parâmetro n seja par (ímpar).

Note que a primeira rotina que aparece no momento da compilação é a função *par*. Ela faz a chamada da função *impar*. Como esta função *impar* está definida abaixo da função *par*, a compilação do programa daria erro (por não ter sido previamente definida), caso não colocássemos uma informação dizendo a "cara" da função *impar*. Isto pode ser feito colocando antes da sua chamada uma diretiva dizendo como é a "cara" da função *impar*. Colocando apenas o cabeçalho da função ou procedimento, seguido da palavra **forward;**, estaremos dizendo ao compilador que tipo de parâmetros ela aceita e que tipo de resultado ela retorna. Com isso, poderemos fazer chamadas da rotina *impar*, mesmo que sua especificação seja feita bem depois destas chamadas.

Note também que o tamanho das instâncias das chamadas recursivas das funções *par* e *impar* também diminui de tamanho.

Projetando algoritmos por Indução

Projetar algoritmos por indução é muito parecido com o desenvolvimento de demonstrações matemáticas por indução.

Primeiro verifique se seu problema pode ser resolvido através de problemas menores do mesmo tipo que o original. Neste caso, ao projetar seu algoritmo, *assuma que você já tem rotinas para resolver os problemas menores*. Com isto em mente, construa seu algoritmo fazendo chamadas recursivas para resolver os problemas menores.

Lembrando que toda seqüência de chamadas recursivas deve parar em algum momento, construa uma condição para resolver os problemas suficientemente pequenos de maneira direta, sem fazer chamadas recursivas (base da recursão). Por fim, simule sua rotina para ver se todos os casos estão sendo cobertos.

10.3 Torres de Hanoi

Vamos ver um exemplo de programa recursivo, usando projeto indutivo, para se resolver um problema, chamado Torres de Hanoi.

Torre de Hanoi: Há um conjunto de 3 pinos: 1,2,3. Um deles tem n discos de tamanhos diferentes, sendo que os maiores estão embaixo dos menores. Os outros dois pinos estão vazios. O objetivo é mover todos os discos do primeiro pino para o terceiro pino, podendo se usar o segundo como pino auxiliar. As regras para resolver o problema são.
1. Somente um disco pode ser movido de cada vez.
2. Nenhum disco pode ser colocado sobre um disco menor que ele.
3. Observando-se a regra 2, qualquer disco pode ser movido para qualquer pino.
Escreva a ordem dos movimentos para se resolver o problema.

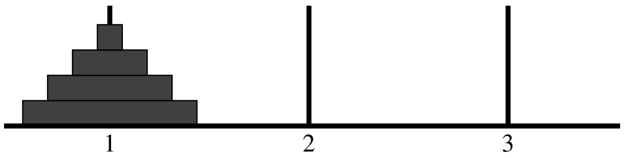


Figura 30: Torres de Hanoi com 4 discos.

Note que este problema consiste em passar os n discos de um pino de origem, para um pino destino, usando mais um pino auxiliar. Para resolver o problema das torres de hanoi com n discos, podemos usar a seguinte estratégia (veja figura 31):

1. Passar os $(n-1)$ discos menores do primeiro pino para o segundo pino.

2. Passar o maior disco para o terceiro pino.
3. Passar os $(n-1)$ discos do segundo pino para o terceiro pino.

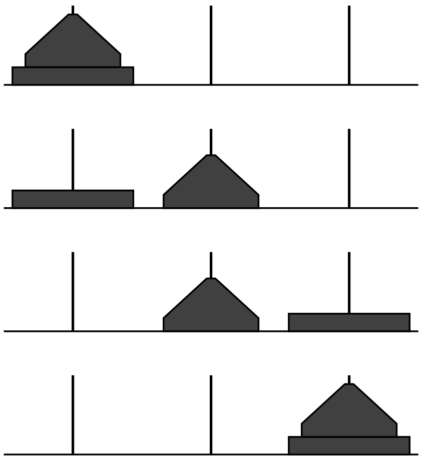


Figura 31: Estratégia do algoritmo das Torres de Hanoi.

Note que no item 1 acima, passamos $(n-1)$ discos de um pino para o outro. Aqui é importante observar que todos estes $(n-1)$ discos são menores que o maior disco que ficou no pino. Assim, podemos trabalhar com estes $(n-1)$ discos como se não existisse o disco grande. Pois mesmo que algum destes $(n-1)$ discos fique em cima do disco maior, isto não será nenhum problema, já que todos os disco em cima do maior disco são menores que este. Este mesmo raciocínio se propaga para se mover os $(n-1)$ discos em etapas.

Observe que resolvemos o problema maior, com n discos, sabendo se resolver um problema de $n-1$ discos. Além disso, para uma instância suficientemente pequena, com 0 discos, resolvemos o problema de maneira direta (base da recursão).

```
program ProgramaHanoi;
procedure Hanoi(n,origem,auxiliar,destino : integer);
begin
  if n>0 then begin
    hanoi(n−1,origem,destino,auxiliar);
    writeln('Mova de ',origem,' para ',destino);
    hanoi(n−1,auxiliar,origem,destino);
  end;
end;
var n : integer;
begin
  write('Entre com a quantidade de discos no primeiro pino: '); readln(n);
  writeln('A seqüência de movimentos para mover do pino 1 para o pino 3 é:');
  hanoi(n,1,2,3);
end.
```

10.4 Quando não usar recursão

Há momentos em que mesmo que o cálculo seja definido de forma recursiva, devemos evitar o uso de recursão. Veremos dois casos onde o uso de recursão deve ser evitado.

Chamada recursiva no início ou fim da rotina

Quando temos apenas uma chamada que ocorre logo no início ou no fim da rotina, podemos transformar a rotina em outra iterativa (sem recursão) usando um loop. De fato, nestes casos a rotina recursiva simplesmente está simulando uma rotina iterativa que usa uma estrutura de repetição. Este é o caso da função fatorial que vimos anteriormente e que pode ser transformado em uma função não recursiva como no exemplo 7.9.

O motivo de preferirmos a versão não recursiva é porque cada chamada recursiva aloca memória para as variáveis locais e parâmetros (além da memória de controle) como ilustrado na figura 27. Assim, a função fatorial recursiva chega a gastar uma memória que é proporcional ao valor n dado como parâmetro da função. Por outro lado, a função fatorial iterativa do exemplo 7.9 gasta uma quantidade pequena e constante de memória local. Assim, a versão iterativa é mais rápida e usa menos memória para sua execução.

A seguir apresentamos um outro exemplo onde isto ocorre.

Exemplo 10.1 *A função para fazer a busca binária usa uma estratégia tipicamente recursiva com uso de projeto por indução. A rotina deve encontrar um determinado elemento em um vetor ordenado. A idéia da estratégia é tomar um elemento do meio de um vetor ordenado e compará-lo com o elemento procurado. Caso seja o próprio elemento, a função retorna a posição deste elemento. Caso contrário, a função continua sua busca da mesma maneira em uma das metades do vetor. Certamente a busca na metade do vetor pode ser feita de maneira recursiva, como apresentamos na figura 32. Na figura 33 apresentamos a versão iterativa.*

Note que a base da recursão foi o vetor vazio que certamente é um vetor suficientemente pequeno para resolvermos o problema de maneira direta.

```
const MAX        = 100;
type  TipoVet = array[1..MAX] of real;

function BuscaBin(var v        : TipoVet;
                  inicio,fim : integer;
                  x          : real):integer;
var meio: integer;
begin
  if (inicio>fim) then  {vetor vazio}
    BuscaBinaria := 0  {nao achou}
  else begin
    meio := (inicio+fim) div 2;
    if (x<v[meio]) then
      BuscaBin := BuscaBin(v,inicio,meio−1,x)
    else if (x>v[meio]) then
      BuscaBin := BuscaBin(v,meio+1,fim,x)
    else BuscaBin := meio;
  end;
end; { BuscaBin }
```

Figura 32: Busca binária recursivo.

```
const MAX        = 100;
type  TipoVet = array[1..MAX] of real;

function BuscaBin(var v        : TipoVet;
                  inicio,fim : integer;
                  x          : real):integer;
var pos,i,inicio,fim,meio: integer;
begin
  pos := 0;
  while (inicio<=fim) and (pos=0) do begin
    meio := (inicio+fim) div 2;
    if (x<v[meio]) then fim := meio−1
    else if (x>v[meio]) then inicio := meio+1
    else pos:=meio;
  end;
  BuscaBin := pos;
end; { BuscaBin }
```

Figura 33: Busca binária não recursivo.

Repetição de processamento

Quando fazemos uso de recursão com várias chamadas recursivas ocorre na maioria das vezes que cada uma destas chamadas recursivas é independente uma da outra. Caso ocorram os mesmos cálculos entre duas chamadas recursivas independentes, estes cálculos serão repetidos, uma para cada chamada. Este tipo de comportamento pode provocar uma quantidade de cálculos repetidos muito grande e muitas vezes torna o programa inviável. Um exemplo claro disto ocorre na função Fibonacci, que apesar de ter sua definição de maneira recursiva, o uso de uma função recursiva causa um número exponencial de cálculos, enquanto a versão iterativa pode ser feito em tempo proporcional a n (o parâmetro da função).

$$Fibonacci(n) = \begin{cases} 0 \quad \text{se } n = 0\,, \\ 1 \quad \text{se } n = 1 \text{ e} \\ Fibonacci(n-1) + Fibonacci(n-2) \text{se } n > 1. \end{cases}$$

A função Fibonacci recursiva, apresentada a seguir, é praticamente a tradução de sua definição para Pascal.

```
function fibo(n : integer):integer;
begin
  if n=0 then fibo:=0
  else if n=1 then fibo:=1
  else fibo:=fibo(n−1)+fibo(n−2);
end;
```

Na figura 34, apresentamos as diversas chamadas recursivas da função fibonacci, descrita acima, com $n = 5$. Por simplicidade, chamamos a função Fibonacci de F.

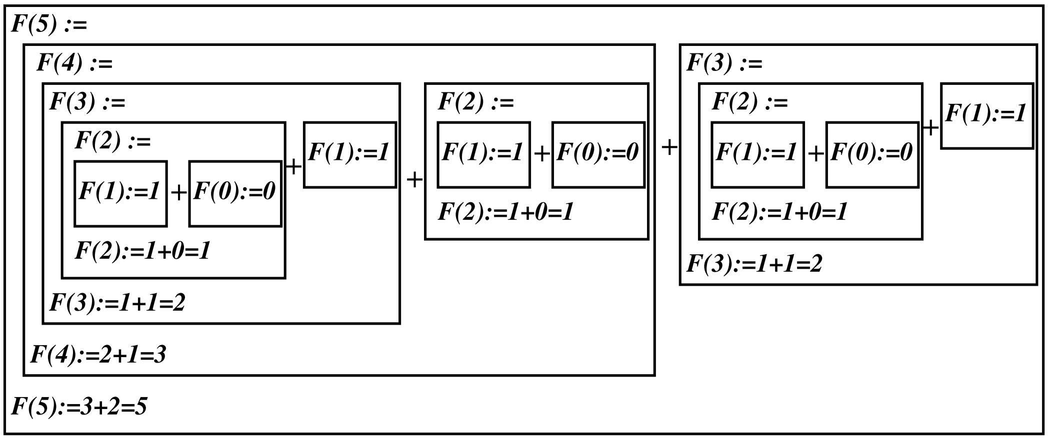


Figura 34: Chamadas recursivas feitas por Fibonacci(5).

Note que cada chamada recursiva é desenvolvida independente das chamadas recursivas anteriores. Isto provoca uma repetição dos cálculos para muitas das chamadas. Por exemplo: quando foi feita a chamada $F(3)$ (mais a direita na figura 34) para o cálculo de $F(5)$, poderíamos ter aproveitado o cálculo de $F(3)$ que tinha sido feito anteriormente para se calcular $F(4)$. Este tipo de duplicação de chamadas ocorre diversas vezes. Neste pequeno exemplo já é possível ver que a chamada de $F(1)$ ocorreu 5 vezes. De fato, uma análise mais detalhada mostraria que a quantidade de processamento feito por esta função é exponencial em n (com base um pouco menor que 2) enquanto o processamento feito pela correspondente função iterativa é linear em n.

Exercício 10.1 *Seja n e k inteiros tal que $0 \leq k < n$. Usando a identidade*

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1},$$

faça uma função recursiva que calcula $\binom{n}{k}$.
Qual a relação acima com o triângulo de Pascal ? Faça uma função que calcula o valor de $\binom{n}{k}$ aproveitando a estratégia do cálculo do triângulo de Pascal.

Exercício 10.2

1. *Uma planilha linear é formada por um vetor (com índices de 1 a MaxInd=100) onde cada elemento é chamado de célula.*
2. *Cada célula da planilha pode conter um valor real ou uma fórmula simples. Para diferenciar estas duas formas, cada célula apresenta um campo chamado Tipo que pode ser um dos seguintes caracteres: ('n','+','-','*','/').*
3. *Caso a célula tenha Tipo igual a 'n', a célula contém um outro campo chamado valor que armazena um número real.*
4. *Se Tipo tem o caracter '+' (resp. '-','*','/'), então há dois campos (Ind1 e Ind2) que contém índices do vetor e indicam que o valor deste elemento é a soma (resp. subtração, multiplicação e divisão) dos valores associados às células dos índices Ind1 e Ind2.*
   Por exemplo, a fórmula simples

   $$\boxed{+\quad 4\quad 5}$$

   indica que o valor desta célula é obtido somando-se os valores das células 4 e 5.
5. *Para uma ilustração desta planilha, veja a figura seguinte:*

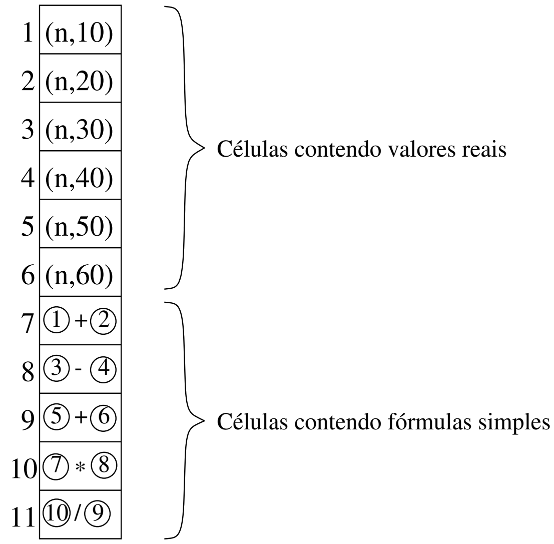


Figura 35: Planilha com valores reais e fórmulas simples.

6. *Inicialmente todos as células da planilha contém um valor real igual a zero (i.e., Tipo='n').*
7. *Para obter o valor de uma célula o programa deve proceder da seguinte maneira:*
   Se a célula contém um valor real, então este é o próprio valor da célula. Caso contrário, obtemos os valores das células indicadas em Ind1 e Ind2 e aplicamos a operação correspondente.
8. *Você pode considerar que uma célula é referenciada no máximo uma vez.*

9. *Para manipular os dados nesta planilha, você deverá fazer um programa para ler uma seqüência de linhas que pode ter um dos seguintes formatos:*

Linha lida	*Explicação do comando*
n [Ind] [Val]	*Coloca na célula [Ind] o valor [Val].*
+ *[Ind] [Ind1] [Ind2]*	*O valor da célula [Ind] é o valor da célula [Ind1] somado ao valor da célula [Ind2].*
- *[Ind] [Ind1] [Ind2]*	*O valor da célula [Ind] é o valor da célula [Ind1] subtraido do valor da célula [Ind2].*
$\star$ *[Ind] [Ind1] [Ind2]*	*O valor da célula [Ind] é o valor da célula [Ind1] multiplicado com o valor da célula [Ind2].*
/ *[Ind] [Ind1] [Ind2]*	*O valor da célula [Ind] é o valor da célula [Ind1] dividido pelo valor da célula [Ind2].*
p [Ind]	*Imprime na tela o valor da célula [Ind]. Este comando não altera a planilha.*
.	*Este comando (.) finaliza o programa.*

10. *Exemplo: Vamos considerar que queremos avaliar o valor da fórmula* $((A + B) * (C - D))/(E + F)$*, para diferentes valores de* A, B, C, D, E, F*. Podemos considerar o valor de* A *na célula 1, o valor de* B *na célula 2, ..., o valor de* F *na célula 6. Com isso, podemos montar nossa expressão com os seguintes comandos:*

Linhas de entrada				*Comentários*	
n	*1*	*10*		$[1] \leftarrow 10$	$A \leftarrow 10$
n	*2*	*20*		$[2] \leftarrow 20$	$B \leftarrow 20$
n	*3*	*30*		$[3] \leftarrow 30$	$C \leftarrow 30$
n	*4*	*40*		$[4] \leftarrow 40$	$D \leftarrow 40$
n	*5*	*50*		$[5] \leftarrow 50$	$E \leftarrow 50$
n	*6*	*60*		$[6] \leftarrow 60$	$F \leftarrow 60$
+	*7*	*1*	*2*	$[7] \leftarrow ([1] + [2])$	$[7] \leftarrow (A + B)$
-	*8*	*3*	*4*	$[8] \leftarrow ([3] - [4])$	$[8] \leftarrow (C - F)$
+	*9*	*5*	*6*	$[9] \leftarrow ([5] + [6])$	$[9] \leftarrow (E + F)$
$\star$	*10*	*7*	*8*	$[10] \leftarrow [7] * [8]$	$[10] \leftarrow (A + B) * (C - D)$
/	*11*	*10*	*9*	$[11] \leftarrow [10]/[9]$	$[11] \leftarrow ((A + B) * (C - D))/(E + F)$
				Neste ponto a planilha apresenta a configuração da figura 35	
p	*11*			*Imprime o cálculo de* $((10 + 20) * (30 - 40))/(50 + 60)$	*Imprime -2.73*
n	*1*	*1*		$[1] \leftarrow 1$	$A \leftarrow 1$
n	*5*	*5*		$[5] \leftarrow 5$	$E \leftarrow 5$
p	*11*			*Imprime o cálculo de* $((1 + 20) * (30 - 40))/(5 + 60)$	*Imprime -3.23*
.				*Fim do processamento*	

Exercício adicional 1: *Considere agora que cada célula pode ser referenciada mais de uma vez, mas a obtenção do valor de cada célula não deve fazer uso dela mesma. Caso isto ocorra, dizemos que a planilha está inconsistente. Faça uma rotina que verifica se uma planilha está inconsistente.*

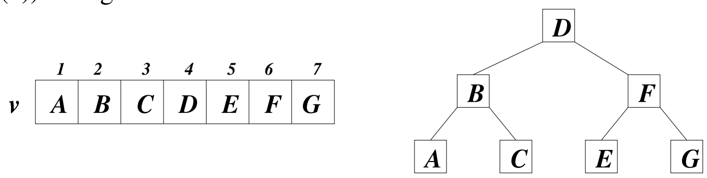
Exercício adicional 2: *Considere agora uma planilha bidimensional, conde cada célula deve ser especificada por dois índices. Faça o mesmo programa, com as devidas modificações, para trabalhar com este tipo de planilha.*

10.5 Exercícios

1. Faça uma função recursiva para encontrar um elemento em um vetor. A rotina deve ter como parâmetros: o vetor, o número de elementos do vetor (o primeiro elemento do vetor começa no índice 1), e um valor a ser procurado. A rotina retorna o índice do elemento no vetor, caso este se encontre no vetor, -1 caso contrário.
2. Faça uma rotina recursiva para imprimir os elementos de um vetor, na ordem do menor índice primeiro.
3. Faça uma rotina recursiva para imprimir os elementos de um vetor, na ordem do maior índice primeiro.
4. A função de Acherman é definida recursivamente nos números não negativos como segue:

$$\begin{array}{ll} a(m,n) = n+1 & \text{Se } m = 0, \\ a(m,n) = a(m-1,1) & \text{Se } m \neq 0 \text{ e } n = 0, \\ a(m,n) = a(m-1, a(m, n-1)) & \text{Se } m \neq 0 \text{ e } n \neq 0. \end{array}$$

   Faça um procedimento recursivo para computar a função de Ackerman. Obs.: Esta função cresce muito rápido, assim ela deve poder ser impressa para valores pequenos de m e n.
5. Faça uma rotina recursiva para imprimir todas as permutações dos números de 1 a n, uma permutação por linha.
6. Dado uma cadeia de caracteres de comprimento n, escreva uma rotina recursiva que inverte a seqüência dos caracteres. I.e., o primeiro caracter será o último e o último será o primeiro.
7. Um vetor tem $2^k - 1$ valores inteiros (figura (a)), onde k é um inteiro positivo, $k \geq 1$. Este vetor representa uma figura hierárquica (figura (b)) da seguinte maneira:



(a) *(b)*

   Você pode imaginar que este vetor está representando uma árvore genealógica de 3 níveis. Infelizmente, o usuário do programa que faz uso deste vetor necessita de algo mais amigável para ver esta estrutura. Faça uma rotina recursiva que dado este vetor v e o valor k, imprime as seguintes linhas:

```
          G-----------
     F----------------
          E-----------
D---------------------
          C-----------
     B----------------
          A-----------
```

   Note que fica bem mais fácil para enxergar a hierarquia visualizando este desenho. A profundiade que é impresso cada elemento é feita controlando os espaços definidos para a profundidade de elemento na hierarquia.
8. Escreva uma função recursiva para calcular o *máximo divisor comum*, **mdc**, de dois inteiros positivos da seguinte maneira:

$$\begin{array}{ll} \mathbf{mdc}(x,y) = y & \textit{se } (y \leq x) \textit{ e } x \textit{ mod } y = 0; \\ \mathbf{mdc}(x,y) = mdc(y,x) & \textit{se } (x < y); \\ \mathbf{mdc}(x,y) = mdc(y, x \ \mathbf{mod}\ y) & \textit{caso contrário.} \end{array}$$

9. Cálculo de determinantes por co-fatores. Seja A uma matriz quadrada de ordem n. O *Menor Complementar* M_{ij}, de um elemento a_{ij} da matriz A é definido como o determinante da matriz quadrada de ordem $(n-1)$ obtida a partir da matriz A, excluindo os elementos da linha i e da coluna j. O *Co-Fator* α_{ij} de A é definido como:

$$\alpha_{ij} = (-1)^{i+j} M_{ij} \ .$$

   O determinante de uma matriz quadrada A de ordem n pode ser calculado usando os co-fatores da linha i da seguinte maneira:

$$\mathbf{det}(A) = \alpha_{i1}A_{i1} + \alpha_{i2}A_{i2} + \cdots + \alpha_{in}A_{in}.$$

O mesmo cálculo pode ser feito pelos co-fatores da coluna j da seguinte maneira:

$$\mathbf{det}(A) = \alpha_{1j}A_{1j} + \alpha_{2j}A_{2j} + \cdots + \alpha_{nj}A_{nj}.$$

Faça uma rotina recursiva para calcular o determinante de uma matriz de ordem n usando o método descrito acima, onde a rotina tem o seguinte cabeçalho:

function *determinante*(**var** *A:TipoMatrizReal; n:***integer**):**real**;

onde TipoMatrizReal é um tipo adequado para definir matriz e n é a ordem da matriz.

Obs.: Existem na literatura outros métodos mais eficientes para se calcular o determinante.

10. Seja $v = (v_1, \ldots, v_i, \ldots, v_f, \ldots, v_n)$ um vetor com n dígitos entre 0 e 9. Faça uma rotina recursiva para verificar se os elementos $(v_i, \ldots, v_f)$ formam um número palíndrome, $1 \leq i \leq f \leq n$. Obs.: Pode considerar que os números podem começar com alguns dígitos 0's, assim, o número 0012100 é palíndrome.

11 Algoritmos de Ordenação

Nesta seção vamos considerar outros algoritmos para ordenação: *InsertionSort, MergeSort* e *QuickSort*. Apresentaremos os algoritmos MergeSort e QuickSort implementados recursivamente e utilizando uma importante técnica chamada *Divisão e Conquista*. Estes dois algoritmos estão entre os algoritmos mais rápidos para ordenação usando apenas comparação entre elementos. Neste tipo de ordenação, o algoritmo QuickSort é o que tem o melhor tempo médio para se ordenar seqüências em geral.

11.1 Algoritmo InsertionSort

O algoritmo InsertionSort também usa a técnica de projeto de algoritmo por indução para resolver o problema, i.e., supondo saber resolver um problema pequeno, resolve se um maior.

Considere um vetor $v = (v_1, v_2, \ldots, v_{n-1}, v_n)$. Vamos supor que já sabemos ordenar o vetor $v' = (v_1, v_2, \ldots, v_{n-1})$ (com $n-1$ elementos, e portanto menor que n). A idéia é ordenar o vetor v' e depois inserir o elemento v_n na posição correta (daí o nome InsertionSort). Primeiramente apresentamos um algoritmo recursivo que implementa esta idéia.

```
procedure InsertionSortRecursivo(var v : TipoVetorReal;n:integer);
var i  : integer;
  aux : real;
begin
  if n>1 then begin
    InsertionSortRecursivo(v,n−1); {Ordena os n−1 primeiros elementos}
    aux:=v[n];  i:=n;
    while (i>1) and (v[i−1]>aux) do begin
      v[i] := v[i−1];
      i:=i−1;
    end;
    v[i] := aux;
  end;
end; { InsertionSortRecursivo }
```

O algoritmo tem uma estratégia indutiva e fica bem simples implementá-lo de maneira recursiva. Note que a base da recursão é para o vetor com no máximo 1 elemento (nestes caso o vetor já está ordenado e portanto o problema para este vetor já está resolvido). Uma vez que desenvolvemos a estratégia, podemos observar que podemos descrevê-lo melhor de forma iterativa, uma vez que o algoritmo acima faz uma chamada recursiva no início da rotina (veja seção 10.4). Assim, uma estratégia mais eficiente, usando duas estruturas de repetição, é apresentada a seguir.

```
procedure InsertionSort(var v : TipoVetorReal;n:integer);
var i,j : integer;
  aux : real;
begin
  for i:=2 to n do begin
    aux:=v[i];  j:=i;
    while (j>1) and (v[j−1]>aux) do begin
      v[j] := v[j−1];
      j:=j−1;
    end;
    v[j] := aux;
  end;
end; { InsertionSort }
```

No caso médio este algoritmo também gasta um tempo computacional quadrático em relação a quantidade de elementos. Por outro lado, se a instância estiver *quase ordenada*, este algoritmo é bastante rápido.

11.2 Algoritmo MergeSort e Projeto por Divisão e Conquista

O algoritmo MergeSort também usa a estratégia por indução. Além disso, o algoritmo usa de outra técnica bastante importante chamada Divisão e Conquista. Neste tipo de projeto, temos as seguintes etapas:

- Problema suficientemente pequeno: Resolvido de forma direta.
- Problema não é suficientemente pequeno:

  Divisão: O problema é dividido em problemas menores.
  Conquista: Cada problema menor é resolvido (recursivamente).
  Combinar: A solução dos problemas menores é combinada de forma a construir a solução do problema.

No algoritmo MergeSort, a etapa de divisão consiste em dividir o vetor, com n elementos, em dois vetores (subvetores) de mesmo tamanho ou diferindo de no máximo um elemento. No caso, um vetor $v = (v_1, \ldots, v_n)$ é dividido em vetores $v' = (v_1, \ldots, v_{\lfloor n/2 \rfloor})$ e $v'' = (v_{\lceil n/2 \rceil}, \ldots, v_n)$.

Para facilitar, em vez de realmente dividir o vetor em dois outros vetores, usaremos índices para indicar o início e o fim de um subvetor no vetor original. Assim, estaremos sempre trabalhando com o mesmo vetor, mas o subvetor a ser ordenado em cada chamada recursiva será definido através destes índices. Caso o vetor tenha no máximo 1 elemento, o vetor já está ordenado e não precisamos subdividir em partes menores (base da recursão).

Uma vez que o vetor (problema) foi dividido em dois subvetores (problemas menores), vamos ordenar (conquistar) cada subvetor (subproblema) recursivamente.

A combinação das soluções dos subproblemas é feita intercalando os dois subvetores ordenados em apenas um vetor. Para isso, usaremos dois índices para percorrer os dois subvetores e um terceiro para percorrer o vetor que receberá os dois vetores intercalados, vetor resultante. A idéia é começar os índices no início dos dois vetores e comparar os dois elementos localizados por estes índices e atribuir no vetor resultante o menor valor. Em seguida, incrementamos o índice que tinha o menor valor. Este processo se repete até que tenhamos intercalado os dois vetores no vetor resultante.

No quadro seguinte apresentamos o procedimento para intercalar os vetores $V[inicio, \ldots, meio]$ e $V[meio + 1, \ldots, fim]$ e na figura 36 ilustramos seu comportamento.

```
procedure IntercalaMergeSort(var v : TipoVetorReal; Inicio,Meio,Fim:integer);
  var i,j,k : integer; {v1 = [Inicio..Meio], v2 = [Meio+1..Fim]}
  begin { supondo a declaração de Vaux no procedimento principal }
    i:=Inicio;        j:=Meio+1;        k:=Inicio;
    while (i<=Meio) and (j<=Fim) do begin
      if (v[i]>v[j]) then begin {Sempre inserindo o menor em Vaux}
        vaux[k]:=v[j];   j:=j+1;
      end else begin
        vaux[k]:=v[i];   i:=i+1;
      end;
      k:=k+1;
    end;
    while (i<=Meio) do begin
      vaux[k]:=v[i]; {Inserindo os elementos do primeiro vetor}
      i:=i+1;  k:=k+1;
    end;
    while (j<=Fim) do begin
      vaux[k]:=v[j]; {Inserindo os elementos do segundo  vetor}
      j:=j+1;  k:=k+1;
    end;
    for k:=Inicio to Fim do v[k] := vaux[k]; {Copiando para o vetor original}
  end; { IntercalaMergeSort }
```

A seguir, descrevemos o algoritmo MergeSort.

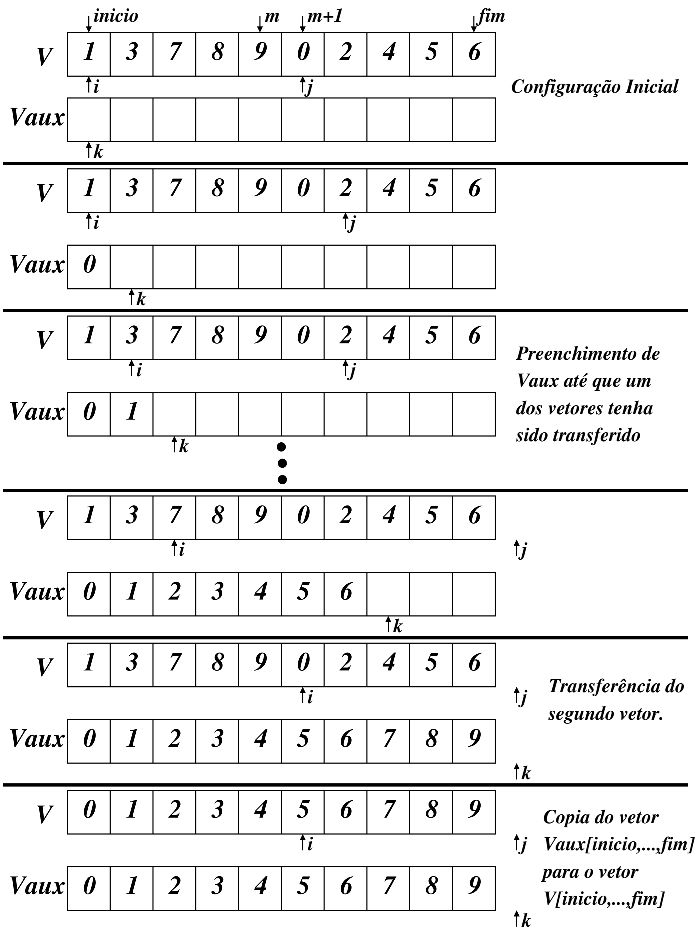


Figura 36: Intercalação de dois sub-vetores.

```
procedure MergeSort(var V : TipoVetorReal; n:integer);
var Vaux : TipoVetorReal;  {vetor auxiliar para fazer intercalacoes}

{ --> Inserir rotina Intercala IntercalaMergeSort <-- }

procedure MergeSortRecursivo(var V : TipoVetorReal; inicio,fim:integer);
  var meio : integer;
  begin
    if (fim > inicio) then begin {Se tiver quantidade suficiente de elementos}
      meio:=(fim+inicio) div 2;          {Dividindo o problema}
      MergeSortRecursivo(V,inicio,meio); {Conquistando subproblema 1}
      MergeSortRecursivo(V,meio+1,fim);  {Conquistando subproblema 2}
      IntercalaMergeSort(V,inicio,meio,fim); {Combinando subproblemas 1 e 2}
    end;
  end; { MergeSortRecursivo }

begin
  MergeSortRecursivo(V,1,n);
end; { MergeSort }
```

Na figura 37 apresentamos uma simulação do algoritmo MergeSort para o vetor $\boxed{7,3,8,1,4,2,6,5}$. Denotamos a chamado da rotina *MergeSort(V)* por *M(V)*, e a rotina IntercalaMergeSort por *I*(v_1, v_2). Se uma chamada da rotina $M(V)$ exige chamadas recursivas, esta é substituída pelas chamadas $I(\ M(V')\ ,\ M(V'')\ )$, onde V' e V'' são as duas partes de V.

```
M(        7 ,  3 ,   8 ,   1 ,  4 ,  2 ,  6 ,  5)
I( M(     7 ,  3 ,   8 ,   1)  ,M(4 ,  2 ,  6 ,  5) )
I( I( M(  7 ,  3)  ,M( 8 ,   1)) ,M(4 ,  2 ,  6 ,  5) )
I( I( I(M(7)  ,M(3))  ,M( 8 ,   1)) ,M(4 ,  2 ,  6 ,  5) )
I( I( I(   7  ,M(3) ) ,M( 8 ,   1)) ,M(4 ,  2 ,  6 ,  5) )
I( I( I(   7 ,   3)  ,M( 8 ,   1) ) ,M(4 ,  2 ,  6 ,  5) )
I( I(    (3 ,   7)  ,M( 8 ,   1) ) ,M(4 ,  2 ,  6 ,  5) )
I( I(    (3 ,   7)  ,I(M(8)  , M(1))) ,M(4 ,  2 ,  6 ,  5) )
I(  I(   (3 ,   7)  ,I(  8   , M(1))) ,M(4 ,  2 ,  6 ,  5) )
I(  I(   (3 ,   7)  ,I(  8 ,   1)) ,M(4 ,  2 ,  6 ,  5) )
I( I(    (3 ,   7) ,  (1 ,   8)) ,M(4 ,  2 ,  6 ,  5) )
I(       (1 ,  3 ,   7 ,   8)  ,M(4 ,  2 ,  6 ,  5) )
                      ⋮
I(       (1 ,  3 ,   7 ,   8) , (2 ,  4 ,  5 ,  6) )
(         1 ,  2 ,   3 ,   4 ,  5 ,  6 ,  7 ,  8)
```

Figura 37: Simulação das chamadas recursivas do procedimento MergeSort para o vetor $\boxed{7,3,8,1,4,2,6,5}$.

11.3 Algoritmo QuickSort

O algoritmo QuickSort também usa a técnica de divisão e conquista, dividindo cada problema inicial em duas partes. Neste caso, a etapa de divisão em dois subproblemas é mais sofisticada mas por outro lado a etapa de combinação é simples (lembrando que no algoritmo MergeSort, dividir é simples enquanto combinar é mais sofisticado).

Para dividir um vetor $v = (v_1, \ldots, v_n)$ em dois subvetores $v' = (v'_1, \ldots, v'_p)$ e $v'' = (v''_1, \ldots, v''_q)$, o algoritmo usa de um pivo X tal que $v'_i \leq X, i = 1, \ldots, p$ e $X < v''_j, j = 1, \ldots, q$.

Uma vez que o vetor v foi dividido nos subvetores v' e v'', estes são ordenados recursivamente (conquistados).

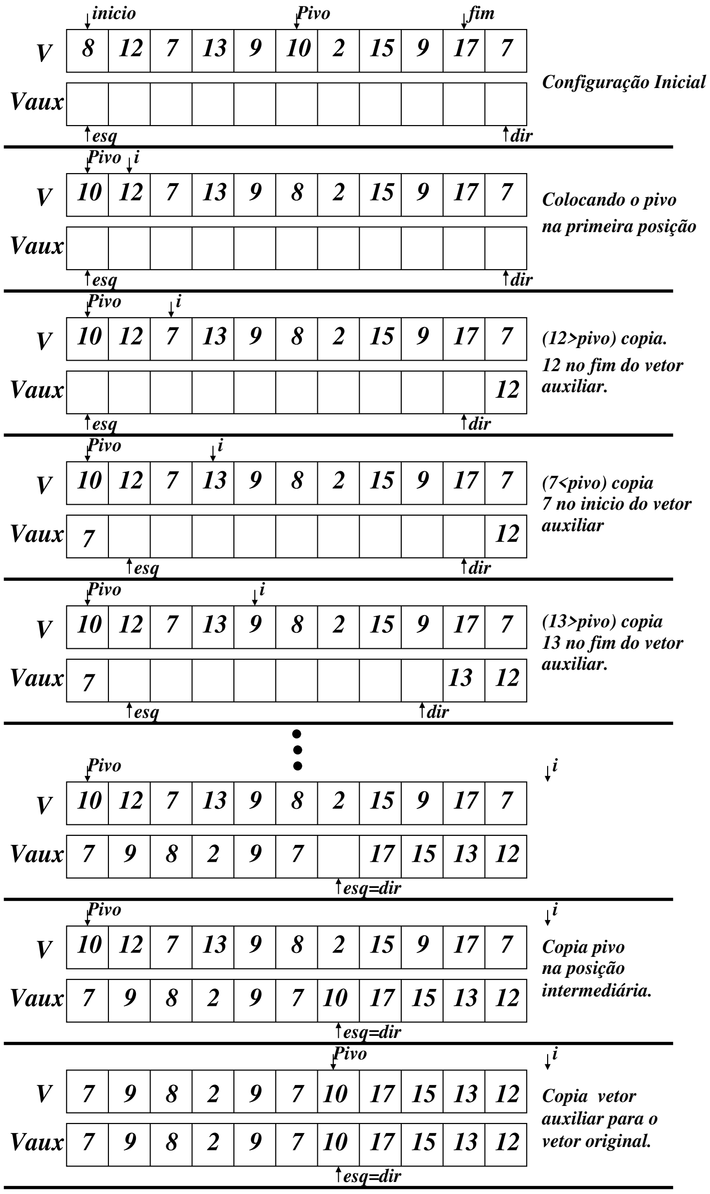
Por fim, os vetores são combinados, que nada mais é que a concatenação dos vetores ordenados v' e v''.

A etapa mais complicada neste algoritmo é a etapa de divisão, que pode ser feita eficientemente no mesmo vetor $v = (v_1, \ldots, v_n)$. Primeiro escolhemos um pivo X. Uma das vantagens de se escolher este pivo do próprio vetor a ser ordenado é que podemos implementar a estratégia garantindo sempre que cada subvetor é estritamente menor que o vetor original. O que faremos é tomar o pivo $X \in v$ e dividir o vetor v em subvetores v' e v'' de tal maneira que a ordenação final se torne o vetor $v' \| (X) \| v''$, onde o operador $\|$ é a concatenação de vetores.

Antes de apresentar o algoritmo de particionamento em um vetor, vamos apresentá-lo usando um vetor auxiliar. A implementação é mais simples e fica mais fácil de se entender a idéia do programa. Primeiro a rotina escolhe um pivo (posição $(\mathit{IndPivo} \leftarrow \lfloor \mathit{inicio} + \mathit{fim})/2 \rfloor)$. Troca com o primeiro elemento ($V[\mathit{inicio}]$). Em seguida percorre-se todos os demais elementos $V[\mathit{inicio} + 1..\mathit{fim}]$ e insere os elementos menores ou iguais ao pivo, lado a lado, no início do vetor auxiliar. Os elementos maiores que o pivo são inseridos lado a lado a partir do fim do vetor auxiliar. Por fim, o pivo é inserido na posição intermediária e o vetor auxiliar é copiado para o vetor auxiliar.

```
function ParticionaQuickSort(var v : TipoVetorReal;inicio,fim:integer):integer;
var esq,dir,IndicePivo:integer;      Pivo:real;
  { Supondo a declaração de Vaux (vetor auxiliar p/ partição) no procedimento principal}
begin
  troca(v[inicio],v[(inicio+fim) div 2]);{escolha de um pivo}
  Pivo := v[Inicio]; {Pivo fica na posição Inicio }
  esq:=Inicio;     dir:=Fim;
  for i:=Inicio+1 to fim do begin
    if (V[i] > Pivo) then begin
      Vaux[dir] := V[i];          dec(dir);
    end else begin
      Vaux[esq] := V[i];          inc(esq);
    end;
  end;
  V[esq] := Pivo; {esq=dir}
  for i:=Inicio to fim do V[i] := Vaux[i]; {Copia Vaux[inicio..fim] para V[inicio..fim]}
  ParticionaQuickSort := esq;
end;
```

No quadro seguinte mostramos comportamento deste algoritmo.



Agora vamos descrever o procedimento para particionar um vetor, através de um pivo e sem uso de vetor auxiliar. Primeiramente colocaremos o pivo na primeira posição do vetor. Para obter os subvetores v' e v'', usaremos de dois índices no vetor original, *esq* e *dir*. Inicialmente *esq* começa no segundo elemento do vetor (logo depois do pivo) e *dir* começa no último elemento do vetor. Em seguida, fazemos o índice *esq* "andar" para a direita enquanto o índice apontar para um elemento menor ou igual ao pivo. Fazemos o mesmo com o índice *dir*, "andando-o" para a esquerda enquanto o elemento apontado por ele for maior que o pivo. Isto fará com que o índice *esq* vá pulando todos os elementos que vão pertencer ao vetor v' e o índice *dir* pula todos os elementos que vão pertencer ao vetor v''. Quando os dois índices pararem, estes vão estar parados em elementos que não são os pertencentes aos correspondentes vetores e neste caso, trocamos os dois elementos e continuamos o processo. Para manter a separação entre v' e v'' viável, vamos fazer este processo sempre enquanto $esq < dir$. Quando todo este processo terminar (i.e., quando $esq \geq dir$), iremos inserir o Pivo (que estava na primeira posição) separando os dois vetores. O algoritmo QuickSort está descrito na página 115. Existem outras implementações do algoritmo QuickSort que são mais eficientes que a versão que apresentamos, embora são também um pouco mais complicadas.

```
function ParticionaQuickSort(var v : TipoVetorReal;inicio,fim:integer):integer;
var esq,dir,IndicePivo : integer;
  Pivo                 : real;
begin
  troca(v[inicio],v[(inicio+fim) div 2]);{escolha de um pivo}
  Pivo := v[Inicio]; {Pivo fica na posição Inicio }
  esq:=Inicio+1; {primeiro elemento mais a esquerda, pulando o pivo}
  dir:=Fim;    {primeiro elemento mais a direita}
  while (esq<dir) do begin
    while (esq<dir) and (v[esq] <= Pivo) do inc(esq);
    while (esq<dir) and (pivo <  v[dir]) do dec(dir);
    if esq<dir then begin
      troca(V[esq],V[dir]);
      inc(esq);      dec(dir);
    end;
  end; {dir <= esq, e |esq−dir|<=1}
  if v[dir]<=Pivo then IndicePivo := dir {dir=esq−1 ou dir=esq}
  else IndicePivo := dir−1;           {esq=dir}
  troca(v[IndicePivo],v[Inicio]); {Coloca o pivo separando os dois subvetores}
  ParticionaQuickSort := IndicePivo;
end;
```

A seguir, apresentamos a descrição do algoritmo QuickSort.

```
procedure QuickSort(var V : TipoVetorReal; n:integer);
{ −−> Declarar Vaux e inserir rotina Intercala ParticionaQuickSort <−− }
procedure QuickSortRecursivo(var V :TipoVetorReal; inicio,fim:integer);
  var IndicePivo : integer;
  begin
    if (fim > inicio) then begin {se tem pelo menos 2 elementos}
      IndicePivo := ParticionaQuickSort(v,Inicio,Fim); {Dividindo o problema}
      QuickSortRecursivo(V,Inicio,IndicePivo−1); {Conquistando subproblema 1}
      QuickSortRecursivo(V,IndicePivo+1,Fim);   {Conquistando subproblema 2}
    end;      {Não é preciso fazer nada para combinar os dois subproblemas}
  end;
begin
  QuickSortRecursivo(V,1,n);
end;
```

11.4 Exercícios

1. Faça uma rotina recursiva para ordenar um vetor usando a estratégia do algoritmo SelectionSort.
2. Faça um algoritmo para ordenar um vetor usando uma estratégia parecida com a usada pelo algoritmo MergeSort só que em vez de dividir em duas partes, divide em três partes. (I.e., se o vetor for suficientemente pequeno, ordena-se de forma direta. Caso contrário este algoritmo divide o vetor em 3 partes iguais (ou diferindo de no máximo um elemento). Ordena recursivamente cada um dos três subvetores e por fim o algoritmo intercala os três vetores, obtendo o vetor ordenado.)
3. Seja $(v_i, \ldots, v_f)$ elementos consecutivos de um vetor v do índice i ao índice f. Projete um procedimento recursivo com o cabeçalho

   procedure minmax(**var** v: *TipoVetorReal*; i,f: **integer**; var minimo, maximo: **real**);

   que retorna em *minimo* e *maximo* o menor e o maior valor que aparecem no vetor v dos índices i ao índice f, respectivamente. O procedimento deve dividir o vetor em dois subvetores de tamanhos quase iguais (diferindo de no máximo 1 elemento).
4. Faça uma rotina recursiva para *bagunçar* os elementos de um vetor (com índices de 1 a n). Use a função *Random* para gerar um número i entre 1 e n e troque o n-ésimo elemento com o i-ésimo elemento. Repita recursivamente o mesmo processo para os elementos de 1 a $n-1$.
5. *(K-ésimo)* O problema do K-ésimo consiste em, dado um vetor com n elementos, encontrar o k-ésimo menor elemento do vetor. Note que se o vetor já estiver ordenado, podemos resolver este problema obtendo o elemento que está na posição k do vetor. Isto nos dá um algoritmo que usa a ordenação para resolver o problema do k-ésimo. Entretanto, é possível se resolver este problema de maneira mais eficiente, no caso médio. A idéia é combinar duas estratégias:

   (a) Estratégia da busca binária, de descartar uma parte do vetor.
   (b) Estratégia do algoritmo *ParticionaQuickSort*, que divide o vetor V em três partes, $V = (V' \| (X) \| V'')$, onde V' contém os elementos menores ou iguais a X e V'' contém os elementos maiores que X.

   Use estas duas estratégias para desenvolver uma função que dado V, n e k, devolve o k-ésimo menor elemento do vetor.
6. Faça um programa contendo procedimentos com os seguintes cabeçalhos:

   - **procedure** MergeSort(**var** v:*TipoVetor*; n:**integer**);
   - **procedure** QuickSort(**var** v:*TipoVetor*; n:**integer**);
   - **procedure** SelectionSort(**var** v:*TipoVetor*; n:**integer**);
   - **procedure** InsertionSort(**var** v:*TipoVetor*; n:**integer**);
   - **procedure** aleatorio(**var** v:*TipoVetor*; n:**integer**);

   onde *TipoVetor* é um vetor de números:

   type TipoVetor=**array** [1..30000] of **integer**;

   As rotinas MergeSort, QuickSort, InsertionSort e SelectionSort são rotinas para ordenar usando os algoritmos do mesmo nome. A rotina aleatorio coloca no vetor v a quantidade de n elementos gerados (pseudo) aleatoriamente pela função random(MAXINT).

   O programa deve ter rotinas auxiliares para leitura e impressão de vetores. O programa deve ter um menu com opcoes:

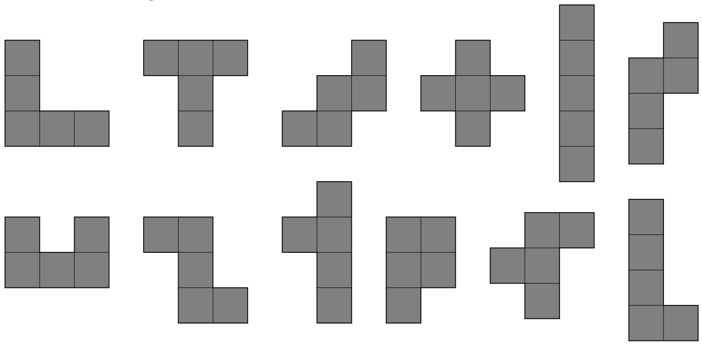
   (a) Ler um vetor com n inteiros (n também é lido). Inicialmente $n = 0$.
   (b) Gerar vetor aleatório com n elementos (n é lido).
   (c) Ordenar por MergeSort.
   (d) Ordenar por QuickSort.
   (e) Ordenar por InsertionSort.
   (f) Ordenar por SelectionSort.
   (g) Imprimir o vetor.
   (h) Sair do programa.

   O aluno deve cronometrar os tempos gastos pelas rotinas de ordenação para ordenar n inteiros (pseudo) aleatórios, onde n pode ser $n = 1000, 5000, 10000, 15000, 20000, 25000, 30000$ (ou o quanto o computador suportar). Caso algum método seja muito lento, ignore sua execução. Não é necessário entregar estes tempos.

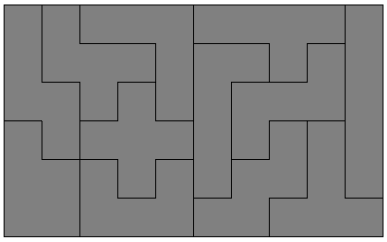
7. *(Passeio do Cavalo)* Considere um tabuleiro retangular com $n \times m$ casas. O cavalo é uma peça que "anda" neste tabuleiro, inicialmente em uma posição específica. O cavalo pode andar duas casas em um dos sentidos, horizontalmente (verticalmente), e uma casa verticalmente (horizontalmente). Dado um tabuleiro $n \times m$, e uma posição inicial do cavalo (i, j), $1 \leq i \leq n$ e $1 \leq j \leq m$, faça um programa que resolve os seguintes itens:

   (a) Diga se o cavalo consegue fazer um passeio passando por todas as $n \cdot m$ casas do tabuleiro.
   (b) Dado uma posição (a, b), $1 \leq a \leq n$ e $1 \leq b \leq m$, diga se existe um passeio da casa de origem (i, j) até a casa (a, b). Em caso positivo, imprima uma seqüência de movimentos que o cavalo precisa fazer para ir de (i, j) para (a, b).

8. *(Problema das 8 Rainhas)* O problema das 8 rainhas consiste em encontrar uma disposição de 8 rainhas em um tabuleiro de xadrez. Uma rainha pode atacar qualquer outra que esteja na mesma linha, coluna ou diagonal do tabuleiro. Portanto o objetivo é dispor as 8 rainhas de maneira que qualquer duas delas não se ataquem. Faça um programa que encontra a solução para este problema, com n rainhas em um tabuleiro $n \times n$.
9. *(Pentominos)* Um Pentomino é uma figura formada juntando 5 quadrados iguais pelos lados. As 12 figuras possíveis, exceto por rotações, são as seguintes:



Dado um retângulo $n \times m$, o objetivo é preencher este retângulo com os pentominos $p_1, \ldots, p_{12}$ apresentados acima, sem sobreposição entre os pentominos e sem que um pentomino ultrapasse a borda do retângulo. Por exemplo, a figura seguinte mostra um retângulo 6×10 preenchida com pentominos. Além disso, um pentomino pode ser girado antes de ser encaixado no retângulo. Considere que você pode usar até n_i pentominos do tipo p_i, $i = 1, \ldots, 12$, onde cada n_i é lido.



Faça um programa recursivo para resolver este problemas, dado um retângulo $n \times m$.

12 Passagem de funções e procedimentos como parâmetros

A linguagem pascal permite passar rotinas como parâmetros. Isto nos possibilita fazer rotinas focalizadas em um método de forma bem mais genérica usando rotinas particulares para cada tipo de dado, que são passadas como parâmetros.

Mas antes de apresentar a sintaxe destas declarações de parâmetros, precisamos fazer algumas observações entre dois dos principais padrões da linguagem Pascal: o do Pascal Extendido e o Borland Pascal. Existem algumas diferenças entre estes dois padrões e a passagem de funções e procedimentos como parâmetros é uma delas.

12.1 Diferenças entre Borland/Turbo Pascal e Extended Pascal

Borland Pascal O padrão Borland Pascal apresenta declaração de novos tipos associados a procedimentos ou funções, mas não aceita a descrição do tipo do procedimento ou função na declaração do parâmetro. Assim, para declarar um parâmetro que é um procedimento ou função precisamos primeiro definir tipos associados a estas rotinas e só depois declarar as rotinas como parâmetros usando estes tipos.

type
  IdentificadorTipoFuncao = **function**(*Lista_de_Parâmetros*): Tipo_Retorno_Função;
  IdentificadorTipoProcedure = **procedure**(*Lista_de_Parâmetros*);

O tipo da rotina é igual a declaração do cabeçalho da rotina, tirando o identificador da rotina. Com isso, podemos declarar um parâmetro que é uma rotina como um parâmetro normal.

Obs.: No Turbo Pascal é necessário colocar a opção de funções e procedimentos com endereçamento **far** (Options, Compiler, Force Far Calls: On) ou colocar a diretiva {$F+} antes da declaração das rotinas. No Free Pascal é necessário compilar o programa com a opção de `-So` na linha de comando.

Extended Pascal O padrão Extended Pascal não apresenta declaração de novos tipos como sendo procedimentos ou funções. Estes parâmetros devem ser declarados diretamente na declaração do parâmetro.

As sintaxes das declarações de parâmetro como procedimento ou função é a mesma que a usada nos cabeçalhos de funções.

Exemplo: **procedure** P(**function** F(x: **real**): real);
No cabeçalho do procedimento P, temos a declaração de um parâmetro chamado F que é uma função. F é uma função que tem um parâmetro real e retorna um valor real.

O padrão do pascal extendido (*Extended Pascal standard*) foi completado em 1989 e é tanto um padrão ANSI/IEEE como ISO/IEC.

Exemplo 12.1 *No exemplo das figuras 38 e 39, temos um procedimento chamado imprime, com 4 parâmetros: dois números reais (a e b), uma função (fun) e uma string (msg). O procedimento imprime um texto que contém os dois valores reais e o resultado da chamada da função sobre estes dois números. Note que esta rotina poderia ter sido chamada com qualquer função que tem dois parâmetros com valores reais e retorna um valor real. No caso, a rotina imprime foi chamada com as funções maximo, minimo e media.*
Note que na declaração, foram colocados identificadores associados aos parâmetros. A declaração destes tipos necessita que tenhamos identificadores, mas seu nome não precisa estar associado aos nomes dos identificadores na descrição da rotina.

A tradução de programas escritos em Extended Pascal para Borland Pascal é simples, uma vez que precisamos apenas mudar a declaração de rotinas que são parâmetros, de forma a usar tipos. Já a tradução de um programa escrito em Borland Pascal para Extended Pascal pode ser mais complicada, se aquele usar os tipos de rotinas dentro de estruturas como vetores e registros.

A partir de agora usaremos apenas uma destas sintaxes. Escolhemos a sintaxe do Borland Pascal, uma vez que esta sintaxe pode ser usada na maioria dos compiladores Pascal, tais como: Turbo Pascal (versão 5 em diante), Borland Pascal, Delphi, gpc (Gnu Pascal Compiler) e Free Pascal. Acreditamos que futuras padronizações da linguagem Pascal venham a contemplar este tipo de sintaxe.

```
program ParametroFuncaoBorlandPascal;
type TipoFuncao = function(a,a : real):real;
    TipoMsg = string[100];
function maximo(a,b : real):real;
begin
  if (a>b) then maximo := a
  else maximo := b;
end;
function minimo(a,b : real):real;
begin
  if (a>b) then minimo := b
  else minimo := a;
end;
function media(a,b : real):real;
begin
  media := (a+b)/2;
end;
procedure imprime(a,b : real; fun: tipoFuncao;
            msg : TipoMsg);
begin
  writeln(msg,' de ',a:7:2,' e ',b:7:2,' é: ',
       fun(a,b):7:2);
end;

var n1,n2 : real;
begin
  write('Entre com dois números: ');
  readln(n1,n2);
  imprime(n1,n2,maximo,'O máximo');
  imprime(n1,n2,minimo,'O mínimo');
  imprime(n1,n2,media,'A média');
end.
```

Figura 38: Exemplo de rotinas como parâmetros em Borland Pascal

```
program ParametroFuncaoExtendedPascal;
type TipoMsg = string[100];
function maximo(a,b : real):real;
begin
  if (a>b) then maximo := a
  else maximo := b;
end;
function minimo(a,b : real):real;
begin
  if (a>b) then minimo := b
  else minimo := a;
end;
function media(a,b : real):real;
begin
  media := (a+b)/2;
end;
procedure imprime(a,b            : real;
            function fun(a,a : real):real;
            msg              : TipoMsg);
begin
  writeln(msg,' de ',a:7:2,' e ',b:7:2,' é: ',
       fun(a,b):7:2);
end;

var n1,n2 : real;
begin
  write('Entre com dois números: ');
  readln(n1,n2);
  imprime(n1,n2,maximo,'O máximo');
  imprime(n1,n2,minimo,'O mínimo');
  imprime(n1,n2,media,'A média');
end.
```

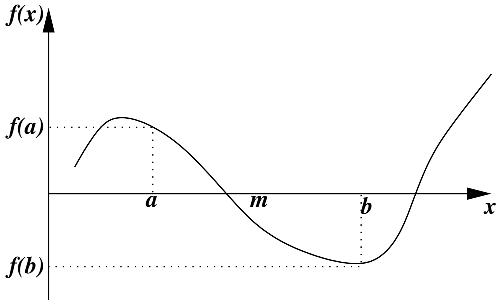
Figura 39: Exemplo de rotinas como parâmetros em Extended Pascal

12.2 Método de bisseção para encontrar raízes de funções

Um método para encontrar soluções de uma função é através do método de bisseção. Para usar estes método devemos ter:

- Uma função $f(x)$ para poder calcular um valor de x_0 tal que $f(x_0) = 0$.
- Um intervalo de busca da solução $[a, b]$, de tal forma que $f(a) \cdot f(b) \leq 0$.

Note que o segundo item nos garante que ou a é solução, ou b é solução ou deve existir uma solução x_0 no intervalo $[a, b]$ tal que $f(x_0) = 0$ (veja a figura seguinte).



A idéia é dividir este intervalo em duas partes pelo meio, m, digamos $[a, m]$ e $[m, b]$. Devemos ter pelo menos uma das condições seguintes:

1. $f(a) \cdot f(m) \leq 0$
2. $f(m) \cdot f(b) \leq 0$.

Se $f(a) \cdot f(m) \leq 0$ então necessariamente temos uma solução dentro do intervalo $[a, m]$ e repetimos o mesmo processo, desta vez iterando com o intervalo $[a, m]$ no lugar do intervalo $[a, b]$. Caso contrário, teremos uma solução dentro do intervalo $[m, b]$ e continuamos o processo neste intervalo. A cada iteração o tamanho do intervalo diminui para metade do tamanho do intervalo anterior. Assim, o processo pode parar quando o comprimento do intervalo for suficientemente pequeno.

Com isto, podemos escrever uma rotina que encontra uma solução para uma certa função (recebida pela rotina como parâmetro) e o intervalo de atuação.

O método de bisseção já pressupõe um intervalo $[a, b]$ válido tal que $f(a) \cdot f(b) \leq 0$. Para encontrar tal intervalo, podemos começar com um intervalo suficientemente grande $[a_0, a_k]$, dividimos este intervalo em partes pequenas, digamos em k partes iguais, $[a_0, a_1]$, $[a_1, a_2]$, ..., $[a_{k-1}, a_k]$ e verificamos se algum destes intervalos digamos $[a_{i-1}, a_i]$, $i \in \{1, \ldots, k\}$ é tal que $f(a_i) \cdot f(a_{i+1}) \leq 0$. Caso exista tal intervalo, usamos o método da bisseção no intervalo, caso contrário terminamos sem sucesso. Naturalmente podem ocorrer instâncias onde este método não encontra soluções, mesmo que tal solução se encontre em um dos intervalos iniciais. Além disso, dificilmente este método irá encontrar soluções de funções não negativas ou não positivas.

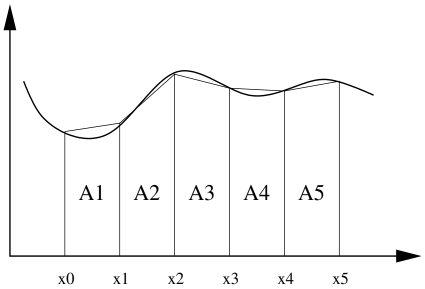
O programa a seguir apresenta a implementação do método descrito acima.

```
program ProgramaBicessao;
const EPS = 0.000000001;
type TipoFuncaoReal = function (x:real):real;
{Encontrando o intervalo, retorna
 sucesso=true e a0 e ak atualizados com
 o intervalo, caso contrário, retorna com
 sucesso=false}
procedure EncontraIntervalo(f: TipoFuncaoReal ;
        var a0,ak: real;k:integer;
        var sucesso: boolean);
var delta,a: real;
begin
  a:=a0;
  delta := (ak-a)/k;
  while (a < ak) and (f(a)*f(a+delta)>0)
    do a := a + delta;
  if (f(a)*f(a+delta)>0) then sucesso := false
  else begin
    a0 := a;
    ak := a+delta;
    sucesso := true;
  end;
end;

{Encontrando solução, retorna sucesso=true e
 a raiz em solução, cc. sucesso=false}
procedure bicessao(f: TipoFuncaoReal; a,b:real;
      var sucesso:boolean;var solucao:real);
var m : real;
begin
  EncontraIntervalo(f,a,b,1000,sucesso);
  if (sucesso) then begin
    while (abs(a-b)>EPS) do begin
      m := (a+b)/2;
      if (f(a)*f(m)<=0) then b:=m
      else a:=m;
    end;
    solucao := m;
  end;
end;

{função real: f(x)=x^3 + 2.x^2 - x + 10 }
function f1(x : real):real;
begin
  f1 := x*x*x + 2*x*x - x + 10;
end; { f1 }
```

```
{função real: f(x)=(x-5)*(x-2)*(x-3)*(x-4)*(x-6)}
function f2(x : real):real;
begin
  f2 := (x-5)*(x-2)*(x-3)*(x-4)*(x-6);
end; { f1 }

var sucesso : boolean;
   raiz: real;
begin
  bicessao(f1,-10,10,sucesso,raiz);
  if (sucesso) then
    writeln('Uma solução de f(x)=x^3+2.x^2-
x+10 é: ',
    raiz:10:8);
  bicessao(f2,-10,10,sucesso,raiz);
  if (sucesso) then
    writeln('Uma solução de ',
         'f(x)=(x-5)*(x-2)*(x-3)*(x-4)*(x-
6) é: ',
         raiz:10:8);
end.
```

Exercício 12.1 *(Integral de funções) O cálculo da integral de uma curva definida f pode ser calculado por aproximações utilizando-se o método dos trapézios. Neste método dividimos o intervalo de integração em n partes iguais, digamos divididas pelos pontos $x_0, x_1, \ldots, x_n$ e calculamos a área do trapézio substituindo a curva em cada intervalo x_{i-1}, x_i por uma reta ligando os pontos $f(x_{i-1})$ e $f(x_i)$, $i = 1, \ldots, n$ (veja a figura seguinte). O valor aproximado da integral definida é a soma das áreas dos n trapézios.*

$$\begin{aligned}
\int_a^b f(x)dx &\approx \sum_{i=1}^{n} A_i, \quad \textit{onde} \\
A_i &= \frac{f(x_{i-1}) + f(x_i)}{2} \cdot \delta x, \quad i = 1, \ldots, n \quad \textit{Área de cada trapézio} \\
\delta x &= x_i - x_{i-1} = \frac{b-a}{n} \\
x_0 &= a \\
x_n &= b.
\end{aligned}$$



Naturalmente, quanto maior o valor de n, melhor a aproximação da integral. Faça um programa contendo uma função chamada integral com o seguinte cabeçalho:

function *integral(f: TipoFuncaoReal; a,b:* **real***; n:* **integer***):* **real***;*

onde TipoFuncaoReal é um tipo de função como declarado como programa de bicessão; os valores a e b definem o intervalo de integração: $[a, b]$; e o valor n indica a quantidade de intervalos a ser usado no método de aproximações.

12.3 Ordenação usando funções de comparação

No exemplo 6.9 vimos como podemos ordenar um vetor colocando na posição correta o maior elemento não considerado a cada instante. Note que o algoritmo faz a seleção do maior elemento fazendo várias comparações entre dois elementos (se menor, igual ou maior). Em muitos problemas, cada elemento do vetor contém informações de um objeto que contém em geral várias características e estes podem ser ordenados de várias maneiras, dependendo das características consideradas.

Por exemplo, vamos considerar que uma universidade, chamada UNICOMP, armazena os dados dos alunos usando o tipo *TipoAluno*, apresentado na página 93. Por simplicidade, vamos considerar os campos *Nome, DataNascimento* e *CR*. Para efeitos de busca e classificação, a universidade necessita que estes dados possam estar em ordem alfabética pelo nome ou listados pelos alunos mais novos primeiro ou listados ordenados pelo CR em ordem decrescente. Como é apenas a comparação entre elementos que muda, podemos fazer apenas uma rotina de ordenação e usar diferentes funções de comparação para decidir qual aluno vem antes de outro.

A função para comparar dois elementos tem o seguinte formato:

function *Compara(***var** *a,b: TipoAluno):char;*

a função retorna '<', '=' ou '>', se a é menor, igual ou maior que b, respectivamente. A seguinte função compara os nomes de dois alunos:

```
function ComparaNome(Aluno1,Aluno2 : TipoAluno):char;
begin
  if (Aluno1.Nome < Aluno2.Nome) then ComparaNome := '<'
  else if (Aluno1.Nome > Aluno2.Nome) then ComparaNome := '>'
  else ComparaNome := '=';
end;
```

A seguir, vamos descrever a nova rotina de ordenação (atualizada a partir do exemplo 7.13) que pode usar a função de comparação por nome, dado acima.

```
const MAX = 100;
type { --> Incluir tipos em TipoAluno <-- }
   TipoVetorAluno = array[1..MAX] of TipoAluno;
   TipoFuncaoComparacao = function (a,b: TipoAluno):char;
procedure SelectionSortAluno(fcomp: TipoFuncaoComparacao; var v : TipoVetorAluno; n:integer);
var m,imax : integer;
 procedure TrocaAluno(var Aluno1, Aluno2 : TipoAluno);
  var AlunoAux : TipoAluno;
  begin  AlunoAux:=Aluno1;   Aluno1:=Aluno2;   Aluno2:=AlunoAux;  end;

  function IndMaxAluno(fcomp : TipoFuncaoComparacao; var v:TipoVetorAluno; n:integer):integer;
  var i, Ind : integer;
  begin
    if (n <=0) then Ind := 0
    else begin
      Ind := 1; {O maior elemento começa com o primeiro elemento do vetor}
      for i:=2 to n do if (fcomp(v[Ind],v[i])='<') then Ind:=i;
    end;
    IndMaxAluno := Ind;
  end;
begin
  for m:=n downto 2 do begin
     imax := IndMaxAluno(fcomp,v,m);
     TrocaAluno(v[m],v[imax]);
  end;
end;
```

Note que a mudança (além do tipo de dado) do programa apresentado no exemplo 6.9 está no uso da função de comparação entre dois elementos, que é um parâmetro usado na chamada da rotina *SelectionSortAluno*. Assim, a chamada da rotina *SelectionSortAluno(ComparaAluno, V, n)* fará a ordenação do vetor V, com n elementos, usando a função de comparação *ComparaAluno*. I.e., o vetor ficará em ordem alfabética pelo nome.

Agora, para fazer a ordenação em ordem crescente de idade (mais novos primeiro), podemos usar o mesmo programa usando apenas uma função de comparação diferente. Mas note que agora queremos os de maior data de nascimento primeiro. Para isso, vamos "*tapear*" a rotina de ordenação, usando uma função de comparação que responde dizendo que o aluno A_1 é *menor* ('<') que o aluno A_2 se a data de nascimento de A_1 for na verdade *maior* que a de A_2, e vice versa. Isto fará com que a rotina ordene de maneira invertida. Lembre que a rotina de ordenação foi feita de maneira que os menores ficassem nas primeiras posições; assim, se usamos a rotina de comparação que diz que um elemento é menor quando na verdade é maior, e vice-versa, obteremos uma ordenação com os maiores nas primeiras posições. A rotina de comparação de alunos por data de nascimento (mais novos primeiro) pode ser dada pela seguinte função:

```
function ComparaIdade(Aluno1,Aluno2 : TipoAluno):char;
begin
  if (Aluno1.DataNascimento.Ano < Aluno2.DataNascimento.Ano) then ComparaIdade := '>'
  else if (Aluno1.DataNascimento.Ano > Aluno1.DataNascimento.Ano) then ComparaIdade := '<'
  else if (Aluno1.DataNascimento.Mes < Aluno1.DataNascimento.Mes) then ComparaIdade := '>'
  else if (Aluno1.DataNascimento.Mes > Aluno1.DataNascimento.Mes) then ComparaIdade := '<'
  else if (Aluno1.DataNascimento.Dia < Aluno1.DataNascimento.Dia) then ComparaIdade := '>'
  else if (Aluno1.DataNascimento.Dia > Aluno1.DataNascimento.Dia) then ComparaIdade := '<'
  else ComparaIdade := '=';
end;
```

Note também que o uso da função de comparação permite fazer comparações de elementos levando em consideração vários dados para desempate. O programa final é dado a seguir.

```
program OrdenaAlunoIdadeNome;
const MAX = 100;
type { --> Incluir tipos em TipoAluno <-- }
   TipoVetorAluno = array[1..MAX] of TipoAluno;
   TipoFuncaoComparacao = function (a,b: TipoAluno):char;
function ComparaNome(Aluno1,Aluno2 : TipoAluno):char;
var Nome1,Nome2       : TipoNome;
begin
  if (Aluno1.Nome < Aluno2.Nome) then ComparaNome := '<'
  else if (Aluno1.Nome > Aluno2.Nome) then ComparaNome := '>'
  else ComparaNome := '=';
end; { ComparaNome }
function ComparaIdade(Aluno1,Aluno2 : TipoAluno):char;
begin { Inserir código da função ComparaIdade, visto anteriormente. } end;

procedure SelectionSortAluno(fcomp: TipoFuncaoComparacao; var v : TipoVetorAluno; n:integer);
var m,imax : integer;
  procedure TrocaAluno(var Aluno1, Aluno2 : TipoAluno);
   var AlunoAux : TipoAluno;
   begin AlunoAux:=Aluno1;  Aluno1:=Aluno2;  Aluno2:=AlunoAux; end;
  function IndMaxAluno(fcomp : TipoFuncaoComparacao; var v:TipoVetorAluno; n:integer):integer;
   var i, Ind : integer;
   begin
     if (n <=0) then Ind := 0
     else begin
       Ind := 1; {O maior elemento começa com o primeiro elemento do vetor}
       for i:=2 to n do if (fcomp(v[Ind],v[i])='<') then Ind:=i;
     end;
     IndMaxAluno := Ind;
   end;
begin
  for m:=n downto 2 do begin
    imax := IndMaxAluno(fcomp,v,m);
    TrocaAluno(v[m],v[imax]);
  end;
end;
procedure ImprimeVetorAluno(var v : TipoVetorAluno; n:integer; msg:TipoString);
begin { Exercício: Rotina para imprimir os Alunos na tela } end;
procedure LeVetorAluno(var v:TipoVetorAluno;var n:integer);
begin { Exercício: Rotina para ler Alunos } end;

var i, n, Ind : integer;
  V       : TipoVetorFunc;
begin
  LeVetorAluno(V,n);
  SelectionSortFunc(comparaNome,v,n);
  ImprimeVetorFunc(v,n,'Ordenados por nome (ordem alfabética)');
  SelectionSortFunc(comparaIdade,v,n);
  ImprimeVetorFunc(v,n,'Ordenados por idade (mais novos primeiro)');
end.
```

Exercício 12.2 *A universidade UNICOMP deseja dar bolsas de estudo para os alunos que apresentam melhores valores do Coeficiente de Rendimento (CR). Assim, faça uma função de comparação para ordenar os alunos, colocando os de maiores valores do CR primeiro. Caso dois alunos possuam o mesmo valor de CR, estes devem ser desempatados pelo RA.*

12.4 Exercícios

1. É dado uma função $f : \mathrm{R} \rightarrow \mathrm{R}$ e um intervalo $[a, b]$ tal que $f(a) \cdot f(b) < 0$. Faça uma função recursiva que encontra uma solução $x \in [a, b]$ usando o método da bisseção (dividindo sucessivamento o intervalo ao meio) tal que $f(x) = 0$. A função pode dar o resultado dentro de uma precisão de $0,0001$ (diferença entre a solução verdadeira e a solução encontrada pela rotina) e deve ter o seguinte cabeçalho:

   function Bissecao(f: TipoFuncao; a, b: real): real;

   onde f é a função para a qual calcularemos a solução e $[a, b]$ define o intervalo que contém a solução. O tipo *TipoFuncao* é definido como:

   type TipoFuncao = function (x:real):real;

2. Para calcular a integral $S = \int_a^b f(x)dx$ pode-se usar a aproximação de uma soma finita de "valores amostrados":

   $$S_k = \frac{h}{3}(f_0 + 4f_1 + 2f_2 + 4f_3 + 2f_4 + \cdots + 4f_{n-3} + 2f_{n-2} + 4f_{n-1} + f_n)$$

   onde

   $$f_i = f(a + ih), \;\; h = (b - a)/n, \;\; \text{e} \;\; n = 2^k.$$

   O número de pontos de amostragem é $n + 1$ e h é a distância entre dois pontos de amostragem adjacentes. O valor S da integral é aproximado pela seqüência $S_1, S_2, S_3, \ldots$, que converge se a função é suficientemente bem comportada (suave). O método acima é chamado de *método de Simpson*.

   Escreva uma função que calcule a integral de uma função $f(x)$ usando o método de Simpson. A função tem como parâmetros: a função $f(x)$, o intervalo de integração $[a, b]$ e o valor k. A função retorna a aproximação da integral de $f(x)$ no intervalo $[a, b]$. Aplique sua função para calcular as integrais:

   $$\int_0^2 x^2 dx \qquad \text{e} \qquad \int_0^{\pi/2} \mathrm{sen}(x)dx$$

3. Faça um programa para manipular um cadastro de alunos, nos moldes do exemplo da seção 12.3, de maneira a ter as seguintes opções:

   (a) Iniciar cadastro vazio (inicialmente o cadastro já começa vazio (número de elementos igual a zero)).
   (b) Inserir um novo elemento no cadastro (se o cadastro estiver cheio, avise que não há memória disponível).
   (c) Ordenar o cadastro por nome em ordem alfabética.
   (d) Ordenar o cadastro por idade, mais novos primeiro.
   (e) Ordenar o cadastro por idade, mais velhos primeiro.
   (f) Ordenar o cadastro por RA, ordem crescente.
   (g) Ordenar o cadastro por CR, maiores primeiro.

   Sempre que o programa executar alguma destas opções, ele deve imprimir a lista de alunos na ordem do vetor e imprimir logo em seguida o menu antes de pedir por nova opção. Além disso, o programa deve usar apenas de uma rotina de ordenação para ordenar os alunos. Os diferentes tipos de ordenação são definidos usando funções de comparação específicas para cada tipo de ordenação que devem ser passadas como parâmetros da rotina de ordenação.

13 Arquivos

Quando um programa trabalha com dados que estão na memória RAM, estes são apagados sempre que o computador é desligado ou mesmo quando o programa é terminado. Mas muitas vezes queremos que os dados se mantenham em memória para que possam ser processados posteriormente. Neste caso, é necessário que estes dados estejam gravados em uma memória não volátil. Este é o caso de se usar *arquivos*, que permite gravar os dados e recuperá-los sem necessidade de reentrada dos dados. Outra razão para se usar arquivos é que estes são armazenados em memória secundária, discos e fitas magnéticas, que têm em geral uma capacidade muito maior de armazenamento.

Podemos distinguir os arquivos suportados pela linguagem Pascal como sendo de dois tipos: arquivos de texto e arquivos binários.

13.1 Arquivos Texto

Quando trabalhamos com arquivos de texto, supomos que não iremos trabalhar com qualquer tipo de dado, mas com dados que nos representem um texto. Neste caso, vamos supor que os dados de um texto estão organizadas por linhas, onde cada linha contém uma seqüência de caracteres ASCII que podem ser impressos. Exemplo destes caracteres são as letras (maiúsculas/minúsculas/com acento), dígitos e os caracteres especiais (como: $, %, #, , ?, ...). Além disso, um arquivo texto contém alguns caracteres de controle, como por exemplo os caracteres que indicam o fim de uma linha. A unidade básica em arquivos texto é o caracter.

Um programa em Pascal considera o vídeo e o teclado como arquivos texto especiais. O vídeo como um arquivo onde podemos apenas escrever e o teclado como um arquivo que podemos apenas ler.

Existem algumas diferenças entre um arquivo texto feito para o sistema operacional MS-DOS e um arquivo texto feito para o ambiente UNIX. Um arquivo texto no ambiente MS-DOS usa dois caracteres para especificar que começará a próxima linha, o caracter <u>Carriage Return</u> seguido do caracter <u>Line Feed</u>, que tem os códigos internos 13 e 10, respectivamente. No caso dos arquivos texto em ambiente UNIX, a próxima linha começa apenas depois de um caracter, <u>Line Feed</u>, de código interno 10.

Quando trabalhamos com arquivos, é importante considerar que todo arquivo tem um indicador da posição corrente de leitura e escrita (gravação). Inicialmente este indicador está posicionado no primeiro byte do arquivo. Se for realizada uma operação de leitura ou gravação de n bytes, então o indicador da posição corrente no arquivo se deslocará destes n bytes para frente. Além disso, existe uma posição que será a seguinte do último byte do arquivo (posição de fim de arquivo) e não poderemos ler nenhum byte seguinte, mas poderemos escrever outros bytes a partir desta posição.

A sintaxe para declarar uma variável para trabalhar com arquivos texto é a seguinte:

var Lista_de_Identificadores_de_Arquivos: **Text**;

O identificador declarado desta forma será usado para trabalhar com um arquivo que está gravado em disco. O arquivo que está em disco (ou outro meio de armazenamento secundário) tem um nome e a primeira operação para trabalhar com ele é *associar* este nome *(nome externo)* com a variável de arquivo. O comando para fazer esta associação é a seguinte:

assign(Variável_de_arquivo,Nome_Arquivo_Externo);

Esta operação apenas "diz" para o programa qual o arquivo (que está em disco) que iremos trabalhar pela variável de arquivo. Mas isto não faz com que os dados que estão no arquivo (em disco) já estejam disponíveis para serem manipulados. Para isso, precisamos dar outro comando que de fato irá possibilitar esta manipulação. Informalmente, este comando irá "<u>abrir</u>" o arquivo para ser manipulado. Podemos ter três tipos de comandos para abrir um arquivo texto, um que é somente para leitura dos dados, outro que é para escrita em um arquivo novo e outro que é para escrita a partir do fim do arquivo texto.

Para abrir um arquivo <u>somente para leitura</u> (a partir do início do arquivo), usamos o seguinte comando:

reset(Variável_de_arquivo);

Note que não precisamos dar o nome externo do arquivo, uma vez que isto já foi feito pelo comando **assign**. Neste caso, o indicador de posição corrente fica apontando para o início do arquivo.

Para abrir um arquivo <u>somente para escrita</u> começando com um arquivo vazio, usamos o seguinte comando:

rewrite(Variável_de_arquivo);

Se já existir um arquivo com o mesmo nome, o arquivo que existia antes é reinicializado como arquivo vazio. Como neste caso sempre começamos com um arquivo vazio, o indicador da posição corrente começa no início do arquivo.

Por fim, para abrir um arquivo somente para escrita a partir do fim do arquivo (i.e., o conteúdo inicial do arquivo é preservado) usamos o seguinte comando:

append(Variável_de_arquivo);

Neste caso, o indicador da posição corrente no arquivo fica inicializado como o fim de arquivo.

O comando de abertura de arquivo, em um programa Pascal, usa uma memória auxiliar (em memória RAM) para colocar parte dos dados que são lidos/escritos do/para o arquivo. Isto é porque em geral, qualquer mecanismo de armazenamento secundário (disco rígido, fitas magnéticas, disquetes,...) funcionam através de meios eletro-mecânicos, o que força com que a leitura e escrita de dados usando estes dispositivos seja muito lenta. Assim, sempre que uma operação de leitura ou escrita é feita, ela não é feita diretamente no arquivo, mas sim nesta memória auxiliar (que é extremamente rápida). Caso algum comando faça uso de um dado do arquivo que não está nesta memória auxiliar, o programa usa serviços do sistema operacional para atualizar o arquivo com a memória auxiliar e colocar na memória auxiliar os novos dados que o programa precisa manipular.

Se um programa for finalizado sem nenhum comando especial, os arquivos que ele estiver manipulando poderão não estar atualizados corretamente, já que parte dos dados pode estar na memória auxiliar. Assim, quando não precisarmos mais trabalhar com o arquivo, é necessário usar de um comando que descarrega todos os dados desta memória auxiliar para o arquivo propriamente dito (além de liberar esta memória auxiliar para futuro uso do sistema operacional). Isto é feito com o comando **close** da seguinte maneira:

close(Variável_de_arquivo);

Uma vez que o arquivo foi aberto, poderemos ler e gravar dados (dependendo do modo como o arquivo foi aberto) no arquivo. Lembrando que para ler do teclado, usamos o comando read/readln, e para escrever no vídeo, usamos o comando write/writeln. Do mesmo jeito, para ler e escrever (gravar) em um arquivo texto, usamos os comandos read, readln, write e writeln. A única diferença é que colocamos um parâmetro a mais nestes comandos, especificando em qual arquivo será feita a operação de leitura ou escrita.

Assim, vamos supor que em determinado momento os dados do arquivo e a posição corrente no arquivo estão como apresentados no quadro seguinte, onde apresentamos as três primeiras linhas do arquivo e a posição corrente está indicado por uma seta.

```
1   2   3   Esta é a primeira linha do arquivo texto

123  456  3.14   999  Esta é a segunda linha
     ↑
Terceira Linha
```

se dermos o comando:

read(Variável_de_arquivo,a, b, c);

onde a, b e c são variáveis reais, teremos o mesmo resultado se déssemos o comando **read**(a, b, c); e entrássemos pelo teclado com os dados 456 3.14 999. I.e., as variáveis a, b e c ficarão com os valores 456, 3.14 e 999, respectivamente, e o indicador da posição corrente é atualizado de maneira a ficar com a seguinte configuração:

```
1   2   3   Esta é a primeira linha do arquivo texto

123  456  3.14   999  Esta é a segunda linha
                      ↑
Terceira Linha
```

Outra rotina importante é a função para verificar se estamos no fim de arquivo, pois neste caso não poderemos ler mais nenhum dado a partir do arquivo texto. A função para fazer esta verificação é a função booleana **eof**, cuja sintaxe é:

eof(Variável_de_arquivo)

A função **eof** retorna verdadeiro (**true**) caso o indicador da posição corrente esteja no fim do arquivo e falso (**false**) caso contrário.

Além do comando **read**, também podemos usar o comando **readln**. Neste caso, os dados são lidos como no comando **read**, mas se ainda existir alguns dados na mesma linha corrente do arquivo texto, estes são ignorados e o indicador da posição corrente fica situado no primeiro caracter da próxima linha.

Se a variável a ser lida é uma cadeia de caracteres (**string**), a quantidade de caracteres é definida pelo número de caracteres que ela foi definida. Além disso, se a rotina **read** for aplicada para ler uma cadeia de caracteres e a quantidade de caracteres a ler ultrapassa a linha do arquivo, o indicador da posição corrente fica posicionado no fim desta linha. I.e., se uma próxima leitura com o comando **read** for usado para ler uma string, o indicador de posição corrente se manterá naquele mesmo fim de linha. Neste caso, para começar a ler a próxima linha como string, o comando de leitura anterior deve ser o **readln**. Caso a leitura seja de algum outro tipo (**integer, real, ...**), mesmo que se use o comando **read**, e o dado não estiver na linha corrente, o comando busca o dado a ser lido nas próximas linhas.

Os comandos de impressão (gravação), **write** e **writeln** têm resultado semelhante aos correspondentes comandos para imprimir na tela. A diferença é que o resultado da impressão será feita no arquivo. Além disso, um arquvo texto não tem a limitação de número de colunas, como temos em um vídeo. Lembrando que para ir para a próxima linha, em um arquivo texto MS-DOS são usados os caracteres *Carriage_Return* e *Line_Feed*, enquanto em arquivos Unix, é usado apenas o caracter *Line_Feed*.

Em um arquivo texto, como em um arquivo binário, podemos obter a quantidade de elementos básicos do arquivo através da função *filesize*, cuja sintaxe é:

filesize(Variável_de_arquivo)

e retorna a quantidade de elementos básicos do arquivo.

Exemplo 13.1 *O seguinte programa lê o nome de um arquivo texto e o lista na tela, especificando o número de cada linha. Por fim, o programa também imprime o número de linhas e o número de bytes do arquivo.*

```
program ListaArquivo(input,output);
type TipoString = string[255];
procedure LeArquivoTexto(NomeArquivo:TipoString);
var ArquivoTexto : text;
  linha      : TipoString;
  nlinha     : integer;
begin
  assign(ArquivoTexto,NomeArquivo); {Associar o nome do arquivo com a variável}
  reset(ArquivoTexto);              {Abrir o arquivo para leitura}
  nlinha:=0;
  while not eof(ArquivoTexto) do begin
    nlinha:=nlinha+1;    readln(ArquivoTexto,linha);    writeln('[',nlinha:3,'] ',linha);
  end;
  writeln(NomeArquivo,' tem ',nlinha,' linhas e usa ',filesize(ArquivoTexto),' bytes.');
  close(ArquivoTexto); {Fecha o arquivo}
end; { LeArquivoTexto }
var NomeArq : TipoString;
begin
  write('Entre com o nome do arquivo a listar: ');
  readln(NomeArq);
  LeArquivoTexto(NomeArq);
end.
```

Exemplo 13.2 *O programa seguinte elimina os espaços em branco repetidos (seguidos) de um arquivo texto.*

```
program Brancos;
var
  linha                  : String;
  nomeEntrada, nomeSaida : String[20];
  arqEntrada, arqSaida   : Text;

procedure EliminaBrancos(var s: String);
var
  i,desl,n : Integer;
  p        : char;
begin
  desl := 0;  n := Length(s); p := 'x';
  for i:=1 to n do
  begin
    if s[i]<>' '
      then s[i−desl] := s[i]
    else if p<>' '
      then s[i−desl] := s[i]
    else inc(desl);
    p := s[i]
  end;
  Setlength(s,n−desl)
end; {EliminaBrancos}

begin
  Write('Forneça o nome do arquivo de entrada: ');
  Readln(nomeEntrada);
  Write('Forneça o nome do arquivo de saída: ');
  Readln(nomeSaida);
  AssignFile(arqEntrada,nomeEntrada);
  AssignFile(arqSaida,nomeSaida);
  Reset(arqEntrada);
  Rewrite(arqSaida);
  while not eof(arqEntrada) do
  begin
    Readln(arqEntrada,linha);
    EliminaBrancos(linha);
    Writeln(arqSaida,linha)
  end;
  Close(arqSaida)
end.
```

Exercício 13.1 *Os professores do curso de Algoritmos e Programação de Computadores desenvolveram uma linguagem assembler, chamada HIP, para um computador hipotético com conjunto de comandos bastante reduzido. A linguagem é voltada para manipulação de inteiros e tem 16 instruções que explicitaremos a seguir: O único tipo de dado existente em HIP é o tipo inteiro. Cada valor inteiro deve ser armazenado em uma variável representada por um identificador que usa apenas uma letra. Além de variáveis, os identificadores podem representar labels que podem ser pontos de desvios do programa.*

- **var** *<Ident>*
  Faz a declaração da variável <Ident>, inicializada com valor 0 (zero).
- **ler** *<Ident>*
  Lê o valor da variável <Ident> pelo teclado.
- **atr** *<Ident> <Constante>*
  Atribui a constante <Constante> (número inteiro) para a variável <Ident>.
- **mov** *<Ident> <Ident1>*
  Atribui o valor da variável <Ident1> para a variável <Ident>.
- **sum** *<Ident> <Ident1> <Ident2>*
  Soma os valores das variáveis <Ident1> e <Ident2> e atribui o resultado na variável <Ident>.
- **sub** *<Ident> <Ident1> <Ident2>*
  Subtrai o valor da variável <Ident1> do valor da variável <Ident2> e atribui o resultado na variável <Ident>.
- **mul** *<Ident> <Ident1> <Ident2>*
  Multiplica os valores das variáveis <Ident1> e <Ident2> e atribui o resultado na variável <Ident>.
- **div** *<Ident> <Ident1> <Ident2>*
  Divide o valor da variável <Ident1> por <Ident2> e atribui a parte inteira na variável <Ident>.
- **mod** *<Ident> <Ident1> <Ident2>*
  Este comando atribui o resto da divisão inteira de <Ident1> por <Ident2> na variável <Ident>.
- **lab** *<Ident>*
  Este comando define um label (ponto de desvio do programa) com o nome <Ident>.
- **des** *<Ident>*
  Este comando desvia o fluxo de execução do programa para o comando com label <Ident>.
- **ifz** *<Ident> <Ident1>*
  Este comando desvia o fluxo de execução do programa para o comando com label <Ident1> se o valor da variável identificada por <Ident> tiver valor 0 (zero).
- **ifp** *<Ident> <Ident1>*
  Este comando desvia o fluxo de execução do programa para o comando com label <Ident1> se o valor da variável identificada por <Ident> tiver valor positivo.
- **ifn** *<Ident> <Ident1>*
  Este comando desvia o fluxo de execução do programa para o comando com label <Ident1> se o valor da variável identificada por <Ident> tiver valor negativo.
- **imp** *<Ident>*
  Imprime o valor da variável <Ident> na tela.
- **fim**
  Termina a execução do programa.

Vamos exemplificar esta linguagem com um programa feito em HIP para calcular o máximo divisor comum pelo algoritmo de Euclides.

Linha lida				*Explicação do comando*
var	a			*Declara a variável* a.
var	b			*Declara a variável* b.
var	r			*Declara a variável* r.
ler	a			*Lê um valor na variável* a *através do teclado.*
ler	b			*Lê um valor na variável* b *através do teclado.*
lab	x			*Declara ponto de desvio* x $\{$*Início da estrutura de repetição*$\}$
mod	r	a	b	$r \leftarrow a$ *mod* b.
mov	a	b		$a \leftarrow b$.
mov	b	r		$b \leftarrow r$.
ifz	r	y		*Até que* $r = 0$. *Neste caso, desvio o ponto* y.
des	x			*Volta a repetir desde o ponto* x.
lab	y			*Ponto de desvio fora da estrutura de repetição*
imp	a			*Impressão do MDC.*
fim				*Fim do programa.*

Faça um programa que lê um programa em HIP (armazenado como um arquivo texto) e faz a execução do programa. Para facilitar, armazene o programa em um vetor, comando por comando, incluindo as operações **var** *e* **lab**. *A posição que tiver o comando* **var** *contém um campo chamado valor que pode armazenar um valor inteiro.*

Exercício adicional: *Implemente programas em HIP para: (i) Calcular o fatorial de um número positivo. (ii) Verificar se um número positivo é primo.*

Exercício adicional: *Modifique o programa para que comandos como* **lab** *e* **var** *não sejam mais representados no vetor. As variáveis são representadas em um vetor adicional, chamado memória e os comandos de desvios são armazenado com o índice da posição no vetor para onde o fluxo é desviado. Os comandos que fazem manipulação de variáveis devem armazenar seus respectivos índices na memória. Estas modificações agilizam a execução de um programa em HIP, uma vez que não há necessidade de se buscar um identificador dentro do programa.*

13.2 Arquivos Binários

Um arquivo binário é um arquivo que não tem necessariamente a restrição de ter apenas texto. A linguagem Pascal trata os arquivos binários como sendo uma seqüência de elementos de arquivo. Os elementos de um mesmo arquivo tem o mesmo tipo, que pode ser um tipo básico da linguagem Pascal, ou outro definido pelo programador.

Os comandos **read** e **write** também podem ser usadas em arquivos binários (as funções readln e writeln não são válidas para arquivos binários), mas neste caso, usamos pelos menos dois parâmetros para estas funções, primeiro o arquivo a ler/escrever e a seguir as variáveis (de mesmo tipo que o elemento básico) que iremos ler/escrever.

Um arquivo binário é declarado da forma:

FILE of tipo_elemento_básico;

Exemplo 13.3 *A seguir apresentamos alguns exemplos de declaração de arquivos.*
ArqDados: File of TipoAluno;
ArqNums: File of Integer;
ArqVetCars: File of array [1..100] of char;
ArqCars: File of char;

Obs.: Arquivos texto (**text**) são muito semelhantes com arquivos do tipo **file of char**, mas arquivos do tipo **text** tem operações como EOLN *(End of Line)*, o que não ocorre com arquivos do tipo **file of char**.

Para associar o nome externo (nome do arquivo em disco) também usamos o comando assign, da mesma forma que usada para arquivos texto.

Os comandos de "abertura" de um arquivo binário são o **reset** e o **rewrite**.

Para abrir um arquivo para leitura e gravação de um arquivo já existente, usamos o comando:

reset(Variável_de_arquivo);

Note a diferença deste comando para arquivos texto e arquivos binários. Usando este comando para arquivos texto, só podemos ler do arquivo, enquanto para arquivos binários, podemos ler e escrever. Inicialmente os dados já existentes no arquivo são mantidos, podendo ser atualizados por comandos de escrita. Caso o arquivo já esteja aberto, este comando reposiciona o indicador de posição corrente do arquivo para o início do arquivo.

Para abrir um arquivo novo (vazio) para leitura e gravação começando com um arquivo vazio, usamos o comando **rewrite**:

rewrite(Variável_de_arquivo);

Se já existir um arquivo com o mesmo nome, o arquivo que existia antes é reinicializado como arquivo vazio. Com este comando também podemos ler, escrever/atualizar no arquivo binário.

Tanto usando o comando **reset** como **rewrite**, o indicador de posição corrente é inicializado no início do arquivo.

Para ler um elemento do arquivo (definido pela posição do indicador corrente do arquivo) podemos usar o comando **read**: **read**(Variável_de_arquivo,Variável_Elemento_Básico);
onde Variável_Elemento_Básico é a variável que irá receber o valor do elemento que está indicado no arquivo. Após a leitura deste elemento o indicador de posição corrente passa para o próximo elemento (fica em fim de arquivo, caso não exista o próximo elemento).

Para escrever um elemento no arquivo (na posição indicada pelo indicador de posição corrente do arquivo) usamos o comando **write**: **write**(Variável_de_arquivo,Variável_Elemento_Básico);
onde Variável_Elemento_Básico é a variável que irá receber o valor do elemento que está indicado no arquivo. Após a escrita deste elemento o indicador de posição corrente passa para o próximo elemento (fica em fim de arquivo, caso não exista o próximo elemento).

Quando usamos arquivos binários, podemos usar a função **eof**, que retorna **true** caso estejamos no fim de arquivo e **false** caso contrário e também podemos redefinir a posição do indicador corrente de arquivo, tanto para frente como para trás. O comando que nos permite mudar este indicador é o comando **seek** cuja sintaxe é a seguinte.

seek(Variável_de_arquivo,NovaPosição);

onde NovaPosição é um inteiro entre 0 e *(filesize(Variável_de_arquivo)-1)*. Com este comando, o indicador irá ficar posicionado no elemento de arquivo de número *NovaPosição*. Lembrando que o primeiro elemento está na posição 0 (zero) e filesize(Variável_de_arquivo) retorna o número de elementos de arquivo.

Exemplo 13.4 *A universidade UNICOMP mantém um cadastro de alunos, cada aluno do tipo TipoAluno, como declarado no exercício 9.2 da página 94. Para que os dados sejam armazenados de maneira permanente, em algum tipo de memória secundária (discos flexiveis, discos rígidos, ...) é necessário guardar os dados usando arquivos. Faça mais duas rotinas para o programa do exercício 9.2 de maneira que o programa possa gravar os alunos em um arquivo, lidos a partir de um vetor, e ler os alunos de um arquivo inserindo-os em um vetor. O nome do arquivo deve ser fornecido pelo usuário.*

```
program LeituraGravacaoDeAlunos;
type
{ --> Inserir aqui a declaração de TipoAluno e TipoCadastro <-- }
  mystring       = string[255];

procedure LeArquivo(var C : TipoCadastro; NomeArquivoDisco:mystring);
var Arq: File of TipoAluno; { Cada elemento básico do arquivo é do tipo TipoAluno }
begin
  assign(Arq,NomeArquivoDisco);
  reset(Arq);
  C.quant:=0;
  while not eof(Arq) do begin
    inc(C.quant);
    read(Arq,C.Aluno[C.quant]); { Le o n-ésimo elemento básico do arquivo }
  end; { Quando terminar o loop, n terá a quantidade de elementos }
  close(Arq);
end;

procedure GravaArquivo(var C: TipoCadastro; NomeArquivoDisco:mystring);
var i  : integer;
  Arq: File of TipoAluno; { Cada elemento básico do arquivo é do tipo TipoAluno }
begin
  assign(Arq,NomeArquivoDisco);
  rewrite(Arq); { Começa um arquivo vazio }
  for i:=1 to C.quant do write(Arq,C.Aluno[i]); { Grava n elementos básicos (alunos) }
  close(Arq);
end;

var i                  : integer;
  Cadastro           : tipocadastro;
  NomeArquivoDisco : mystring;
begin
  writeln('Entre com o nome do arquivo contendo os alunos: ');
  readln(NomeArquivoDisco);
  LeArquivo(Cadastro,NomeArquivoDisco);
  writeln('Os nomes dos alunos lidos são:');
  for i:=1 to n do writeln(Cadastro.Aluno[i].nome);
  writeln('Entre com o nome do arquivo para conter uma cópia do arquivo original: ');
  readln(NomeArquivoDisco);
  GravaArquivo(Cadastro,NomeArquivoDisco);
end.
```

Exemplo 13.5 *Vamos considerar que temos um sistema que mantém um cadastro de alunos, cada aluno do tipo TipoAluno, como no exemplo 13.4. Agora vamos supor que temos uma grande restrição de limitação de memória do computador (e conseqüentemente do programa) usado. I.e., não podemos guardar todos os alunos do cadastro em um vetor. Além disso, vamos supor que necessitamos da seguinte operação: Ordenar os alunos, no próprio arquivo e listá-los na tela. Desenvolva uma rotina de ordenação, por nome de aluno, usando não mais que uma quantidade limitada (constante) de memória para guardar alunos.*

```
program OrdenacaoDeAlunos;
uses tipoaluno1;
type
{ --> Inserir aqui a declaração de TipoAluno <-- }
  mystring          = string[255];
  TipoArquivoAluno =  File of TipoAluno;
procedure TrocaAlunoArquivo(var Arq :TipoArquivoAluno; i1,i2 : integer);
var Aluno1,Aluno2: TipoAluno;
begin
  seek(Arq,i1);  read(Arq,Aluno1);  seek(Arq,i2);  read(Arq,Aluno2);
  seek(Arq,i2);  write(Arq,Aluno1); seek(Arq,i1);  write(Arq,Aluno2);
end;
function IndMaximoArquivo(var Arq:TipoArquivoAluno; n : integer) : integer;
var i, Ind   : integer;
  Maior,Aux : TipoAluno;
begin
  seek(Arq,0);  read(Arq,Maior);
  Ind := 0; {O maior elemento começa com o primeiro elemento do vetor}
  for i:=1 to n do begin {Já começa no segundo elemento (posição 1) }
    read(Arq,Aux); {Le um elemento do arquivo}
    if Maior.nome < Aux.nome then begin
      Ind:=i;          {Se o elemento lido é maior que o que encontramos,}
      Maior:=Aux;      {Atualize o maior e o indice do maior }
    end;
  end;
  IndMaximoArquivo := Ind;
end; { IndMaximoArquivo }
procedure OrdenaAlunosArquivo(NomeArquivoDisco:mystring);{ Idéia do SelectionSort}
var Arq          : File of TipoAluno; { Cada elemento básico do arquivo é do tipo TipoAluno }
  n,m,imax : integer;
begin
  assign(Arq,NomeArquivoDisco); reset(Arq);
  n:=filesize(Arq)−1; { Obtém a posição do último elemento do arquivo }
  for m:=n downto 1 do begin { Modificação devido ao arquivo começar com posição 0 }
    imax := IndMaximoArquivo(Arq,m);
    TrocaAlunoArquivo(Arq,m,imax); {Troca os alunos das posições m e imax }
  end;
  close(Arq);
end; { OrdenaAlunosArquivo }
var aluno:TipoAluno;   n,i:integer;   Arq:TipoArquivoAluno;
begin
  assign(Arq,'Arquivo.dat'); reset(Arq);
  n:=filesize(Arq);
  for i:=1 to n do begin
    read(Arq,Aluno);  writeln(Aluno.nome);
  end;
end.
```

Exercício 13.2 *A rotina de ordenação de elementos de arquivo, sem uso de vetores (diretamente no arquivo), apresentada no exemplo 13.5, usa a estratégia do SelectionSort. I.e., localiza o elemento de maior valor e coloca na posição correta, e continua com o mesmo processo com os demais elementos. Faça uma rotina de ordenação, nas mesmas condições apresentadas no exemplo 13.5, mas agora usando o algoritmo QuickSort.*

Exercício 13.3 *A universidade UNICOMP tem milhares de funcionários e todo início de mês, quando deve fazer o pagamento de seus funcionários, deve imprimir uma listagem de funcionários, e para cada funcionário o programa deve imprimir seu nome, cargo e salário. O número de funcionários é grande, mas a quantidade de empregos diferentes é bem pequena. Há apenas 5 tipos de empregos diferentes, conforme a tabela a seguir:*

Cargo	*Salário*
Faxineira	*100,00*
Pedreiro	*110,00*
Carpinteiro	*120,00*
Professor	*130,00*
Reitor	*200,00*

Como os recursos da UNICOMP são escassos, ela deve economizar memória (em disco) do computador que fará a listagem. Assim, o programador que decide fazer o sistema resolve colocar a tabela acima em apenas um arquivo, chamado CARGOS. Cada registro deste arquivo é do seguinte tipo:

```
type
    tipocargo = record
        nomecargo: string[50];
        salario:real;
    end;
```

Outro arquivo a ser manipulado é o arquivo CADASTRO, que contém os funcionários. Este arquivo é da seguinte forma:

Nome	*PosCargo*
José Carlos	*2*
Carlos Alberto	*1*
Alberto Roberto	*0*
Roberto José	*4*
⋮	⋮

Cada registro deste arquivo é do seguinte tipo:

```
type
    tipofuncionario = record
        nome: string[50];
        poscargo:integer;
    end;
```

Desta maneira não é necessário guardar todos os dados do cargo junto com o do funcionário. Basta guardar um indicador de onde está a informação de cargo.

Se um funcionário tem o campo poscargo igual a k, então seu cargo é o cargo que esta na linha $(k+1)$ do arquivo de cargos. Assim, o funcionário "José Carlos"tem cargo "Carpinteiro"e ganha 120,00 reais; o funcionário "Roberto José"tem cargo "Reitor"e ganha 200,00 reais. Faca um programa (com menu) que:

- *Inicialmente gera o arquivo CARGOS, apresentado acima.*
- *Lê sempre que o usuário quiser, um novo funcionário e o inteiro indicando a posição do cargo dele para adicioná-lo no arquivo CADASTRO (inicialmente vazio).*
- *Se o usuário quiser lista todos os funcionários, um por linha, indicando o nome, cargo e salário dele.*

Sugestão: Durante a execução do programa, mantenha os dois arquivos abertos ao mesmo tempo.

Exercício 13.4 *Em um sistema computacional, existem dois arquivos, um chamado Funcionarios.dat e outro chamado Cargos.dat, ambos arquivos binários. O arquivo Funcionarios.dat é um arquivo de elementos do tipo tipofunc e o arquivo Cargos.dat é um arquivo de elementos do tipo tipocargo, definidos a seguir:*

```
type
    tipofunc = record
        nome:string[50];
        codcargo:string[2];
    end;
    tipocargo = record
        codcargo:string[2];
        nomecargo:string[50];
        salario:real;
    end;
```

A seguir apresentamos um exemplo do arquivo Funcionarios.dat e do arquivo cargos.dat. Ambos os arquivos podem ter vários elementos.

Nome	*Codcargo*
José Carlos	*Fa*
Carlos Alberto	*Re*
Alberto Roberto	*Pr*
Roberto José	*Pe*
⋮	⋮

Codcargo	*Cargo*	*Salário*
Fa	*Faxineira*	*100,00*
Pe	*Pedreiro*	*110,00*
Ca	*Carpinteiro*	*120,00*
Pr	*Professor*	*130,00*
Re	*Reitor*	*200,00*
⋮	⋮	⋮

O campo **codcargo** *é um código de cargo composto de duas letras e relaciona o cargo de um funcionário. No exemplo acima, o funcionário Carlos Alberto tem codcargo igual a Re e procurando no arquivo Cargos.dat, vemos que ele tem cargo de Reitor e ganha R$200,00 reais por mês. Faça um procedimento que imprime para cada funcionário (do arquivo Funcionarios.dat) seu nome, seu cargo e seu salário.*

Exercício 13.5 *Suponha que no programa anterior o arquivo Cargos tem seus registros sempre ordenados pelo campo Codcargo. Faça um procedimento, como no exercício anterior, que imprime para cada funcionário (do arquivo Funcionarios.dat) seu nome, seu cargo e seu salário, mas para buscar os dados de um cargo (nome do cargo e o salário) faz uma busca binária no arquivo Cargos pelo campo Codcargo. Faça rotinas auxiliares para inserção e remoção nestes arquivos.*

13.3 Outras funções e procedimentos para manipulação de arquivos

Alguns compiladores oferecem outras rotinas para manipular arquivos. Listamos algumas destas a seguir.

flush(vararq) Flush esvazia o buffer do arquivo (memória auxiliar em RAM) em disco vararq e garante que qualquer operação de atualização seja realmente feita no disco. O procedimento Flush nunca deve ser feito em um arquivo fechado.

erase(vararq) É um procedimento que apaga o arquivo em disco associado com vararq. Ao dar este comando, o arquivo não pode estar aberto, apenas associado (pelo comando assign).

rename(vararq,novonome) O arquivo em disco associado com vararq é renomeado para um novo nome dado pela expressão de seqüência de caracteres novonome. Ao dar este comando, o arquivo não pode estar aberto, apenas associado (pelo comando assign).

filepos(vararq) É uma função inteira que retorna a posição atual do indicador de posição corrente do arquivo. O primeiro componente de um arquivo é 0. Se o arquivo for texto, cada componente é um byte.

Observação: Note que é de responsabilidade do programador garantir a existência do arquivo nomeado por uma string. Se o programa tentar abrir um arquivo para leitura e o mesmo não existir, o programa abortará.

13.4 Exercícios

1. Faça uma rotina que tem como parâmetros dois vetores X e Y, de elementos reais, e um parâmetro inteiro n indicando a quantidade de números em V. Faça uma rotina que escreve os dados de X e Y em um arquivo texto da seguinte maneira: Na primeira linha é impresso o valor n. Em seguida imprima n linhas. Na primeira linha imprima os valores $X[1]$ e $Y[1]$ separados por um espaço em branco. Na segunda linha imprima os valores $X[2]$ e $Y[2]$, assim por diante até os n-ésimos elementos de X e Y.

2. Faça uma rotina que leia um valor n e dois vetores X e Y a partir de um arquivo texto, conforme o exercício anterior.

3. Um grupo de alunos do curso de *Algoritmos e Programação de Computadores* usa um compilador Pascal de linha de comando para compilar seus programas. Entretanto o editor de textos usado por eles não apresenta o número de linhas do programa, ficando difícil para encontrar a linha que contém o erro no programa. Faça um programa para ajudá-los. O programa deve ler o nome de um arquivo texto, *NomeArq* e deve ler o número de uma linha, *nlinha*. O programa deve imprimir na tela as linhas do programa: *nlinha-5, nlinha-4,...,nlinha,nlinha+1,...,nlinha+5*. Caso alguma destas linhas não exista, ela não deve ser apresentada. Mostre também o número de cada uma das linhas impressas (com 4 dígitos).

   Por exemplo, vamos supor que um compilador de linha tenha dado erro na linha 125 de um programa. Então usamos este programa para imprimir as linhas $120, \ldots, 130$. Como mostrado a seguir:

   ```
   120> var esq,dir,IndicePivo : integer;
   121>    Pivo       : real;
   122> begin
   123>    troca(v[inicio],v[(inicio+fim) div 2]);{escolha de um índice do pivô ini
   124>    Pivo := v[Inicio]; {Pivo fica na posição Inicio }
   125>    esq=Inicio+1; {primeiro elemento mais a esquerda, pulando o pivô}
   126>    dir:=Fim;      {primeiro elemento mais a direita}
   127>    while (esq<dir) do begin
   128>       while (esq<dir) and (v[esq] <= Pivo) do esq:=esq+1;
   129>       while (esq<dir) and (pivo <  v[dir]) do dir:=dir-1;
   130>       if esq<dir then begin
   ```

   Uma vez que sabemos qual a linha que tem erro (125), podemos dizer que a atribuição $\boxed{\text{esq=Inicio+1}}$ está errada, e deveríamos consertar para $\boxed{\text{esq:=Inicio+1}}$.
   Obs.: Note que muitas vezes um compilador apresenta um erro em uma linha, quando na verdade o erro ocorreu em alguma linha anterior.

4. Faça um programa que, dados dois nomes de arquivo texto: *NomeArq1, NomeArq2*, lê o texto que está no arquivo de nome NomeArq1 e grava em um arquivo de nome NomeArq2. O texto gravado deve ser o mesmo texto que foi lido, exceto que em minúsculas e sem acento. Este é um programa útil quando precisamos mandar um texto acentuado por uma rede que traduz os acentos em outros códigos, deixando o texto difícil de ser lido para quem o recebe.

5. Faça um programa que, dados dois nomes de arquivo texto: *NomeArq1, NomeArq2* e um inteiro k, $k \in \{0, \ldots, 25\}$, lê o texto que está no arquivo de nome NomeArq1 e grava em um arquivo texto de nome NomeArq2. O texto gravado deve ser o texto que foi lido criptografado com a cifra de Cesar, com deslocamento k. Considere apenas a criptografia das letras.

6. Faça um programa que, dados dois nomes de arquivo texto: *NomeArq1, NomeArq2* e um inteiro k, $k \in \{0, \ldots, 25\}$, lê o texto que está no arquivo de nome NomeArq1 e grava em um arquivo texto de nome NomeArq2. O texto gravado deve ser o texto que foi lido decriptografado com a cifra de Cesar, com deslocamento k.

7. (Busca de Padrão) Faça um programa que, dado um nome de um arquivo texto: *NomeArq*, e uma string: *padrao*, lê o arquivo texto e imprime (na tela) as linhas que contêm a string *padrao*. Além disso, o programa também

deve imprimir o número da linha impressa (sugestão: imprima o número da linha, a string $' > \ '$ e em seguida a linha do arquivo). No fim do programa, o programa deve imprimir quantas vezes o padrao apareceu no texto (considere repetições em uma mesma linha).

8. Um professor do curso de *Algoritmos e Programação de Computadores* tem um arquivo texto chamado *entrada.txt* contendo as notas dos alunos do seu curso. O arquivo tem várias linhas, uma linha para cada aluno. Cada linha tem o RA do aluno (string de 6 caracteres) e três notas (números reais) separadas por pelo menos um espaço em branco:

   RA P_1 P_2 L

   A média M é calculada da seguinte maneira: $M \leftarrow (2 \cdot P_1 + 3 \cdot P_2 + 3 \cdot L)/8$
   Se M é tal que $M \geq 5.0$, o aluno está *Aprovado*, caso contrário o aluno está de *Exame*. Faça um procedimento que lê o arquivo *entrada.txt* e gera um outro arquivo texto chamado *saida.txt* contendo uma linha para cada aluno. Cada linha deve ter o seguinte formato:

   RA M Resultado

   Onde M é a média do aluno e Resultado é a palavra *Aprovado* ou *Exame*, dependendo se o aluno passou ou ficou de exame. Os seguintes dois quadros ilustram o formato destes dois arquivos:

   entrada.txt

991111	7.5	6.5	8.5
992222	9.0	7.0	9.0
993333	3.5	4.0	5.0
		⋮	

   saida.txt

991111	7.5	Aprovado
992222	8.25	Aprovado
993333	4.25	Exame
		⋮

9. Um usuário necessita copiar um programa instalado no computador do seu escritório para o computador da sua casa. Infelizmente o programa usa muita memória e não cabe em apenas um disquete. Ajude este usuário, da seguinte maneira:

   (a) Faça um programa que lê um valor inteiro N e o nome de um arquivo, digamos *NomeArq*, e gera k arquivos com nomes: *NomeArq.1*, *NomeArq.2*, ..., *NomeArq.k*, assim, por diante, onde todos estes arquivos tem exatamente N bytes, exceto talvez o último que pode usar menos bytes.

   (b) Faça um programa que dado uma lista de nomes, digamos *NomeArq.1*, *NomeArq.2*, ..., *NomeArq.k* gera um novo arquivo contendo a concatenação destes arquivos, nesta ordem.

   Com isso, o usuário poderá "quebrar" seu programa em partes menores, gravar cada parte em um disquete, copiar cada arquivo no computador da sua casa e por fim concatená-los, obtendo o programa original.

10. **Os exercícios seguintes consideram as declarações feitas na seção 9.4 (exercício 1).** Faça uma rotina, GravaCadastroArquivo, que tem como parâmetros o cadastro e um nome (nome de arquivo). A rotina grava todos os registros do cadastro neste arquivo, que terá o nome especificado no parâmetro.

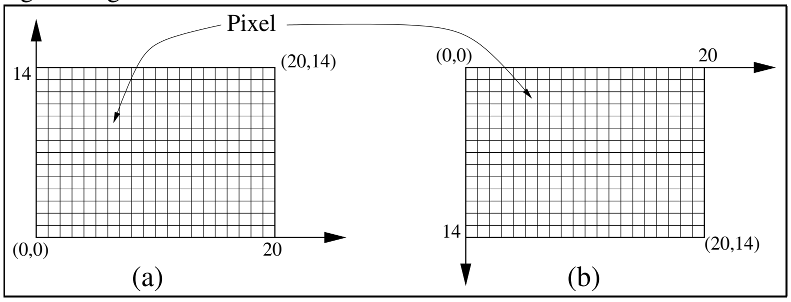
11. Faça uma rotina, LeCadastroArquivo, que tem como parâmetros o cadastro e um nome (nome de arquivo). A rotina lê o cadastro do arquivo com o nome especificado no parâmetro.

12. Faça uma rotina, InsereDeCadastroArquivo, que insere os funcionários que estão em um arquivo dentro do cadastro. Obs.: Os funcionários anteriores que estão no cadastro são permanecidos.

13. Faça as rotinas que fazem impressão/listagem, inserção, atualização de funcionarios, nos exercícios anteriores, mas agora sem uso do cadastro (vetor). Você deve fazer as operações diretamente no arquivo.

14 Figuras e Gráficos

Muitas vezes queremos vizualizar objetos que estão sendo representados no modelo computacional, como figuras, curvas, retas, polígonos, etc. Embora muitos objetos possam ter características tridimensionais, na maioria das vezes precisamos traduzi-los de maneira que possam ser visualizados de maneira bidimensional, uma vez que o vídeo é um dispositivo de apresentação bidimensional. Além disso, um vídeo de computador apresenta os objetos como uma matriz de pontos forçando a apresentar os objetos de maneira discretizada. Assim, vamos considerar que o menor elemento usado para apresentar um gráfico/figura é um ponto *(pixel)*. Deste modo, podemos considerar o desenho de uma reta no vídeo como sendo um conjunto de pontos, onde cada ponto da matriz tem coordenadas específicas nos dois eixos da matriz. Naturalmente, os objetos que iremos apresentar não precisam necessariamente estar definidos através de pontos, podendo ter uma representação interna contínua, como é o caso de funções de retas e curvas. Embora, no momento de sua apresentação na tela, esta deva ser discretizada para pontos, uma vez que o vídeo é uma matriz de pontos. Duas maneiras comuns de representação de uma tela gráfica com matriz de pontos por coordenadas são apresentadas nas duas figuras seguintes.



Para simplificar nossos programas, iremos usar a representação (a).

No processo de apresentação de objetos gráficos no vídeo, os pontos (representados por bits) são lidos de uma memória (memória de vídeo) e são apresentados no vídeo, em torno de 30 vezes por segundo.

Uma figura pode ser apresentada em um dispositivo gráfico usando cores, em geral usando uma quantidade limitada de memória para armazenar cada ponto de uma figura (e conseqüêntemente usando um número limitado de cores). Esta memória dependerá da quantidade de cores possíveis que cada ponto pode assumir. Se consideramos uma figura em preto e branco, cada ponto pode ser representado por apenas um bit. Para representar 256 cores (ou tonalidades diferentes), precisaremos de um byte para representar cada bit. Neste caso, uma figura usando 1024×1024 pontos, onde cada ponto assume 256 cores, usa uma memória de vídeo de pelo menos 1 Megabyte.

Uma maneira muito comum de se representar pontos coloridos é usando o formato RGB *(Red-Green-Blue)*, através de três valores que indicam a intensidade das cores Vermelha, Verde e Azul (lembrando que uma determinada cor pode ser obtida misturando quantidades específicas destas três cores).

14.1 Usando o formato PPM – Portable PixMap

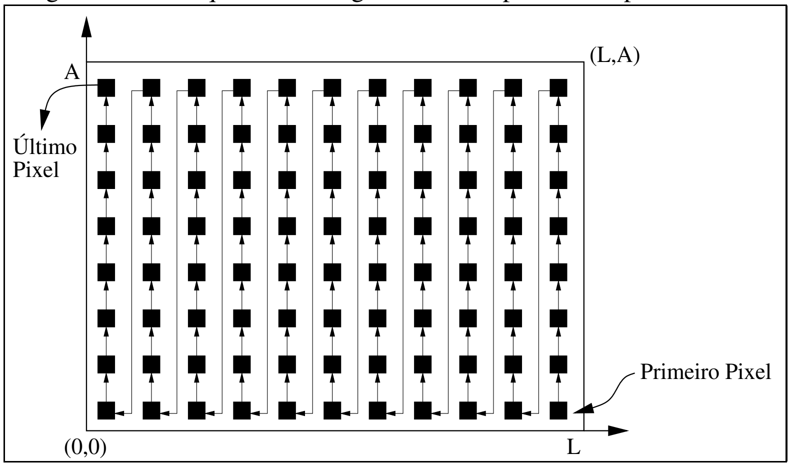
Para podermos trabalhar com gráficos de maneira independente de computador e sistema operacional, vamos usar um formato que pode ser visualizado em diversos ambientes computacionais. Vamos usar o formato PPM (Portable PixMap Format). Este formato representa uma matriz de pontos em um arquivo, usando o formato RGB, onde cada ponto colorido é apresentado por três bytes seguidos, representando a intensidade das cores vermelha, verde e azul. Cada byte apresenta um valor entre 0 e 255. Portanto, cada pixel pode ser uma de 2^{24} cores (naturalmente, sua visualização dependerá da qualidade do vídeo usado). Uma vez que temos esta matriz de pontos, podemos formar uma figura mudando as cores dos elementos desta matriz, ponto a ponto.

Existem diversos programas para visualização de figuras armazenadas em arquivos de diversos formatos gráficos, alguns destes livres ou a preços reduzidos *(Freeware e Shareware)* que podem visualizar figuras no formato PPM. Para o ambiente Linux, podemos usar o programa *GQview*, que pode ser obtido na página `http://gqview.netpedia.net/`. No ambiente Unix/X–Windows, podemos usar o program *xv*, disponível para a maioria das plataformas. Para o ambiente Windows podemos usar o programa *Irfan View32*, disponível livremente na rede `http://sunsite.auc.dk/tucows/g`

Por simplicidade, usaremos um tipo restrito para as figuras do formato PPM. Neste formato usamos um cabeçalho apresentando 4 valores descritos em formato texto *(P6 Largura Altura 255)*, separados por um espaço em branco ou em linhas diferentes. P6 são os caracteres ASCII 'P' e '6', e devem ser os primeiros dois bytes do arquivo. Em seguida devem vir a largura e altura da figura, *Largura* e *Algura*, descritos em números ASCII. Por fim, devem vir o três caracteres ASCII do número 255, indicando que a maior componente de uma cor (intensidade de Vermelho, Verde ou Azul) é 255. Caracteres neste cabeçalho que estejam em uma linha após o símbolo '#' não são considerados (comentários). Logo em seguida, como último byte deste cabeçalho, temos o byte de valor 10 (NewLine) e em seguida devem vir a seqüência de bytes que representam os pixels da figura. A cada três bytes (usando o sistema RGB) representamos um pixel (o primeiro byte para a intensidade de vermelho, o segundo para a intensidade de verde e o terceiro para a intensidade de azul), ao todo temos Largura×Altura pixels. Um exemplo de um cabeçalho para uma figura com 200 pixels de largura e 300 pixels de altura é apresentado no quadro seguinte:

```
P6 200 300 255
```

A seqüência de pixels gravados no arquivo PPM segue a ordem apresentada pelas setas na figura seguinte:



Para trabalhar com a matriz de pixels, vamos usar um tipo chamado *TipoTela*, onde cada pixel é do tipo *TipoCor*, em RGB, declarados no seguinte quadro.

```
const
  MAXPONTOS = 200;
type
  TipoCor = record
            red,green,blue      : Byte;
          end;
  TipoTela = array[0..MAXPONTOS−1,0..MAXPONTOS−1] of TipoCor;
```

Com estas declarações podemos gerar uma figura, usando uma matriz de MAXPONTOS×MAXPONTOS pixels. Além disso, podemos definir algumas cores para facilitar a atribuição de cores. Para definir a cor vermelha, podemos atribuir, em RGB, a intensidade máxima para vermelho (Red) e intensidade zero para as outras duas cores (Green e Blue). Podemos fazer o mesmo processo para definir as cores verde e azul. A cor branca é definida quanto temos a máxima intensidade (valor 255) para as componentes vermelha, verde e azul, e a cor preta é definida pela ausência de cores (valor zero). O quadro da página 141 mostra a definição destas cores e rotinas para iniciar a tela com cor branca e gravar a tela em um arquivo no formato PPM.

Obs.: Além do formato PPM, existem diversos outros formatos gráficos, sendo que alguns incorporam métodos de compressão de dados, como é o caso dos formatos *GIF* e *JPG*.

```
const
  MAXPONTOS = 200;
type
  TipoCor          = record
                      Red,Green,Blue : Byte;
                    end;
  TipoTela         = array[0..MAXPONTOS−1,0..MAXPONTOS−1] of TipoCor;
  TipoNomeArq           = string[100];
  TipoArquivoBinario = file of byte;
var Branco,Vermelho,Verde,Azul,Preto : TipoCor;
procedure InicializaAlgumasCores; {Cores: Branco, Preto, Vermelho, Verde, Azul}
begin  {Definicao das cores usadas no formato RGB (Red−Green−Blue) }
  Branco.Red:=255;  Branco.Green:=255; Branco.Blue:=255;
  Preto.Red:=0;     Preto.Green:=0;   Preto.Blue:=0;
  Vermelho.Red:=255; Vermelho.Green:=0; Vermelho.Blue:=0;
  Verde.Red:=0;     Verde.Green:=255;  Verde.Blue:=0;
  Azul.Red:=0;      Azul.Green:=0;    Azul.Blue:=255;
end;
procedure InicializaTela(var tela: tipotela); {Coloca todos os pixels como Branco}
var i,j     : integer;
begin {Obs.: O ponto (0,0) está no canto inferior esquerdo}
  for i:=0 to MAXPONTOS−1 do
    for j:=0 to MAXPONTOS−1 do
      tela[i,j] := Branco;
end; { inicializatela }
procedure DesenhaPonto(var tela : tipotela; x,y:integer; Cor:TipoCor);
begin
  tela[x,y] := Cor;  {Atribui o pixel (x,y) com Cor}
end;
procedure EscreveCabecalhoPPM(var arq : TipoArquivoBinario;largura,altura:integer);
var i : integer;  str1,str2,cabecalho : string[100];  b:byte;
begin
  str(largura,str1);  str(altura,str2); {Pegar a largura e altura como strings}
  cabecalho := 'P6 '+str1+' '+str2+' 255');
  for i:=1 to length(cabecalho) do begin
    b:=ord(cabecalho[i]);
    write(Arq,b);
  end;
  b:=10;  write(Arq,b); { Caracter para Pular linha }
end;
procedure GravaTela(var t:TipoTela; nomearq:tiponomearq); {Grava a tela em arquivo}
var Arq: TipoArquivoBinario;   lin,col : integer;
begin
  assign(Arq,NomeArq);  rewrite(Arq);
  EscreveCabecalhoPPM(Arq,MAXPONTOS,MAXPONTOS);
  for col:= MAXPONTOS−1 downto 0 do
    for lin:= 0 to MAXPONTOS−1 do begin
      write(Arq,t[lin,col].Red);   {Grava um pixel definido em RGB}
      write(Arq,t[lin,col].Green);
      write(Arq,t[lin,col].Blue);
    end;
  close(Arq);
end; { GravaTela }
```

14.2 Retas e Círculos

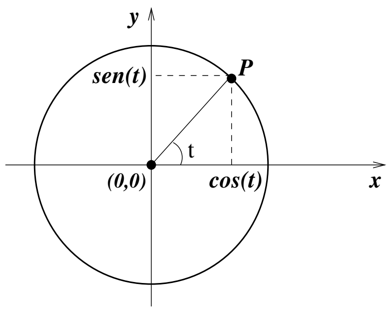
Uma vez que sabemos como imprimir um ponto em uma figura e gravá-la em um arquivo no formato PPM, vamos construir duas primitivas gráficas para construir desenhos mais complexos, as rotinas para desenhar um segmento de reta e um círculo. Para estas duas operações, vamos construir rotinas paramétricas, que definem os pontos do objeto através de um parâmetro.

Podemos definir os pontos de um segmento de reta R, ligando dois pontos $P_1 = (x_1, y_1)$ e $P_2 = (x_2, y_2)$, usando um parâmetro t no intervalo $[0, 1]$ como $R = \{(x(t), y(t)) : t \in [0, 1]\}$, onde

$$x(t) = x_1 + (x_2 - x_1) \cdot t, \qquad y(t) = y_1 + (y_2 - y_1) \cdot t\ .$$

Note que quando $t = 0$ obtemos o ponto P_1 e quando $t = 1$ obtemos o ponto P_2. Para desenhar o segmento de reta R, podemos "plotar" alguns pontos de R, cuja quantidade dependerá da dimensão da matriz de pontos, escolhendo valores de t distribuídos no intervalo $[0, 1]$.

Para desenhar um círculo C, com centro (c_x, c_y) e raio r, também vamos usar um parâmetro t que refletirá um ângulo no intervalo $[0, 2\pi)$. Primeiramente, vamos verificar a seguinte representação para um círculo com centro em $(0, 0)$ e raio 1.



Note que as coordenadas do ponto pertencente ao círculo, definido pelo ângulo t é o ponto $P = (\cos(t), \sin(t))$. Para definir um círculo de raio maior, basta multiplicar as coordenadas dos pontos pelo valor de r, e para que o círculo seja definido com centro (c_x, c_y), adicionamos os valores de c_x e c_y nas coordenadas correspondentes. Desta forma, obtemos a seguinte representação paramétrica para os pontos do círculo: $C = \{(x(t), y(t)) : t \in [0, 2\pi)\}$ onde

$$x(t) = c_x + r \cdot \cos(t), \qquad y(t) = c_y + r \cdot \sin(t)\ .$$

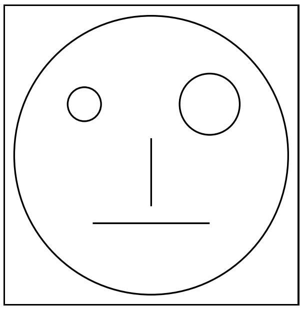
Agora podemos definir uma figura ou objeto gráfico, mais elaborado, como um conjunto de objetos primitivos (retas e círculos). Vamos definir este objeto mais elaborado como o tipo *TipoFigura* da seguinte maneira:

```
const MAXOBJETOS = 200; {Quantidade máxima de objetos que uma figura pode ter}
type
  TipoCor    = record
            red,green,blue : Byte;
          end;
  TipoForma  = (FormaCirculo,FormaReta);
  TipoObjeto = record
            Cor              : TipoCor;
            case Forma       : TipoForma  of
              FormaCirculo : (centrox,centroy:integer; Raio:Real);
              FormaReta       : (p1x,p1y,p2x,p2y:integer);
          end;
  TipoFigura          = record
                nobjetos : 0..MAXOBJETOS; {quantidade de objetos, armazenados em vetor}
                objeto  : array[1..MAXOBJETOS] of TipoObjeto;
              end;
```

A seguir, apresentamos um programa, contendo os procedimentos para desenhar um segmento de reta e um círculo, para gravar uma figura de um desenho de um rosto construído com círculos e retas e gravado em um arquivo no formato PPM.

```
program ProgFigura;
const
  MAXPONTOS  = 200;
  MAXOBJETOS = 200;
type
  TipoCor    = record
              red,green,blue : Byte;
            end;
  TipoTela              = array[0..MAXPONTOS−1,0..MAXPONTOS−1] of TipoCor;
  TipoForma  = (FormaCirculo,FormaReta);
  TipoObjeto = record
              Cor               : TipoCor;
              case Forma          : TipoForma  of
                FormaCirculo : (centrox,centroy:integer;Raio:Real);
                FormaReta          : (p1x,p1y,p2x,p2y:integer);
            end;
  TipoFigura              = record
                    nobjetos : 0..MAXOBJETOS;
                    objeto  : array[1..MAXOBJETOS] of TipoObjeto;
                  end;
  TipoNomeArq              = string[100];
  TipoArquivoBinario = file of byte;
var Branco,Vermelho,Verde,Azul,Preto : TipoCor;
{ Incluir as seguintes rotinas }
{procedure InicializaTela(var tela        : tipotela);}
{procedure DesenhaPonto(var tela : tipotela; x,y:integer; cor:TipoCor);}
{procedure GravaTela(var t : TipoTela; nomearq: tiponomearq);}
procedure DesenhaReta(var tela : tipotela; x1,y1,x2,y2:integer; cor:TipoCor);
var t,delta,maxit,x,y : real; {representacao parametrica}
begin
  delta := 1/MAXPONTOS;
  x:=x1;   y:=y1;   t:=0;
  while t<=1 do begin {0 <= t <= 1}
    x:=x1+(x2−x1)∗t;
    y:=y1+(y2−y1)∗t;
    DesenhaPonto(tela,round(x),round(y),cor);
    t:=t+delta;
  end;
end;
procedure DesenhaCirculo(var tela : tipotela; cx,cy:integer;raio:real; cor:TipoCor);
var t,delta,maxit,x,y,DoisPi : real; {representacao parametrica}
begin
  DoisPi := PI ∗ 2;
  delta := 1/MAXPONTOS;
  t:=0;
  while t<=DoisPi do begin { 0 <= t <= DoisPi}  {Usando Coordenadas Polares}
    x:=cx + raio ∗ cos(t);
    Y:=cy + raio ∗ sin(t);
    DesenhaPonto(tela,round(x),round(y),cor);
    t:=t+delta;
  end;
end;
```

```
procedure DesenhaObjeto(var tela : TipoTela; obj : TipoObjeto);
begin
  case Obj.Forma of
    FormaCirculo  : DesenhaCirculo(tela, obj.centrox, obj.centroy, obj.raio, obj.cor);
    FormaReta     : DesenhaReta(tela, obj.p1x, obj.p1y, obj.p2x, obj.p2y, obj.cor);
  end; { case }
end;
procedure DesenhaFigura(var t : TipoTela; var Fig:TipoFigura);
var i:integer;
begin
  for i:=1 to Fig.nobjetos do DesenhaObjeto(t,Fig.objeto[i]);
end;
procedure EscreveString(var arq : TipoArquivoBinario;str:string);
var i : integer;
begin
  for i:=1 to length(str) do write(Arq,ord(str[i]));
end;
var t      : tipotela;
  face : TipoFigura;
begin
  {Definicao de algumas cores usando o formato RGB (Red-Green-Blue) }
  Branco.Red:=255;   Branco.Green:=255; Branco.Blue:=255;
  Preto.Red:=0;      Preto.Green:=0;    Preto.Blue:=0;
  Vermelho.Red:=255; Vermelho.Green:=0; Vermelho.Blue:=0;
  Verde.Red:=0;      Verde.Green:=255;  Verde.Blue:=0;
  Azul.Red:=0;       Azul.Green:=0;     Azul.Blue:=255;
  inicializatela(t);
  {cabeca}
  face.objeto[1].Forma := FormaCirculo;  face.objeto[1].cor := Preto;
  face.objeto[1].centrox := 100;         face.objeto[1].centroy := 100;
  face.objeto[1].raio := 50;
  {olho esq}
  face.objeto[2].Forma := FormaCirculo;  face.objeto[2].cor := Azul;
  face.objeto[2].centrox := 80;          face.objeto[2].centroy := 120;
  face.objeto[2].raio := 6;
  {olho dir}
  face.objeto[3].Forma := FormaCirculo;  face.objeto[3].cor := Verde;
  face.objeto[3].centrox := 120;         face.objeto[3].centroy := 120;
  face.objeto[3].raio := 10;
  {nariz}
  face.objeto[4].Forma := FormaReta;     face.objeto[4].cor := Preto;
  face.objeto[4].p1x := 100;             face.objeto[4].p1y := 110;
  face.objeto[4].p2x := 100;             face.objeto[4].p2y := 90;
  {boca}
  face.objeto[5].Forma := FormaReta;     face.objeto[5].cor := Vermelho;
  face.objeto[5].p1x := 80;              face.objeto[5].p1y := 80;
  face.objeto[5].p2x := 120;             face.objeto[5].p2y := 80;

  face.nobjetos :=5; {face foi definido com 5 objetos graficos}
  DesenhaFigura(t,face);
  GravaTela(t,'saida.ppm');
end.
```



14.3 Exercícios

1. O formato PGM (Portable Grayscale pixMap) é muito parecido com o formato PPM. Neste formato podemos representar figuras monocromáticas, com intensidade nos pixels. I.e., todos os pixels tem apenas uma cor, mas podem ter intensidade que pode ir de 0 a 255. A seguir, listamos as diferenças/semelhanças nos formatos destes dois arquivos:

   (a) O cabeçalho é descrito da mesma maneira que o cabeçalho de um arquivo PPM, mas em vez de 'P6', você deve colocar 'P5'.

   (b) A seqüência de pixels segue a mesma ordem que no formato PPM.

   (c) Cada pixel usa apenas 1 byte (em vez de 3). O valor deste byte indica a intensidade do pixel. O valor 255 para o branco e 0 para preto. Os valores entre 0 e 255 indicarão uma intensidade intermediária.

   Faça um programa que desenha a figura apresentada na página 144, mas em formato PGM.

2. Faça outras primitivas gráficas, além do segmento de reta e o círculo: Retângulo, Retângulo Preenchido (todo preenchido com uma cor), Círculo Preenchido, Poligono fechado de n pontos.

3. Faça um programa contendo uma rotina que *plota* uma função $f(x)$ em um arquivo de formato PPM, para valores de x em um intervalo $[x_1, x_2]$. A rotina deve ter o seguinte cabeçalho:

   procedure PlotaFuncao(f: TipoFuncaoReal; x_1, x_2, y_1, y_2:real; NomeArquivo:TipoString);

   onde TipoFuncaoReal foi definido como:

   type TipoFuncao = function(x:real):real;

   Com isso, iremos passar a função a plotar como parâmetro. Para plotar os pontos $(x, f(x))$, discretize os valores de x no intervalo $[x_1, x_2]$, imprima os pontos $f(x)$ que caem no intervalo $[y_1, y_2]$. Além disso, imprima as retas dos eixos $x = 0$ e $y = 0$, caso estas estejam nos intervalos de impressão.

   Faça uma imagem de tamanho $L \times C$ pontos (com digamos $L = 300$ e $C = 300$). Naturalmente você terá que fazer uma reparametrização, de maneira que o ponto $(0, 0)$ na matriz de pontos seja equivalente a posição (x_1, y_1) e o ponto (L, C) na matriz de pontos seja equivalente a posição (x_2, y_2).

4. Faça uma rotina que tenha como parâmetros, um vetor de valores reais positivos com n elementos. Faça um programa que gere gráficos de barras, histograma e tipo torta, de acordo com a escolha do usuário.

5. A figura de uma letra pode ser armazenada em uma matriz de tamanho 8×8. Gere matrizes para algumas letras, atribuindo os valores dos pontos através de um programa (vc pode usar as rotinas de reta e círculo para isso). Uma vez que você gerou estas letras, faça uma rotina que tem como parâmetro uma posição (bidimensional) e uma letra. A rotina deve *imprimir* a matriz que forma a letra na tela de pontos. Isto permitirá que possamos escrever na figura.

6. Faça uma rotina que tem como parâmetro os nomes de dois arquivos e um valor real positivo, α. A rotina lê o arquivo do primeiro nome, que é uma figura no formato PPM e gera outro arquivo que é a mesma figura, mas com tamanho reparametrizado de um fator α.

15 Ponteiros

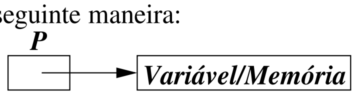
Ponteiros *(pointers)* ou apontadores são variáveis que armazenam um endereço de memória (e.g., endereço de outras variáveis).

Na linguagem Pascal cada ponteiro tem um tipo, podendo ser um ponteiro para um tipo pré-definido da linguagem Pascal ou um tipo definido pelo programador. Assim, podemos ter ponteiros para *integer*, ponteiro para *real*, ponteiro para *TipoAluno* etc.

Quando um ponteiro contém o endereço de uma variável, dizemos que o ponteiro está "apontando" para essa variável.

Como um ponteiro é apenas um endereço de memória, não é preciso muitos bytes para sua definição, em geral são usados 4 bytes para isso.

Vamos representar um ponteiro P da seguinte maneira:



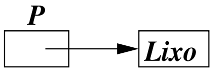
Neste caso, a variável–Ponteiro P, lado esquerdo, está apontando para a variável/memória que está do lado direito. Internamente, a variável P contém o endereço da variável apontada.

- **Declaração:**

Para declarar um ponteiro de um certo tipo, usamos a seguinte sintaxe:

var Lista_Ident_Pont:ˆ*TipoPonteiro;*

Ao declarar um ponteiro P, o endereço inicial contido nesta variável deve ser considerado como *lixo*, i.e.:



- **Atribuindo um endereço para um ponteiro:**

Vamos supor que temos uma variável, V, e queremos que um ponteiro, P, (ponteiro para **integer**) aponte para esta variável. Para isso, usamos o operador @ para obter o endereço de V, da seguinte maneira:

$$P := @V;$$

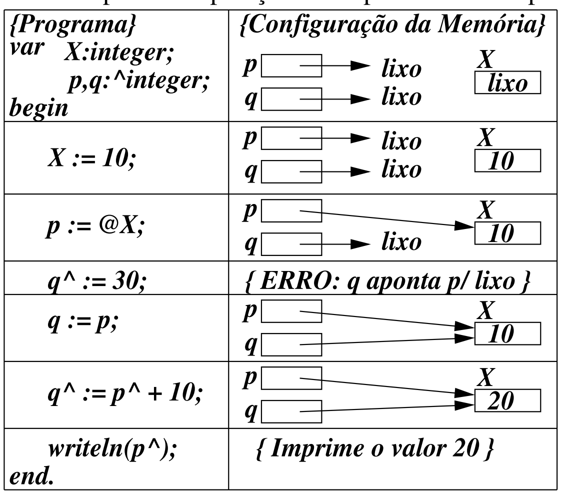
Um ponteiro também pode receber o conteúdo de outro ponteiro. Além disso, existe um endereço especial, chamado **nil** (nulo), que serve para dizer que é um endereço nulo e não teremos nenhuma variável neste endereço de memória. Este endereço é útil para dizer que o ponteiro ainda não tem nenhum endereço válido.

$$P := \mathbf{nil};$$

- **Trabalhando com a memória apontada pelo ponteiro:**

Para acessar a memória que o ponteiro P está apontando, usamos o operador ˆ, com a sintaxe $\boxed{P\hat{}}$. Naturalmente P deverá ter um endereço válido, i.e., P deve estar apontando para uma variável ou para uma memória de tipo compatível.

Na figura seguinte apresentamos exemplos das operações com ponteiros e a representação de cada comando.



O comando $\boxed{\text{q\^{}:=30;}}$ causa um erro no programa; assim, vamos supor que este comando é apenas ilustrativo e não afeta os demais comandos.

Exemplo 15.1 *Considere dois vetores de "alunos" (registros) usando o tipo TipoAluno (veja página 93), que temos que listar ordenados por nome, digamos pelo algoritmo QuickSort (veja página 113). Os dois vetores não devem ser alterados, nem mesmo na ordem dos seus elementos.*

Este problema poderia ser resolvido facilmente usando um terceiro vetor para conter a união dos dois primeiros e em seguida ordenamos este terceiro vetor por nome. Desta maneira manteríamos os dois vetores originais intactos. Entretanto, se cada registro do tipo TipoAluno for muito grande e a quantidade de alunos nos dois vetores também for grande, teríamos dois problemas:

1. *A declaração do terceiro vetor deve usar uma memória grande o suficiente para acomodar a união dos dois primeiros vetores.*
2. *Os algoritmos de ordenação, baseados em troca de elementos, iriam fazer muitas transferências de bytes (lembrando que para trocar dois elementos fazemos 3 atribuições de variáveis, veja rotina troca na página 69).*

Uma maneira mais econômica e rápida para resolver este problema é usar o terceiro vetor como um vetor de ponteiros para alunos. I.e., cada elemento do terceiro vetor aponta para um elemento dos dois vetores. Note que cada elemento do terceiro vetor deve gastar em torno de 4 bytes, já que é apenas um endereço, gastando menos memória que do modo anterior. Na figura 40(a) apresentamos a representação dos três vetores com os elementos do terceiro vetor apontando para os elementos dos dois vetores originais. Na figura 40(b) apresentamos o terceiro vetor já ordenado. No programa da página 148 apresentamos a função de comparação e a rotina de troca de elementos usadas para ordenar o terceiro vetor.

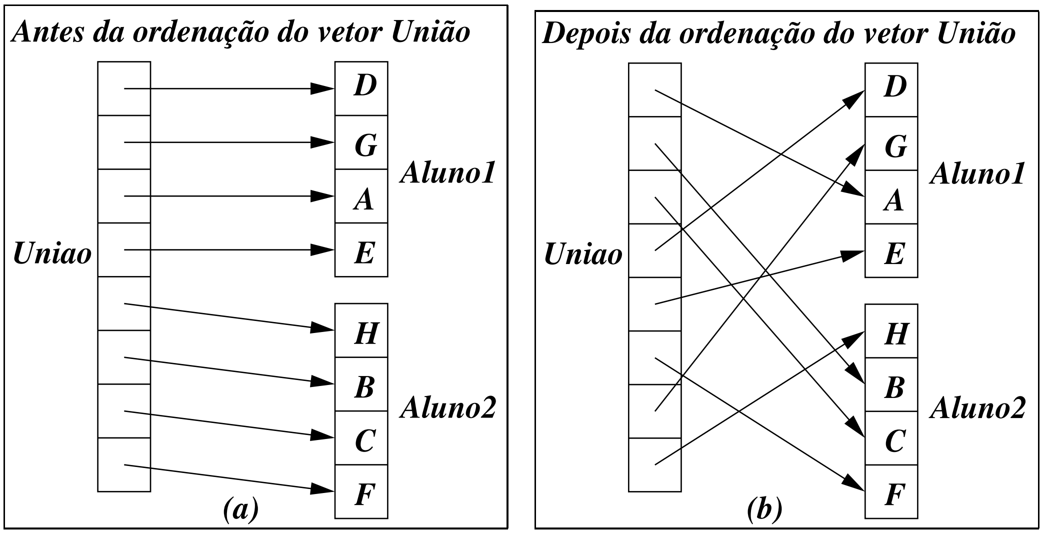


Figura 40: Ordenação de dois vetores, usando um terceiro vetor de ponteiros.

```
program OrdenaDoisVetores;
const MAXALUNOS       = 100;
type
  { Inserir declarações de TipoAluno }
  TipoApontadorAluno = ^TipoAluno;  {Apontador de TipoAluno}
  TipoVetorAluno = array[1..MAXALUNOS] of TipoAluno;
  TipoVetorApontadorAluno = array[1..2*MAXALUNOS] of TipoApontadorAluno; {Vetor de Apontadores}
procedure LeVetorAluno(var V : TipoVetorAluno;var n:integer);
var i : integer;
begin
  write('Entre com a quantidade de alunos a ler: ');  readln(n);
  for i:=1 to n do begin
    write('Entre com o nome do aluno: '); readln(V[i].nome);
    { ... leia outros atributos do aluno ... }
  end;
end; { LeVetorAluno }
procedure TrocaApontadorAluno(var ApontAluno1,ApontAluno2 : TipoApontadorAluno);
var ApontAux : TipoApontadorAluno;
begin
  ApontAux:=ApontAluno1;
  ApontAluno1:=ApontAluno2;
  ApontAluno2:=ApontAux;
end; { TrocaApontadorAluno }
function ComparaApontadorAluno(var ApontAluno1,ApontAluno2:TipoApontadorAluno):char;
begin
  if (ApontAluno1^.nome > ApontAluno2^.nome) then ComparaApontadorAluno:='>'
  else if (ApontAluno1^.nome < ApontAluno2^.nome) then ComparaApontadorAluno:='<'
  else ComparaApontadorAluno:='=';
end;
var Aluno1,Aluno2           : TipoVetorAluno;
  Uniao                 : TipoVetorApontadorAluno;
  i,n1,n2,nuniao          : integer;
begin
  LeVetorAluno(Aluno1,n1);
  LeVetorAluno(Aluno2,n2);
  nuniao := 0;
  for i:=1 to n1 do begin
    nuniao := nuniao + 1;
    Uniao[nuniao] := @Aluno1[i];
  end;
  for i:=1 to n2 do begin
    nuniao := nuniao + 1;
    Uniao[nuniao] := @Aluno2[i];
  end;
  {Chamada da rotina QuickSort usando rotina de comparação: ComparaApontadorAluno
   e rotina de troca de elementos: TrocaApontadorAluno }
  QuickSortApontAluno(Uniao,nuniao,ComparaApontadorAluno);
  for i:=1 to nuniao do begin
    writeln(Uniao[i]^.nome,Uniao[i]^.rg,{ ... } Uniao[i]^.cr);
  end;
end.
```

Exercício 15.1 *Um sistema da Universidade UNICOMP trabalha com um cadastro muito grande, contendo o dados de alunos (veja declaração do tipo TipoAluno na página 93). Uma das rotinas do sistema faz uma seqüência de seleções realizadas pelo usuário no cadastro de alunos, até que o usuário esteja satisfeito. A cada iteração com o usuário a rotina imprime o número de alunos nas condições. Em cada iteração, o usuário pode restringir o atual conjunto com os seguintes tipos de restrição:*

1. *Retringir o atual conjunto para alunos com idade superior a ID, onde ID é um valor lido.*
2. *Retringir o atual conjunto para alunos com idade inferior a ID, onde ID é um valor lido.*
3. *Retringir o atual conjunto para alunos com CR superior a Cr, onde Cr é um valor lido.*

Quando o usuário estiver satisfeito, a rotina deve imprimir os nomes dos alunos selecionados. Implemente esta rotina usando um vetor auxiliar de ponteiros para alunos, sendo que este vetor de ponteiros deve ser atualizado a cada iteração com o usuário. O cadastro de alunos não deve ser modificado, apenas o vetor de ponteiros. Note que cada nova seleção é feita (possivelmente) em um conjunto de dados menor.

15.1 Alocação Dinâmica de Memória

A linguagem Pascal permite que possamos "pedir" mais memória que a usada para o sistema operacional. Este nos retornará o endereço da memória que nos foi alocada ou um aviso que não foi possível alocar a memória requisitada. A memória a ser requisitada deve ter um tipo associado a ela e para receber o endereço desta nova memória, devemos usar um ponteiro do mesmo tipo da memória sendo requisitada. Por exemplo, usamos um ponteiro para *TipoAluno* para obter memória do tipo *TipoAluno*.

O comando para obter memória é através do comando **new** da seguinte maneira:

new(p);

Este comando aloca uma memória que será apontada pelo ponteiro p. Caso o sistema operacional não consiga alocar esta memória, o ponteiro p retornará com valor **nil**. Esta nova memória não é memória de nenhuma variável, mas sim uma memória nova que pode ser usada com o mesmo tipo daquele declarado para o ponteiro p. Além disso, qualquer memória que tenha sido obtida através do comando **new** pode ser liberada novamente para o sistema operacional, que poderá considerá-la para futuras alocações de memória. O comando para liberar uma memória obtida por alocação dinâmica, apontada por um apontador p, é através do comando **dispose** da seguinte maneira:

dispose(p);

A figura seguinte apresenta um programa contendo alguns comandos usando alocação dinâmica e a situação da memória após cada comando. Alguns dos comandos causam erro de execução e foram colocados apenas para fins didáticos, vamos supor que estes não afetam os demais comandos.

program exemplo; *type TipoRegistro = record* *nome: string[50]* *idade:integer;* *end;* *var p,q:^TipoRegistro;* *R: TipoRegistro;* *begin*	***CONFIGURAÇÃO DA MEMÓRIA APÓS COMANDOS*** *p* → *lixo* *R=* [*lixo* | *lixo*] *q* → *lixo*
new(p); *R.nome := 'Carlos';* *R.idade := 50;*	*p* → [*lixo* | *lixo*] *R=* [*Carlos* | *50*] *q* → *lixo*
writeln(p^.nome);	*PROBLEMA: p^.nome ainda não foi inicializado*
p^.nome := 'José'; *p^.idade := 10;*	*p* → [*José* | *10*] *R=* [*Carlos* | *50*] *q* → *lixo*
p:=nil;	*PROBLEMA: Memória alocada antes foi perdida* *p* → *nil* [*José* | *10*] *q* → *lixo* *R=* [*Carlos* | *50*]
new(q); *q^ := R;*	*p* → *nil* [*José* | *10*] *q* → [*Carlos* | *50*] *R=* [*Carlos* | *50*]
writeln(p^.nome);	*ERRO: p não aponta para memória*
p := @R; *q^.idade := 60;*	*p* → R [*José* | *10*] *q* → [*Carlos* | *60*] *R=* [*Carlos* | *50*]
writeln(q^.nome,' ',q^.idade);	*{ Imprime 'Carlos 60' }*
dispose(p);	*ERRO: p não aponta para memória obtida por alocação dinâmica.*
dispose(q); *end.*	*p* → R [*José* | *10*] *q* → *lixo* *R=* [*Carlos* | *50*]

15.2 Listas Ligadas

Uma das grandes restrições de se representar um conjunto de elementos por vetores é a limitação imposta pelo número de elementos na declaração do vetor. O número de elementos declarado no tamanho de um vetor deve ser um valor fixo e nos traz dois inconvenientes:

1. Devemos assumir uma declaração do vetor usando muitos elementos para que não tenhamos problemas quando precisarmos manipular muitos elementos.
2. Uma declaração de um vetor com muitos elementos pode causar desperdício de memória. Note que se todos os programas usassem vetores com tamanhos muito grandes, provavelmente teríamos poucos programas residentes na memória do computador.

Um vetor pode representar vários elementos de maneira simples e direta. Isto é possível pois seus elementos estão em posições contíguas de memória, embora não seja possível estender esta memória quando precisarmos inserir mais elementos que o definido no tamanho do vetor.

Usando alocação dinâmica podemos representar um conjunto de dados usando uma quantidade de memória proporcional ao número de elementos sendo representados no conjunto. Isto pode ser feito fazendo uma interligação entre os elementos do conjunto usando tipos recursivos de dados. Um elemento de um *tipo recursivo* é um registro de um tipo onde alguns dos campos são apontadores para o mesmo tipo. A primeira vista parece estranho, uma vez que estamos definindo um campo que referencia um tipo que ainda estamos definindo. Vamos chamar estes campos de *campos de ligação*. Estes campos de ligação nos permitirão fazer um encadeamento entre os elementos de maneira a poder percorrer todos os elementos de um conjunto através de apenas um ponteiro inicial. Além disso, poderemos inserir novos elementos neste encadeamento, usando alocação dinâmica.

Na linguagem Pascal podemos definir tipos recursivos das seguintes maneiras:

```
type
    TipoRegistro = record
        { Campos com dados do elemento }
        ListaDeCamposDeLigação: ^TipoRegistro;
    end;
```

```
type
    TipoApontadorRegistro = ^TipoRegistro;
    TipoRegistro = record
    { Campos com dados do elemento }
    ListaDeCamposDeLigação:TipoApontadorRegistro;
    end;
```

Um importante exemplo de um tipo recursivo é a estrutura de dados chamada *lista ligada*, que assim como o vetor, representará uma seqüência de n elementos.

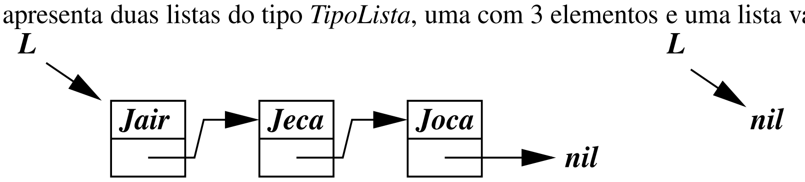
Dado um conjunto de elementos C, uma lista de elementos de C pode ser definida recursivamente como:

1. Uma seqüência vazia.
2. Um elemento de C concatenado a uma lista de elementos C.

Para representar listas ligadas em Pascal podemos usar de registros com apenas um campo de ligação, que indicará o próximo elemento da seqüência. Usaremos um ponteiro para representar toda esta seqüência. Caso este ponteiro esteja apontando para **nil**, estaremos representando a seqüência vazia. Vamos ver um exemplo de declaração de lista ligada:

```
type
        TipoElementoListaAluno = record
                NomeAluno: TipoString;
                proximo: ^TipoElementoListaAluno
        end;
        TipoLista = ^TipoElementoListaAluno
var Lista: TipoLista;
```

A figura seguinte apresenta duas listas do tipo *TipoLista*, uma com 3 elementos e uma lista vazia:



Vamos chamar o elemento (ou ponteiro) que representa o início da lista como *cabeçalho* ou *cabeça* da lista e a lista começando do segundo elemento em diante, também uma lista, chamada de *cauda* da lista.

Primeiramente vamos descrever uma rotina para listar os elementos contidos em uma lista ligada. Para isso, usaremos um outro ponteiro auxiliar para percorrer os elementos da lista. Inicialmente este ponteiro auxiliar começará no cabeçalho. Uma vez que tenhamos processado o elemento apontado por este ponteiro auxiliar, iremos para o próximo elemento, usando o campo *proximo* do elemento corrente. Isto é feito até que o ponteiro auxiliar chegue no fim da lista, i.e., quando o ponteiro auxiliar tiver valor **nil**. A seguir apresentamos uma rotina para imprimir os nomes de alunos de uma lista ligada, bem como os tipos usados.

```
type
    TipoApontElementoLista = ^TipoElementoLista;
    TipoElementoLista = record
                Aluno: TipoAluno;
                proximo: TipoApontElementoLista;
    end;
    TipoLista = TipoApontElementoLista;

procedure ImprimeAlunosLista(Lista: TipoLista);
var Paux:TipoApontElementoLista;
begin
   Paux := Lista;
   while (Paux<>nil) do begin
      writeln(Paux^.nome,' ',Paux^.RA,' ',Paux^.CR);
      Paux := Paux^.proximo;
   end;
end;
```

Vamos ver como podemos inserir um novo elemento no início da lista L. Primeiramente vamos alocar memória para o novo elemento, ler os dados nesta memória e por fim inserir o elemento no início da lista. Para esta última parte devemos tomar especial cuidado na ordem das operações para que nenhum elemento da lista seja perdido. Podemos dividir esta operação nos seguintes passos:

1. Alocar memória para o novo elemento, digamos *Paux*.
2. Ler os dados na memória apontada por *Paux*.
3. Como *Paux* será o primeiro elemento da lista L, os atuais elementos da lista devem seguir *Paux*. Assim, podemos fazer *Paux^.proximo* apontar para o primeiro elemento da lista atual.
4. Atualizar o cabeçalho L para que fique apontando para a posição *Paux*.

A figura 41 apresenta uma rotina para inserção de um elemento no início da lista e a configuração da memória após cada comando.

Note que mesmo que a lista L represente uma lista vazia, a lista resultante conterá uma lista com exatamente um elemento.

A idéia para desenvolver uma rotina para remover o primeiro elemento da lista segue a mesma idéia. A rotina *RemovePrimeiroAlunoLista*, descrita na página 153, apresenta tal operação.

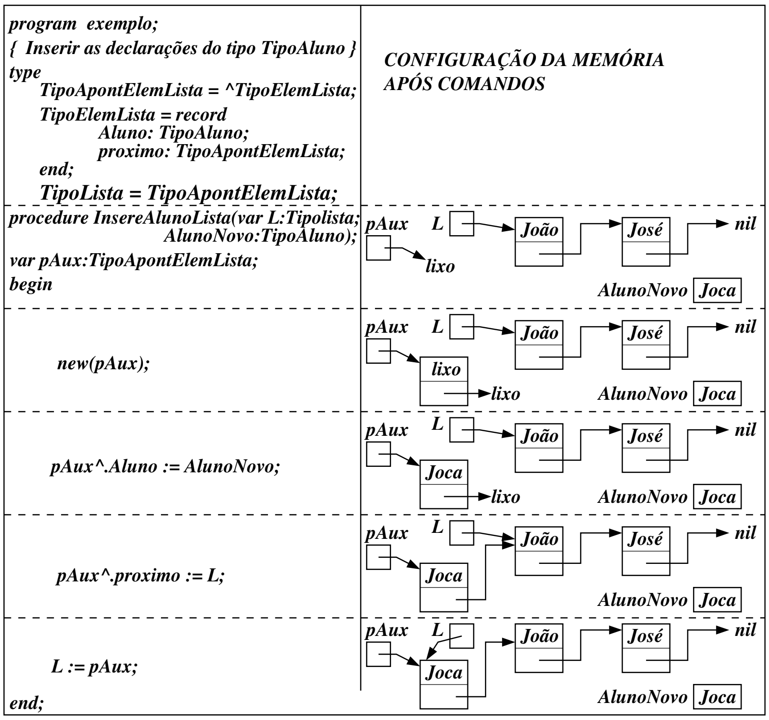


Figura 41: Inserção de elemento em uma lista ligada.

```
function RemovePrimeiroAlunoLista(var L:TipoLista; var AlunoRemovido:TipoAluno):boolean;
var Paux:TipoApontElemLista;
begin
    if L=nil then RemovePrimeiroAlunoLista := false
    else begin
        Paux:=L;   {Guarda o primeiro elemento antes de atualizar L}
        L:= L^.proximo;  {Atualiza L para começar no segundo elemento}
        AlunoRemovido := Paux^.Aluno;   {Copia os dados do elemento para o }
        dispose(Paux); {parâmetro de retorno, e libera a memória alocada para o elemento}
       RemovePrimeiroAlunoLista:=true;
    end;
end;
```

Exemplo 15.2 *O procedimento seguinte inverte a ordem dos elementos da lista.*

```
procedure InverteListaAluno(var p: TipoLista);
var q,r,s : TipoApontElemLista;
begin
  q := nil;
  r := p;
  while r<>nil do begin
    s := r^.proximo;
    r^.proximo := q;
    q := r;
    r := s
  end;
  p := q
end;
```

Listas × Vetores

Como tanto listas como vetores são usados para representar uma seqüência ordenada de elementos, vamos ver algumas diferenças entre estas duas estruturas de dados.

	Vetores	Listas simplesmente ligadas
Número máximo de elementos	restrito ao tamanho declarado no vetor.	Restrito a quantidade de memória disponibilizada pelo sistema operacional.
Acesso ao i-ésimo elemento	Direta.	A partir do primeiro elemento até chegar ao i-ésimo.
Inserção no início	É necessário deslocar todos os elementos de uma posição.	Em tempo constante.
Processamento de trás para frente	Processamento direto na ordem.	Necessita mais processamento ou memória.

15.3 Recursividade e Tipos Recursivos

Muitas vezes o uso de rotinas recursivas para manipular dados com tipos recursivos nos possibilita gerar rotinas simples e de fácil entendimento. Esta facilidade se deve pelo tipo ser recursivo, pois ao acessar um ponteiro para um tipo recursivo, digamos P, podemos visualizá-lo como uma variável que representa todo o conjunto de elementos, digamos C, acessíveis por ele. Do mesmo jeito, se um elemento deste conjunto contém um campo de ligação, digamos P', este representa um subconjunto $C' \subset C$ com as mesmas características de P. Isto permite que possamos chamar a mesma rotina que foi definida para o ponteiro P recursivamente para P' (projeto indutivo). A figura 42 ilustra o tipo recursivo para o caso de lista ligada. O primeiro elemento apontado pelo ponteiro L representa a lista inteira. O ponteiro *L^.proximo* representa uma sublista da lista original.

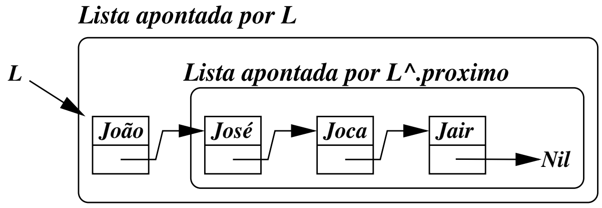


Figura 42: Inserção no fim de uma lista ligada usando recursividade.

Exemplo 15.3 *Este importante exemplo mostra como podemos fazer uso de recursividade e passagem de parâmetros por referência para inserir um elemento no fim de uma lista.*

```
procedure InsereAlunoFimLista(var L:TipoLista; var AlunoNovo:TipoAluno);
begin
  if L=nil then begin {Chegou no fim da lista}
    new(L);          {Aloca a memória para ser apontada por L}
    L^.Aluno := AlunoNovo;   {Insere o último elemento}
    L^.proximo := nil;       {Ultimo elemento deve ter campo proximo igual a nil}
  end
  else InsereAlunoFimLista(L^.proximo,AlunoNovo);  {Insere na lista definida por L^.proximo}
end;
```

Primeiramente verifique que a chamada desta rotina para uma lista vazia (L aponta para nil) insere de fato um novo elemento, fazendo com que a lista fique com um elemento. É importante observar que o ponteiro L retorna atualizado, uma vez que o parâmetro L foi declarado com passagem de parâmetro por referência (usando **var***).*

*Agora observe o caso que a lista contém pelo menos um elemento. Neste caso, a primeira chamada recursiva deve fazer uma nova chamada recursiva para que o elemento seja inserido no fim da lista L^.proximo. Note que L^.proximo é um apontador e também representa uma lista. Assim, esta segunda chamada recursiva deve inserir o elemento no fim desta sublista como desejamos. Note que mesmo que L^.proximo seja uma lista vazia (L^.proximo=*__nil__*), este voltará atualizado com o novo elemento.*

A observação dos seguintes itens pode ajudar no desenvolvimento de rotinas para estruturas dinâmicas, como listas ligadas e árvores.

1. Desenvolva as rotinas simulando e representando a estrutura de dados de maneira gráfica, e.g., desenhadas em um papel. Para cada operação atualize seu desenho.
2. Tome cuidado na ordem das operações efetuadas pela rotina. Verifique se após a execução de um passo, não há memória dinâmica perdida.
3. Desenvolva inicialmente a rotina considerando uma estrutura contendo vários elementos. Verifique se a rotina funciona para a estrutura com poucos elementos, sem elementos ou com 1 elementos. Caso necessário, adapte sua rotina.
4. Aproveite a volta da recursão, para possível processamento posterior do elemento.
5. Se for o caso, faça a atualização dos campos de ligação por passagem de parâmetros por referência.

Exercício 15.2 *Um certo programa manipula cadeias de caracteres que podem ser de tamanhos bem diversos. Algumas muito grandes outras bem pequenas. Por ter cadeias de caracteres muito longas, não é aconselhado usar o tipo string[<número de caracteres>] pois isto fixaria o tamanho da maior cadeia de caracteres. Além disso, mesmo que possamos limitar o tamanho máximo, isto faria com que todas as cadeias fossem definidas usando esta quantidade de caracteres, o que poderia gerar um desperdício de memória.*

Faça um programa em Pascal para manipular cadeias de caracteres, que podem ser de tamanhos bem diversos usando listas ligadas. Cada nó da lista apresenta um campo, cadeia, definida como uma string de tamanho MAXCADEIA. Além disso, todos os nós da lista devem usar esta quantidade de caracteres, exceto o último nó que pode ter menos caracteres. O valor de MAXCADEIA é uma constante inteira apropriada, não muito grande. O programa deve ter os seguintes tipos e rotinas:

```
const
  MAXCADEIA       = 10;
  CadeiaVazia =  nil;
type
  tipoString       = String[MAXCADEIA];
  Str = ^regStrString;
  regStrString = record
               cadeia  : tipoString;
               proximo : Str;
             end;

function  Strlength(s:Str):integer; {Retorna a quantidade de caracteres da string s}
procedure Strwrite(s:Str);  {imprime uma String na tela, sem pular linha}
procedure Strwriteln(s:Str);  {imprime uma String na tela, pulando linha}
procedure StrConcatChar(var s:Str; c:char); {Concatena um caracter no fim da String s.}
procedure StrReadln(var s:Str); {Leitura de uma string, colocando caracter por caracter }
function  StrKesimo(s:Str ; K:integer):char;{Retorna o k−esimo caracter de s}

{retorna em destino uma cadeia do tipo Str comecando do caracter de origem da posicao
 inicio até (inicio+tam−1). Se não houver caracteres da String até a posição
 (inicio+tam−1), ou anteriores, estes não são inseridos na cadeia de destino.}
procedure StrSubString(var destino:str; origem:Str ; inicio , tamsub:integer);
{Retorna em destino uma nova String formada pela concatenação de s1 e  s2}
procedure StrConcat(var destino:str ; origem:Str);
{Retorna em destino uma copia da string origem}
procedure StrCopy(var destino:str ; origem:Str);
{Libera a memória (obtida por alocação dinamica) da lista ligada}
procedure Strlibera(var ls:Str );
```

15.4 Exercícios

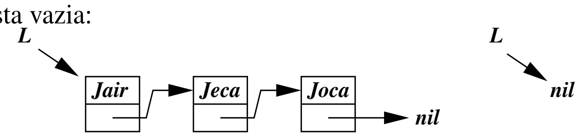
1. Faça um procedimento recursivo para imprimir uma lista de alunos, na ordem da lista.
2. Faça um procedimento recursivo para imprimir uma lista de alunos, na ordem inversa da lista.
3. Faça um procedimento não recursivo para inserir um aluno no fim da lista.
4. Faça um procedimento recursivo para remover um aluno com certo nome da lista.
5. Faça um procedimento recursivo para duplicar uma lista em ordem inversa.
6. Uma lista ligada é declarada da seguinte maneira:

```
type  TipoString = string[255];
      TipoElementoLista = record
                             nome: TipoString;
                             proximo: ^TipoElementoLista;
      end;
      TipoLista = ^TipoElementoLista;
```

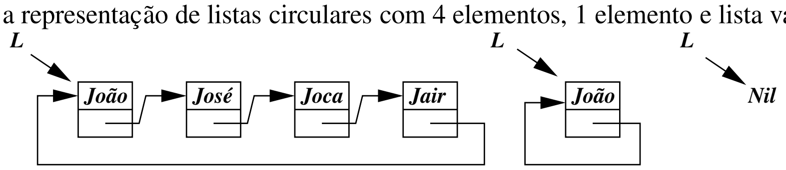
   Faça uma rotina recursiva com o seguinte cabeçalho

   procedure InsereOrdenadoLista(**var** L:TipoLista; Nome:TipoString);

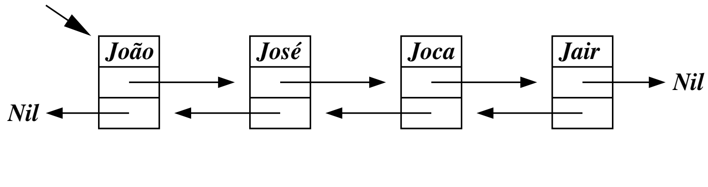
   A rotina insere um novo elemento na lista, com nome dado no parâmetro *Nome*. A função insere de maneira ordenada, i.e., os elementos devem estar ordenados. Além disso, a lista já contém os elementos ordenados antes da inserção. A seguinte figura apresenta exemplo de duas listas ordenadas do tipo *TipoLista*, uma com 3 elementos e uma lista vazia:



7. Faça um procedimento para ordenar uma lista ligada, usando o algoritmo do quicksort tomando sempre o primeiro elemento como pivô.
8. Uma lista é dita circular quando temos uma lista muito parecida com a lista ligada, com exceção que o último elemento da lista (campo próximo do último elemento) aponta para o primeiro elemento da lista. Na figura seguinte apresentamos a representação de listas circulares com 4 elementos, 1 elemento e lista vazia.



   Faça uma rotina que insere um elemento em uma lista circular (não estamos preocupados na posição da lista circular onde o elemento vai ser inserido). Faça uma rotina para remover um elemento da lista, caso este exista.
9. Em um certo jogo temos vários homens que devem ser executados, sendo que eles estão dispostos de maneira a formar um círculo. O carrasco pode escolher um dos homens para ser libertado e usa a seguinte regra: Ele escolhe um primeiro homem a ser executado, executa-o e escolhe o segundo homem após o homem que acabou de ser executado como próximo homem a ser executado. Isto continua até que tenha sobrado apenas um homem, sendo que este homem é libertado. Faça um programa que leia uma lista de nomes de homens e coloque em uma lista circular. Em seguida le o nome do primeiro homem a ser executado. A partir disto, lista na ordem o nome dos homens a serem executados, e por fim o nome do homem libertado.
10. Quando temos uma lista ligada só podemos caminhar por esta lista apenas em um dos sentidos. Uma maneira de contornar este tipo de restrição é usar uma lista duplamente ligada. Neste tipo de lista temos em cada elemento dois campos de ligação. Um chamado *proximo*, que tem a mesma função que o campo proximo em listas ligadas e outro campo chamado *anterior*, que aponta para um elemento anterior na seqüência definida pelo campo proximo. O primeiro elemento da lista tem seu campo anterior com valor **nil**. A figura seguinte apresenta uma lista duplamente ligada com 4 elementos. Faça um programa com rotinas para inserção no início da lista, remoção do primeiro elemento e remoção de um elemento com um determinado nome.



16 Usando e Construindo Biblioteca de Rotinas

Um dos fatores que contribuem para que possamos construir programas de maneira mais produtiva é o uso de biblioteca de rotinas ou unidades. A idéia é ter o código do programa executável gerado por partes a partir de várias unidades.

Apresentamos a seguir algumas vantagens de se usar unidades.

- Reaproveitamento de código. A implementação de uma mesma rotina pode ser usada em diversos programas sem necessidade de recodificação.
- Divulgar rotinas para serem usadas com outros programas em Pascal sem que seja necessário divulgar seu código fonte.
- Uso de bibliotecas livres. Muitos compiladores já oferecem rotinas, na forma de bibliotecas, para manipulação de objetos mais complexos. Exemplo disso são unidades para manipulação de janelas gráficas, disponibilizado por muitos compiladores.
- Divisão funcional do programa. O uso de unidades permite dividir o programa em pedaços menores com o conjunto de rotinas divididas segundo algum critério.
- Agilizar a geração do código do programa executável. Cada unidade pode ser compilada separadamente e seu código objeto pode ser ligado ao de outros para formar o programa executável. Caso haja necessidade da recompilação de uma das unidades, apenas aquela deverá ser compilada.

Até agora descrevemos programas formados por apenas uma unidade, o programa principal. As demais unidades podem ser descritas através de *units* da linguagem Pascal. Uma unit tem uma organização parecida com a de um programa Pascal. As sintaxes do padrão Borland Pascal e Extended Pascal são diferentes, embora parecidas. Optamos por apresentar apenas a sintaxe do padrão Borland Pascal. O leitor interessado não terá dificuldades em transformar programas descritos na sintaxe Borland Pascal para o padrão do Extended Pascal.

16.1 Estrutura de uma unit

Uma unidade, no padrão Borland Pascal, é formada por três partes: uma *Parte Pública*, uma *Parte Privada* e um *Código de Inicialização* (opcional). Na figura seguinte apresentamos a estrutura de uma unidade e nas seções seguintes detalharemos cada uma destas partes.

```
unit <Identificador_Unit>;
interface
      { Parte Pública }

implementation
      { Parte Privada }

[ begin
      { Código de inicialização – Opcional }
]
end.
```

Parte Pública

Na parte pública fazemos as declarações dos objetos que poderão ser visualizados por programas ou outras unidades. Nesta parte podemos declarar constantes, tipos, variáveis e protótipos de rotinas. A declaração de constantes, tipos e variáveis é feita com a mesma sintaxe vista anteriormente. Já os protótipos de rotinas são os cabeçalhos das rotinas que serão visualizadas pelas outras unidades. A seguir exemplificamos os protótipos das rotinas BuscaBin (veja página 104) e SelectionSort (veja página 75), vistas anteriormente.

```
function BuscaBin(var v:TipoVet; inicio,fim:integer; x:real):integer;
procedure SelectionSort(var V:TipoVetorReal; n:integer);
```

Naturalmente todos os tipos não básicos usados nestes protótipos deverão ter sido previamente declarados na unidade ou obtidos através de alguma outra unidade.

A implementação das rotinas não é descrita nesta parte, mas apenas seu cabeçalho para que possamos saber como será feita a interface destas com o programa que fará uso delas.

Dentro da parte privada podemos usar ou aproveitar as declarações de objetos feitas em outras unidades. Para isso, devemos colocar —imediatamente depois da palavra **interface**— o seguinte comando

uses <lista de unidades>;

Com este comando podemos descrever (depois da palavra **uses**) a lista de units que estamos usando para construir a parte pública.

Parte Privada

Nesta parte descrevemos os objetos (constantes, tipos, variáveis, funções e procedimentos) que não poderão ser vistos por outras unidades. Todos os objetos que foram declarados na parte pública podem ser utilizados nesta parte, mas as declarações feitas dentro da parte privada não podem ser utilizados fora da unidades. É nesta parte que iremos descrever a implementação completa das rotinas que tiveram os protótipos declarados na parte pública, bem como de outras rotinas auxiliares. Além disso, o código desta implementação só poderá ser visto pelo usuário se este tiver o código fonte da unidade (um usuário não necessariamente precisa do código fonte de uma unidade para usá-lo).

Aqui também podemos fazer uso de outras unidade. Para isso, devemos colocar —imediatamente depois da palavra **implementation**— o seguinte comando

uses <lista de unidades>;

Código de Inicialização

Esta parte é opcional e serve para executar um código de inicialização da unidade antes da execução do código do programa principal. Para descrever esta parte, devemos colocar a palavra **begin** logo depois da descrição das rotinas da parte privada e antes de **end** (indicador do fim da unit). O código de inicialização da unidade é descrito no bloco de comandos formado por este **begin** e **end**. Caso uma unidade faça uso de várias unidades, suas inicializações serão feitas na ordem descrita no comando **uses**.

16.2 Usando Units

Para declarar o uso de unidades no programa principal, devemos inserir a seguinte cláusula —no início da área de declarações— do programa principal

uses <lista de unidades>;

No quadro seguinte apresentamos um exemplo de como poderíamos declarar uma unidade para manipulação de números complexos.

```
unit complexos;  { Arquivo: complexos.pas }
interface
  type complex = record
              a,b : real; {Número na forma (a+b.i) }
          end;
  procedure LeComplex(var x : complex); { Le um número complexo }
  procedure ImpComplex(x : complex);  { Imprime um número complexo }
  function SomComplex(x,y:complex):complex; { soma de dois números complexos }
  function SubComplex(x,y:complex):complex; { subtração de dois números complexos }
  function MulComplex(x,y:complex):complex; { multiplicação de dois números complexos }
implementation
  procedure LeComplex(var x : complex); { Le um número complexo }
  begin
    writeln('Entre com um número complex (a+b.i) entrando com a e b: ');
    readln(x.a,x.b);
  end; { LeComplex }
  procedure ImpComplex(x : complex);  { Imprime um número complexo }
  begin
    writeln('(',x.a:5:2,'+',x.b:5:2,'i)');
  end; { ImpComplex }
  function SomComplex(x,y:complex):complex; { soma de dois números complexos }
  var z : complex;
  begin
    z.a := x.a+y.a;    z.b:=x.b+y.b;
    SomComplex := z;
  end; { SomComplex }
  function SubComplex(x,y:complex):complex; { subtração de dois números complexos }
  var z : complex;
  begin
    z.a := x.a−y.a;    z.b:=x.b−y.b;
    SubComplex := z;
  end; { SubComplex }
  function MulComplex(x,y:complex):complex; { multiplicação de dois números complexos }
  var z : complex;
  begin
    z.a := x.a*y.a−x.b*y.b;    z.b:=x.a*y.b+x.b*y.a;
    MulComplex := z;
  end; { MulComplex }
end.
```

No programa seguinte apresentamos um exemplo de programa que usa esta unit.

```
program ProgComplex; { Arquivo: principal.pas }
uses complexos;
var x,y       : complex;
begin
  LeComplex(x); LeComplex(y);
  write('A soma dos números complexos lidos é '); ImpComplex(SomComplex(x,y));
  write('A subtração dos números complexos lidos é '); ImpComplex(SubComplex(x,y));
  write('A multiplicação dos números complexos lidos é '); ImpComplex(MulComplex(x,y));
end.
```

Usando GPC para compilar e ligar unidades

Antes de apresentar as alternativas para gerar o programa executável usando unidades no GPC, vamos precisar de duas definições: Arquivos gpi e arquivos com código objeto.

(i) Arquivos GPI (GNU Pascal Interface): São arquivos que contém as informações pré-compiladas da interface da unidade. Não contém o código de máquina das rotinas da unidade. O nome dos arquivos deste tipo tem extensão ".gpi".

(ii) Arquivos com código objeto: São estes arquivos que contém a codificação em linguagem de máquina das unidades ou do programa principal principal. Em geral são arquivos com extensão ".o".

A seguir descrevemos três maneiras para se gerar o programa executável usando unidades no GPC.

1. *Usando a opção automake:* Nesta opção, o compilador compila apenas as unidades que foram modificadas ou aquelas que direta ou indiretamente fazem referência à unidade modificada. Sintaxe:
   gpc --automake --borland-pascal <lista de unidades> <programa_principal.pas> -o <arquivo executável>
   Após a execução deste comando, são gerados os arquivos GPI e os arquivos objetos de cada unidade.

   Caso o nome do arquivo de cada unidade, sem a extensão ".pas", for igual ao identificador da unidade, então não é necessário especificar a lista de unidades no comando acima. Todas as unidades que forem referenciadas direta ou indiretamente serão consideradas na compilação. Neste caso, o comando pode ser resumido para:
   gpc --automake --borland-pascal <programa_principal.pas> -o <arquivo executável>
2. *Especificando todos os arquivos fontes:* Nesta opção, o compilador compila todos os arquivos fontes para gerar um arquivo executável. A seguir apresentamos uma sintaxe para o comando de compilação:
   gpc --borland-pascal <lista de unidades> <programa_principal.pas> -o <arquivo executável>
   No comando acima, a lista de unidades é separada por espaços em branco e deve seguir uma ordem. Se uma unidade B faz uso de uma unidade A, então A deve vir antes de B. O compilador GPC gera um arquivo GPI para cada unidade mas não gera arquivos objetos.

   Esta maneira de compilar os programas é mais lenta, uma vez que todos os arquivos fonte são compilados.
3. *Gerando o código objeto:* Neste caso, a compilação de uma unidade pelo GPC deve ser feita com a opção "-c" (sem aspas). Por exemplo, a unidade complexos.pas poderia ter sido compilada com o seguinte comando:
   gpc --borland-pascal -c complexos.pas
   Após este comando, obtemos os arquivos GPI e objeto da unidade *complexos.pas*. Para gerar o código executável, basta entrar com o seguinte comando:
   gpc complexos.o principal.pas -o executavel
   O compilador distingue os arquivos no formato pascal (.pas) e os arquivos já compilados (.o). Para os arquivos no formato pascal, o compilador gera seu código objeto (correspondente .o) e liga todos estes códigos objetos em apenas um programa executável.

   Uma vez que temos o arquivo gpi e o código objeto da unidade, não mais necessitamos do seu código fonte. Naturalmente, se houver necessidade de mudar o código fonte ou de gerar código para um outro tipo de computador, os arquivos gpi e o código objeto deverão ser gerados novamente a partir do arquivo fonte da unidade.

16.3 Exercícios

1. Faça uma unit em um arquivo chamado *numerico.pas* com as seguintes operações para manipulação de números e rotinas numéricas: *(i)* Cálculo de uma raiz de função usando o método da bisseção, *(ii)* cálculo da integral definida *(iii)* verificação da primalidade de um número, *(iv)* funções numéricas como: cálculo da raiz quadrada, seno, cosseno, fatorial, potência, exponencial etc.
2. Faça uma unit em um arquivo chamado *cadeias.pas* com as seguintes operações para manipulação de cadeia de caracteres: *(i)* Inserção de cadeias em outras (insert), *(ii)* busca de uma cadeia em outra, *(iii)* demais operações como: mudança para maiúsculas ou minúsculas, remoção de acentos etc.
3. Faça uma unit em um arquivo chamado *ordena.pas* com as seguintes rotinas para ordenar vetores de números reais: SelectionSort, MergeSort e QuickSort. Além disso insira rotinas auxiliares como: para fazer busca binária, intercalar dois vetores ordenados em um terceiro vetor etc.

4. Faça uma unit para manipulação de inteiros longos. Cada inteiro longo será armazenado como um vetor de dígitos. Além disso cada inteiro longo terá um atributo que deve ser o número de dígitos usado pelo número representado. Faça rotinas para manipular estes números, como por exemplo: *(i)* Leitura de inteiro longo, *(ii)* Soma, *(iii)* Subtração, *(iv)* Multiplicação, *(v)* Divisão inteira, *(vi)* Resto de divisão inteira de dois inteiros longos e *(vii)* Impressão de inteiros longos.

Índice Remissivo