-- LÓGICA DE PROGRAMAÇÃO --

Olá!

Chegamos na disciplina de Lógica de Programação, onde abordar os conceitos da lógica computacional, o raciocínio lógico como um todo e como projetar soluções em forma de código computacional para resolução de problemas simples. A disciplina está dividida em 7 capítulos, como dispostos nesta apostila. Além de toda fundamentação teórica e exercício propostos, a apostila contém seções especiais, explicadas abaixo:

CÓDIGO

Nas seções de código, teremos sempre um exemplo em Portugol e o mesmo exemplo em Pascal, de modo a exercitarmos e observarmos a lógica em Português e em Inglês.

EXERCÍCIOS

No final da apostila, há uma série de exercícios sem resolução para treinamento em lógica de programação.

1. INTRODUÇÃO AS LINGUAGENS E À LÓGICA

Você já deve ter tido, por diversas vezes, que usar seu raciocínio lógico. Seja para resolver problemas de matemática num concurso ou para o simples cotidiano como preencher uma palavra cruzada. E foi assim, treinando, que certamente você aprendeu a raciocinar e a pensar. E se você tivesse que ensinar um robô a pensar da mesma forma que você? Imagina que um robô deva aprender a somar, calcular, responder perguntas. Temos então o grande desafio de trazer o raciocínio humano para o mundo da lógica computacional. E pra isso dispomos de uma importante ferramenta: as linguagens de programação.

Mas o que são as linguagens? São simplesmente códigos usados para escrever qualquer tipo de instrução para uma máquina, seja um robô, um carro, um celular, um servidor web ou um computador doméstico. Esses códigos são organizados (e executados) numa sequência lógica, linha-a-linha, de modo que ao executar cada última linha, o "problema" proposto fique mais próximo da resolução. Costumamos medir as linguagens em níveis, de acordo com a proximidade e familiaridade do código com o programador. Quanto mais próximo do programador, mais alto seu nível. Vamos conhecer os níveis de linguagem.

1.1. Linguagens de Baixo Nível

Também chamadas de linguagens de máquina, as linguagem de máquina são poucos inteligível para os humanos. Resume-se em dígitos binários (0 e 1) organizados para executar tarefas de *hardware*, como alocação de memória, gerenciamento do disco e acionamento do processador. Contudo, os processadores e outros *hardwares* continuam sendo desenvolvidos. Como então são programados?

Pela sua dificuldade de compreensão, a linguagem de máquina ganhou uma linguagem "intermediária", o *Assembly*. Apesar de extremamente limitada, é mais (minimamente, por sinal) amigável ao programador e permite desenvolver programas que requerem um contato mais "íntimo" com o *hardware*, como *drivers* e *firmwares*. Para executar as instruções lógicas, o Assembly utiliza recursos do próprio hardware como os registradores do processador. Após o código ser escrito, é necessário tratá-lo em um *software* específico, o Assembler, que irá "traduzir" e montar o código em Assembly em código binário puro. Vencida esta etapa, o código está pronto para uso.

1.2. Linguagens de Alto Nível

Em contraste com as linguagens de máquina, as linguagens de alto nível são extremamente simples de compreender do ponto de vista humano, por sua proximidade como o idioma inglês e de notações matemáticas para realizar as instruções lógicas. Como as instruções são mais "humanas", o trabalho pesado cabe a um "tradutor" converter em linguagem de máquina (já que é a única coisa que o computador entende) para o código ser compreendido pelos *hardwares*.

Uma grande e crucial diferença entre as linguagens de baixo e alto nível (além da compreensão de ambas) é a interoperabilidade (hein?). Quando escrevemos Assembly, o código é direcionado para um hardware específico, com arquitetura específica (64 bits, por exemplo). Já no código de alto nível, a execução é "interoperável", isto é, pode operar em qualquer plataforma, arquitetura ou sistema operacional com a certeza de funcionamento. Essa propriedade também é referenciada como Estrutura Dinâmica de Dados (EDD), na qual o funcionamento interno código se adequa ao ambiente computacional em que está inserido.

Alfabeto Binário

Você já deve saber que nosso processador entende apenas sequências binárias. E quando você digita seu nome num editor de texto? Veja o alfabeto em binário:

Minúsculo		**Maiúsculo**	
a	01100001	A	01000001
b	01100010	B	01000010
c	01100011	C	01000011
d	01100100	D	01000100
e	01100101	E	01000101
f	01100110	F	01000110
g	01100111	G	01000111
h	01101000	H	01001000
i	01101001	I	01001001
j	01101010	J	01001010
k	01101011	K	01001011
l	01101100	L	01001100
m	01101101	M	01001101
n	01101110	N	01001110
o	01101111	O	01001111
p	01110000	P	01010000
q	01110001	Q	01010001
r	01110010	R	01010010
s	01110011	S	01010011
t	01110100	T	01010100
u	01110101	U	01010101
v	01110110	V	01010110
x	01111000	X	01011000
w	01110111	W	01010111
y	01111001	Y	01011001
z	01111010	Z	01011010

1.2.1. Programação não estruturada

Dentre as linguagens de alto-nível, temos alguns paradigmas (modelos) de programação. Um deles são os tipos não estruturados, também chamados de programação linear, que se caracterizam não aceitar blocos de códigos estruturados, organizando o código de forma sequencial (linha a linha). Quando há a necessidade de executar outra linha que não esteja na sequência, o código utiliza **desvios condicionais**, indicando um novo ponto de partida para a execução (comumente através de comandos do tipo GOTO ["vá para"]). É como se todo o código não tivesse um início e um fim bem definidos.

No quadro de exemplo abaixo, podemos ver um código na linguagem FORTRAN, em que são utilizados desvios para determinar qual linha deve ser executada.

CÓDIGO	RESULTADO

`goto 20`
`10 write (*,*) 'João'`
`20 write (*,*) ' Maria'`
`30 write (*,*) ' José'`
`end`
	`Maria`
	`José`

Logo no início tem um GOTO para a linha 20, onde o código irá escrever ('write') na tela a palavra "Maria", escrevendo em seguida a palavra "José".

1.2.2. Programação estruturada

Outro paradigma é a programação estruturada, também chamada de programação modular, onde o código é organizado em blocos estruturados e sequenciais. Sua principal diferença é continuidade da estrutura do programa, sem desvios que possam retardar a execução e com início e fim bem definidos. A estruturação objetiva tornar o código mais compreensível e mais fácil de ser depurado, em caso de erros na execução. Observe abaixo no quadro, um exemplo na linguagem PASCAL, baseado no exemplo anterior em FORTRAN.

CÓDIGO	RESULTADO

`begin`
`1 writeln ('João');`
`2 writeln ('Maria');`
`3 writeln ('José');`
`end`
	`João`
	`Maria`
	`José`

Como não há desvios nesse exemplo, o código é executado de forma sequencial, linha a linha, escrevendo nomes na tela, na ordem natural em que o código foi escrito.

1.2.3. Programação orientada a objetos

Paradigma de programação usado em grande escala atualmente, representa uma grande mudança no modelo tradicional de programação de sistemas. A base é a **abstração** (simplificação) dos comportamentos ou ações dos objetos do mundo real que serão representados no mundo computacional. Além dos comportamentos, são também representadas as características desse objeto que, de maneira geral, são alteradas conforme seus comportamentos.

Para entender vamos pensar em um objeto qualquer: um elevador, por exemplo. Ele tem características reais (peso máximo, tamanho, andar atual) que são modificadas com alguns comportamentos (subir, descer, entrar ou sair pessoas). Portanto, ao criarmos um programa sobre elevadores, todas essas características e comportamentos deverão ser possíveis de se realizar (computacionalmente falando) via código. Acompanhe abaixo, exemplos de códigos na linguagem JAVA, na construção de um objeto elevador, suas características e seus comportamentos.

MODELANDO O ELEVADOR COM SUAS CARACATERÍSTICAS

```
public class Elevador
{
   private int pesoMaximo;
   private int andarAtual;
   private int tamanho;
   private int numeroDePessoas;
}
```

No código acima, é criado um modelo (chamado de 'Classe') de elevadores. Nas linhas abaixo, são escritas as características que os objetos criados terão, no caso, um peso máximo, um tamanho, a quantidade de pessoas atualmente no elevador e seu andar atual.

CRIANDO OS COMPORTAMENTOS DOS OBJETOS

```
public void movimentar(int x){
   andarAtual = x;
}
public void entrarPessoas(int y){
   numeroDePessoas += y;
}
```

No comportamento "movimentar", deve-se indicar um andar para deslocamento (x). Após isso, o andar atual será o indicado. Do mesmo modo, o número de pessoas será acrescido no número adicional indicado (y).

1.3. Histórico de linguagens

A programação começou 'oficialmente' na década de 1960, evoluindo-se desde então, passando por vários paradigmas e objetivos. Vamos ver um breve histórico das linguagens de programação.

1.3.1. ALGOL

Criada em 1960, foi a primeira linguagem de programação com sintaxe (forma de programar) definida. Acabou influenciando várias linguagens posteriores, embora o próprio ALGOL seja pouquíssimo utilizado atualmente.

1.3.2. COBOL

Também de 1960, foi a primeira linguagem de programação recomendada pelo DoD, o departamento de defesa estadunidense. Ainda é usada, embora de modo bastante discreto, principalmente em aplicações comerciais antigas. Considerada de boa legibilidade, mas com má redigibilidade.

1.3.3. BASIC

Foi criada em 1964, na Universidade de Darmouth. É considerada de fácil aprendizado para uso de estudantes e ciência humana. Não mais usada atualmente.

1.3.4. PASCAL

Criada em 1971, é uma linguagem extremamente simples. Por este motivo, é usada até hoje para aprendizado de programação estruturada e algoritmos.

1.3.5. C

Considerada uma grande "mãe" para várias linguagens, foi criada em 1972, projetada para programação de sistemas. O Unix, tradicional sistema operacional que deu origem ao núcleo do Linux, foi inteiramente criado em linguagem C.

1.3.6. PROLOG

Também criada em 1972, é uma linguagem que trabalha exclusivamente com lógica, sendo, por este motivo, bastante usada em inteligência artificial.

1.3.7. SMALLTALK

Outra linguagem de 1972, representou uma grande revolução, sendo a primeira linguagem orientada a objetos e a primeira a utilizar ambiente gráfico.

1.3.8. ADA

Criada em 1983, levou 8 anos para ser desenvolvida, sendo grande e complexa. É utilizada principalmente em programação concorrente e sistemas em tempo real.

1.3.9. C++

Como o nome indica, é baseada em linguagem C e foi criada em 1985. Orientada a objetos, foi rapidamente aceita, mas tornou-se complexa.

1.3.10. JAVA

Umas grandes plataformas atuais de programação, a linguagem Java foi criada em 1995, baseada em C++, porém bem mais simples. É de fácil aprendizado, com foco na portabilidade, na qual o código pode ser executado em qualquer plataforma ou sistema operacional.

1.4. Aprendendo Lógica

Quando aprendemos lógica computacional, logo debatemos com a lógica idealizada por Aristóteles, aquela na qual temos apenas 2 estados, sem ambiguidades. Ou é 1 ou é 0. Ou está ligado ou está desligado. Ou é sim ou é não. E a lógica de programação nada mais é do que escrever instruções para um computador executar uma tarefa ou outra, sem duplo sentido. É importante frisar que durante o aprendizado de lógica, pouco a importa a linguagem de programação que iremos lidar.

1.4.1. Argumentos

São proposições que englobam premissas, seguidas de uma conclusão. Para argumentos válidos, a conclusão deve ser sempre justificada pelas premissas.

EXEMPLO

```
1 Todo aluno de Computação precisa estudar lógica.
2 José é aluno de Computação.
3 Logo, José precisa estudar lógica.
```

Argumentos também podem levar a conclusões erradas, por não se observar as premissas ou dados do problema.

EXEMPLO

```
1 Se eu ganhar na loteria serei rico.
2 Não ganhei na loteria.
3 Logo, sou pobre.
```

A Lógica Computacional procura validar uma conclusão, verificando se está de acordo com as premissas. Ao analisar premissas e propor uma conclusão, estamos realizando um processo chamado **inferência**.

As conclusões dos argumentos podem ser formuladas por 2 métodos, sempre observando-se as premissas. A **dedução** especifica uma veracidade definitiva da conclusão, assumindo uma impossibilidade da conclusão não ocorrer.

DEDUÇÃO

1 Todos os diamantes são duros.
2 Alguns diamantes são joias.
3 Algumas joias são duras.

INDUÇÃO

1 A vacina funcionou nos ratos.
2 A vacina funcionou nos macacos.
3 Logo irá funcionar nos humanos.

Já a **indução** representa uma possibilidade/probabilidade da veracidade da conclusão (a conclusão é formada por observação ou experiência).

TENTE VOCÊ MESMO

"Lucimar que realiza as compras da Empresa XYZ, tinha uma reunião marcada às 10 horas no escritório do Sr. Smith, para discutir os termos de um grande pedido. No caminho para esse escritório, escorregou no chão recém-encerado, e como resultado machucou gravemente a perna. Quando o Sr. Smith foi informado do acidente, Lucimar estava a caminho do hospital para tirar radiografias. O Sr. Smith telefonou para o hospital para saber notícias, mas a atendente parecia saber nada a respeito de Lucimar. É possível que o Sr. Smith tenha telefonado para o hospital errado?"

Com base nas premissas acima, analise as conclusões e coloque (D)EDUÇÃO para verdades definitivas ou (I)NDUÇÃO para verdades prováveis.

- **O senhor Lucimar é um comprador ( )**
- **Era esperado que Lucimar se reunisse com Sr. Smith ( )**
- **O acidente ocorreu na Empresa XYZ ( )**
- **Levaram Lucimar ao hospital para fazer radiografia ( )**
- **Lucimar tinha uma reunião marcada para as 10 horas ( )**
- **Ninguém no hospital sabia qualquer coisa sobre de Lucimar ( )**

1.4.2. Tabela Verdade

Utiliza operadores lógicos para analisar premissas e propor conclusões. Esses operadores podem ser de conjunção (E/AND) ou disjunção (OU/OR).

Imagine que para ser programador, você precisa aprender Java **E** C++. Então teríamos a seguinte tabela:

VOCÊ CONHECE JAVA	VOCÊ CONHECE C++	RESULTADO
NÃO	NÃO	VOCÊ NÃO É PROGRAMADOR
NÃO	SIM	VOCÊ NÃO É PROGRAMADOR
SIM	NÃO	VOCÊ NÃO É PROGRAMADOR
SIM	SIM	VOCÊ É PROGRAMADOR

Perceba que com o operador E, apenas se **todas** as premissas forem verdadeiras, o resultado também será verdadeiro.

Agora imagine que para ser programador, você precisa aprender Java **OU** C++. Então teríamos a seguinte tabela:

VOCÊ CONHECE JAVA	VOCÊ CONHECE C++	RESULTADO
NÃO	NÃO	VOCÊ NÃO É PROGRAMADOR
NÃO	SIM	VOCÊ É PROGRAMADOR
SIM	NÃO	VOCÊ É PROGRAMADOR
SIM	SIM	VOCÊ É PROGRAMADOR

Perceba que no operador OU, basta qualquer premissa ser verdadeira para o resultado também ser verdadeiro.

2. INTRODUÇÃO AOS ALGORITMOS

Um algoritmo nada mais é do que uma sequencia de passos organizados, de modo a produzir um resultado que é a solução do problema proposto. O algoritmo não pode ser ambíguo, isto é, não pode ter dois caminhos possíveis e seu tempo de execução deve ser, claro, finito. Historicamente, algoritmos se mostram como o cerne da computação, desde o tempo das válvulas computacionais e ao longo da história, sua otimização e nível de eficiência tem sido os principais desafios no desenvolvimento.

Podemos pensar em algoritmos como qualquer outro processo computacional, onde exista entrada de dados, processamento dos dados e saída de resultado.

ENTRADA	PROCESSAMENTO	SAÍDA
Dados iniciais que serão inseridos no programa	Cálculos, manipulação de arquivos, exclusões, etc.	Pode ser uma mensagem, um resultado numérico ou uma ação específica.

Fazendo uma analogia com o mundo real, imagine uma máquina de carne. Ao inserir o produto no equipamento, temos a "entrada de dados". Logo após, giramos a manivela para o "processamento" dos dados iniciais. Ao final, temos a saída esperada que é a carne em forma de linguiça, por exemplo. Mas como representar os algoritmos no mundo computacional? Tradicionalmente, podemos representá-los de 3 formas distintas.

2.1. Narração ou Descrição

A forma narrativa utiliza palavras de nosso idioma nativo para explicar os passos do algoritmo, em forma própria de narração. Em uma analogia, seria como a receita de um bolo, com as etapas descritas de forma natural.

NARRAÇÃO OU DESCRIÇÃO - TOMANDO UM BANHO

```
1 Despir-se.
2 Abrir o chuveiro.
3 Enxaguar-se.
4 Ensaboar-se.
5 Enxaguar-se novamente.
6 Fechar o chuveiro.
7 Secar-se.
8 Vestir-se.
```

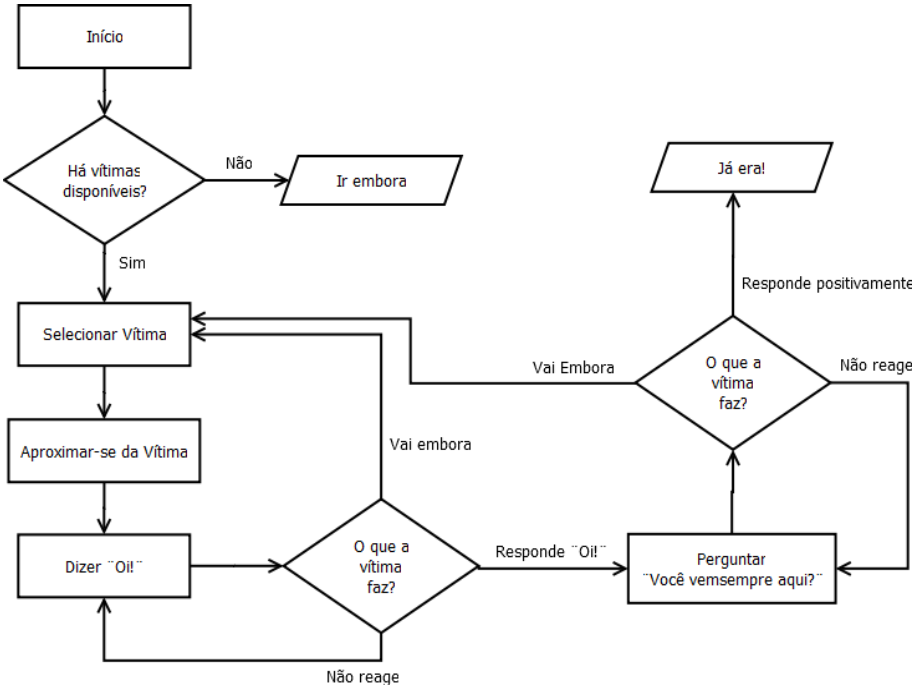
Quer tentar? Imagine um rio onde você tenha que atravessar com um leão, uma cabra e uma cesta de capim, um de cada vez. Narre a travessia, evitando a "morte" de qualquer um deles.

2.2. Fluxogramas (*Flowchart*)

Apresentam um padrão mundial na representação de algoritmos por meio de figuras. Embora eficiente, é indicado para algoritmos de pequeno porte, para fácil entendimento, não dando atenção aos dados e não oferecendo recursos para representá-los ou descrevê-los.

Processo	Processo Alternativo	Decisão	Dados
Processo Pré-definido	Armazenamento Interno	Documento	Vários Documentos
Terminação	Preparação	Entrada Manual	Operação Manual
Conector	Conector Fora de Página	Cartão	Fita Perfurada
Somador	Ou	Agrupar	Classificar
Extrair	Mesclar	Dados Armazenados	Atraso
Armazenamento de Acesso Sequencial	Disco Magnético	Armazenamento de Acesso Direto	Exibir

Vamos a uma pequena brincadeira. Imagine que você vá uma festa e quer "paquerar" alguém, mas não sabe como fazer isso. Um(a) amigo(a) resolve ajudar e lhe orienta através de um fluxograma. Como seria esse fluxograma? A resposta pode estar ao lado. Confira!



2.3. Pseudocódigo

Trata-se do código, propriamente dito, escrito de maneira lógica e fiel a sequência do fluxograma e da narração, organizadas de maneira estruturada. É amplamente disseminada no idioma inglês, principalmente em linguagem PASCAL, porém há uma versão em língua portuguesa chamada "portugol" (português+algoritmo) em que os comandos são traduzidos para o idioma português brasileiro. Vejamos um exemplo comparativo sobre um algoritmo para somar 2 números quaisquer.

PASCAL	PORTUGOL

`Program Soma`
`var`
`a, b, c:integer`
 	
`Begin`
`  read(a)`
`  read(b)`
`  c:=(a+b)`
`  write (c)`
`End`
	`Algoritmo Soma`
	`var`
	`a, b, c:inteiro`
	
	`inicio`
	`  leia(a)`
	`  leia(b)`
	`  c<-a+b`
	`  escreva(c)`
	`fimalgoritmo`

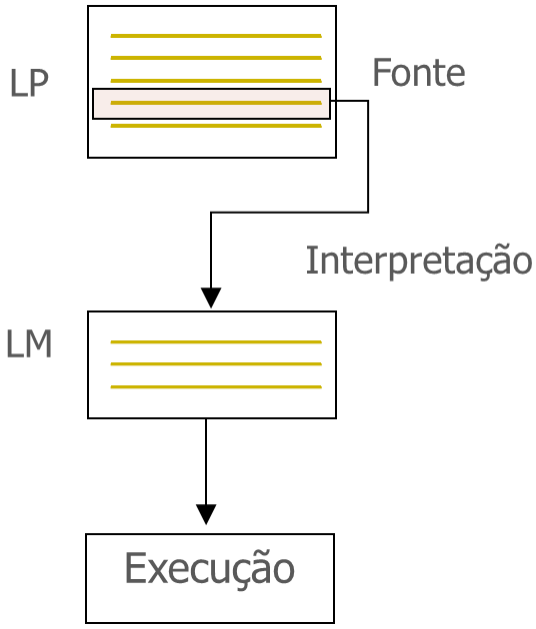
Os dois códigos acima, fazem exatamente a mesma coisa. Não nos preocupemos ainda com o significado dos comandos, concentremo-nos nas sintaxes e estruturas do algoritmo. O código começa declarando o nome do algoritmo ('Soma') e logo após, 3 elementos (a, b e c) contendo um número inteiro. Temos então o início do algoritmo com a leitura (*read*) dos elementos a e b, isto é, a atribuição de um número inteiro a cada um deles. Esta atribuição pode ser feita através do teclado (digitando algum número) ou de forma sistêmica (o próprio programa define um número) e irão caracterizar as entradas do programa, ou seja, os dados iniciais. Logo após, temos um processamento, um cálculo de soma, entre os elementos a e b. O resultado desta soma é atribuído ao elemento c. Por fim, o elemento c é exibido na tela do sistema.

Teste de Mesa

É sempre interessante, para fins de testes, simular a execução do algoritmo de forma escrita, comparando o resultado obtido com o resultado esperado. À essa simulação damos o nome de Teste de Mesa. Mas como fazer?

Pegue o exemplo do algoritmo de Soma. Na linha "leia (a)", atribua um número ao elemento a (por exemplo, 5). Faça o mesmo com o elemento b (por exemplo, 8). Na linha "c<-a+b", o elemento c receberá o resultado da soma (13). Confira se aparece o 13 na tela. Se sim, seu teste de mesa funcionou.

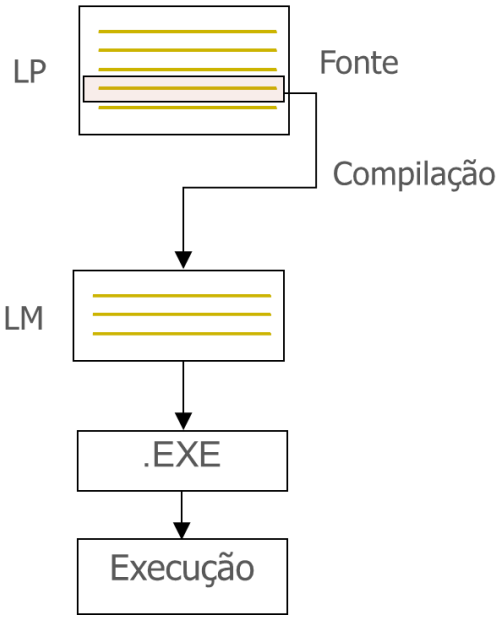
2.4. Interpretadores



Temos então uma das grandes diferenças em linguagens de programação. Algoritmos interpretados são os mais velozes. A razão é que ao ser lido, o código é executado "em tempo real", isto é, após ler uma linha, o interpretador já transforma em linguagem de máquina e já envia para execução, passando então para a próxima linha, reiniciando então o processo. Apesar de ser veloz, há uma "desvantagem" em linguagens de programação que são interpretadas. Quando uma linha ou um bloco de código é executado, a próxima instrução não se inicia, enquanto esta anterior não for totalmente executada.

Isto acontece comumente com páginas *web*, quando acessamos um determinado site e a página não 'carrega' rapidamente e a tela fica branca ou quando um conteúdo demora a aparecer na tela. A causa pode ser um bloco de código que precisa ser interpretado primeiro e demora mais do que o normal, então o interpretador não consegue executar as linhas de baixo (que contém o conteúdo que está demorando a aparecer), pois está "preso" ao script anterior.

2.5. Compiladores



Diferentemente de linguagens interpretadas, a linguagem compilada não é executada "em tempo real", linha a linha, sendo, por este motivo, ligeiramente mais lenta. O compilador lê todas as linhas do programa e gera um arquivo que não inteligível para os humanos. Somente após esse processo, o arquivo é enviado para execução. A "desvantagem" da compilação está na atualização do código, isto é, qualquer mudança nas linhas, será necessário compilar o algoritmo novamente, gerando um novo arquivo executável.

O arquivo gerado pode ser em diversas extensões, embora seja tratado como um .exe (de '*executable*'). É este arquivo que deve ser distribuído quando um programa em Java for finalizado. (ou você pensou que iria distribuir os arquivos com seu código pra todo mundo?)

3. VARIÁVEIS

Durante a execução de um algoritmo, podem ser armazenadas várias informações na memória, cada qual em um espaço diferente e isso requer uma forma de organização eficiente. O armazenamento de informações é necessário para que se possam reaproveitar os dados inseridos e não perdê-los ao longo do código, até que o algoritmo termine de executar. Por exemplo, um nome que seja digitado logo nas primeiras linhas de um algoritmo pode ser útil mais abaixo, sendo necessário então armazenar a informação para conseguir recuperá-la mais tarde. Temos então as variáveis.

As variáveis nada mais são do que espaços na memória para guardarmos informações. Esses espaços podem armazenar diferentes informações ao longo do algoritmo ou até em execuções diferentes (cada vez que executamos o código, iremos digitar um nome diferente), variando, portanto de valor, dai o nome de variáveis. O contrário de uma variável é chamado de constante, que são aquelas informações cujo valor não se altera durante a execução do algoritmo (por exemplo, o número π [pi]).

3.1. Tipos de variáveis

Cada informação armazenada em uma variável contém um tipo. Este tipo irá definir, entre outras coisas, o tamanho do espaço que será preparado na memória para guardar tal informação. Temos 4 tipos principais de dados.

3.1.1. Inteiro (*Integer* ou *Int*)

Variáveis do tipo Inteiro podem armazenar informações numéricas inteiras positivas ou negativas (valor zero também é aceito). Casas decimais ou valores fracionários não são aceitos. Alguns exemplos dessas informações são valores que representam idades, filhos, anos, números de conta corrente, etc.

3.1.2. Real (*Real*)

Variáveis do tipo Real podem armazenar quaisquer números existentes, tanto inteiros quanto fracionários, tanto positivos quanto negativos, incluindo o zero. Algumas linguagens aceitam a vírgula (,) como separador de casa decimal, outras aceitam o ponto (.). Alguns exemplos dessas informações são valores que representam salários, alturas humanas, juros bancários, etc. Em algumas linguagens, o tipo Real pode ter variação na precisão das casas decimais, como por exemplo, na linguagem JAVA, onde o tipo *float* trabalha com precisão de até 7 casas decimais e o tipo *double* com precisão de 14 dígitos decimais.

3.1.3. Literal (*String*)

É o tipo destinado para textos. Qualquer informação que não seja efetivamente numérica ou "calculável" é considerada literal. As informações literais vêm representadas sempre entre aspas. Isto vale também quando um número está aspas, deixando de ser considerado um número propriamente, passando a valer como texto. Informações literais podem ser, por exemplo, nomes, endereços, cargos, etc.

3.1.4. Lógico (*Boolean*)

Variáveis do tipo Lógico remetem à lógica clássica de Aristóteles. Somente podem ser armazenados valores não ambíguos, do tipo Sim/Não, Ligado/Desligado, Verdadeiro/Falso, Ativado/Desativado, em que somente uma única resposta pode ser armazenada entre as duas únicas opções existentes. Exemplos são dependentes (ou você têm ou você não têm), fumante (ou você é ou você não é), etc.

3.2. Nomes de variáveis

Todas as variáveis, sem exceção, devem possuir um nome, de modo que possa ser possível recuperar um valor armazenado através do nome de sua variável. É que como se tivéssemos um armário gigante (a memória) com milhares de compartimentos (variáveis) para guardar coisas. Seria então conveniente colocar uma etiqueta em cada compartimento para lembrar o que guardamos ali. (ou você prefere memorizar tudo?).

Os nomes de variáveis seguem regras específicas. Por exemplo, não se pode nomear uma variável com palavras reservadas da linguagem, ou seja, palavras que são usadas para comando. Veja abaixo uma lista de palavras reservadas do Portugol.

PALAVRAS RESERVADAS			
aleatorio	e	grauprad	passo
abs	eco	inicio	pausa
algoritmo	enquanto	int	pi
arccos	entao	interrompa	pos
arcsen	escolha	leia	procedimento
arctan	escreva	literal	quad
arquivo	exp	log	radpgrau
asc	faca	logico	raizq
ate	falso	logn	rand
caracter	fimalgoritmo	maiusc	randi
caso	fimenquanto	mensagem	repita
compr	fimescolha	minusc	se
copia	fimfuncao	nao	sen
cos	fimpara	numerico	senao
cotan	fimprocedimento	numpcarac	timer
cronometro	fimrepita	ou	tan
debug	fimse	outrocaso	verdadeiro
declare	função	para	xou

Palavras Reservadas do Pascal				
and	downto	in	packed	to
Array	else	inline	procedure	type
asm	end	interface	program	unit
begin	file	label	record	until
case	for	mod	repeat	uses
const	foward	nil	set	var
constructor	function	not	shl	while
destructor	goto	object	shr	with
div	if	of	string	xor
do	Implementation	Or	Then	

Além das palavras reservadas, há outras regras gerais que orientam como nomear variáveis corretamente e evitar que seu programa gere um erro por causa de nomes inválidos. etc) e não

- Nomes de variáveis não podem conter espaço (use *underline* para separar palavras);
- Nomes de variáveis não podem conter acentos ou caracteres especiais (vírgulas, etc);
- Nomes de variáveis não devem começar com números (mas podem conter números);
- Nomes de variáveis, costumeiramente, devem ser escritos com letras minúsculas;

3.3. Atribuição de valores

As variáveis podem conter valores atribuídos pelo próprio sistema, inclusive valores vazios. O símbolo de atribuição varia de linguagem para linguagem. No PASCAL, o símbolo é representado por dois pontos seguido de igual (:=). Já no Portugol, o símbolo é o sinal de menor e um hífen (<-), representando uma seta. Veja abaixo, alguns exemplo de atribuição de valores nas variáveis com seus respectivos nomes.

- **nome<-"João" ou nome:="João"**
- **altura<-1.70 ou altura:=1,70**
- **cônjuge<-" " ou cônjuge:=" "**
- **fumante<-Verdadeiro ou fumante:=true**

3.4. Expressões matemáticas

Expressões matemáticas em algoritmos são bastante comuns. Mas por questões tecnológicas não é possível escrever as expressões da mesma forma que vemos nos livros. Por esta razão as expressões devem ser sempre linearizadas para possibilitar seu processamento em algoritmos.

$$\left\{\left[\frac{2}{3}-(5-3)\right]+1\right\}.5 \qquad ((2/3-(5-3))+1)*5$$

Observe que as frações foram linearizadas e as chaves e colchetes foram representados inteiramente por parênteses, organizados de acordo com a lógica matemática (primeiro resolve-se os parênteses internos e depois os externos).

Os operadores matemáticos também são representados de forma particular em algoritmos. Os símbolos como sempre, podem variar de linguagem para linguagem, sendo comum à maioria delas. Outras linguagens apresentam comandos próprios para certas operações matemáticas, como por exemplo, o Portugol com o comando **raizq** para cálculos de raiz quadrada (**sqrt** em PASCAL) ou o comando radpgrau para conversão de radianos em graus. Acompanhe a tabela abaixo com os operadores matemáticos mais comuns na maioria das linguagens.

OPERADOR	SÍMBOLO	EXEMPLO
Adição	+	a+b
Subtração	-	a-b
Multiplicação	*	a*b
Divisão	/	a/b
Divisão inteira	\	a\b
Resto	%	a%b
Exponenciação	^(base^expoente)	a^b

Agora que aprendemos alguns conceitos básicos de algoritmos, vamos tentar construir nossos primeiros algoritmos com alguns exercícios?

Exercícios

- **Construa um algoritmo que calcule a área do círculo;**
- **Faça um algoritmo para calcular o Índice de Massa Corporal de uma pessoa. O índice é obtido dividindo o peso pela altura ao quadrado;**
- **Faça um algoritmo para calcular a média aritmética do seu curso, sendo 2 testes, 1 trabalho e uma prova com peso 2.**

4. ESTRUTURAS DE SELEÇÃO

Em programação, temos algumas estruturas ou blocos de comandos que auxiliam a realizar tarefas de modo facilitado. As estruturas de seleção, também chamadas de desvio condicional, permitem selecionar uma alternativa dentre várias para determinar um resultado (seja ele verdadeiro ou falso). Vamos conhecer as estruturas de seleção.

4.1. SE..SENAO (*IF..ELSE*)

A estrutura SE (*IF* em PASCAL) é utilizada quando queremos selecionar um resultado dentre 2 possíveis (verdadeiro ou falso), fazendo comparações entre sentenças, testando-as. Para realizar comparações, podemos utilizar os operadores listados abaixo.

OPERADOR	SÍMBOLO	EXEMPLO
Maior	>	a>b
Menor	<	a<b
Maior ou igual	>=	a>=b
Menor ou igual	<=	a<=b
Igual	=	a=b
Diferente	<>	a<>b

Imagine que queremos deteminar se uma pessoa pode ou não obter a carteria de habilitação no Brasil. Faremos isso testando a idade e exibindo uma mensagem na tela. Acompanhe os códigos abaixo.

PASCAL

```
Program CNH
var idade:integer;

Begin
  read(idade);
  if (idade>=18) then
      write ('Apto a dirigir')
  else
      write ('Inapto a dirigir')
End
```

PORTUGOL

```
Algoritmo CNH
var
idade:inteiro

inicio
   leia(idade)
   se (idade>=18) entao
      escreva("Apto a dirigir")
   senao
      escreva("Inapto a dirigir")
   fimse
fimalgoritmo
```

Na primeira linha da estrutura, testamos se a idade é maior ou igual a 18. Se for (resultado verdadeiro), o algoritmo irá executar a primeira opção, escrevendo na tela "Apto a dirigir". Caso

não seja (resultado falso), o algoritmo irá executar a linha SENAO/*ELSE*, escrevendo na tela "Inapto a dirigir". No Portugol é obrigatorio finalizar a estrutura com o comando FIMSE.

4.2. ESCOLHA (*CASE OF*)

A estrutura ESCOLHA (*CASE OF* em PASCAL), funciona de modo semelhante a estrutura SE/*IF*, fazendo a seleção de uma solução dentre várias possíveis. A principal diferença é que as comparações só podem ser feitas com operador de **igualdade**, enquanto que com a estrutura SE/*IF* podemos fazer outras comparações (maior, menor, maior ou igual, menor ou igual). É indicado usar a estrutura ESCOLHA/*CASE OF* quando temos um grande número de possibilidades e, claro, estas puderem ser comparadas com sinal de igualdade.

Imagine que em uma loja, a vendedora deve digitar um código para determinar o modo de pagamento, seguindo a regra: 1 – CARTÃO, 2 – DINHEIRO e 3 – CHEQUE. Veja o exemplo de código para esta situação:

PASCAL	PORTUGOL
Program Pagamento **var codigo:integer;** **Begin** **read(codigo);** **case codigo of** **1 : write ('Cartão');** **2 : write ('Dinheiro');** **3 : write ('Cheque');** **end** **End**	**Algoritmo Pagamento** **var** **codigo:inteiro** **inicio** **leia(codigo)** **escolha(codigo)** **caso 1** **escreva("Cartão")** **caso 2** **escreva("Dinheiro")** **caso 3** **escreva ("Cheque")** **fimescolha** **fimalgoritmo**

No exemplo acima, a estrutura de seleção escolha primeiramente uma variável para "testar" (código). Lembre-se que a estrutura ESCOLHA/*CASE OF* só permite operações de igualdade, isto é, só é possível testar se o código é **igual** a alguma coisa. A variável testada precisa necessariamente não ser do tipo REAL ou então perderemos a precisão do código. Nos dois algoritmos, a variável irá passar por testes e caso seja igual a algum valor (1, 2 ou 3), o código correspondente será executado. No Portugol é obrigatório finalizar o bloco da estrutura com o comando FIMESCOLHA. No Pascal, use apenas o comando *end*.

5. ESTRUTURAS DE REPETIÇÃO

Como o nome sugere, as estruturas de repetição, também chamada de laços, auxiliam o algoritmo na tarefa de repetir linhas de código de modo finito, a fim de evitar a redigitação de algum trecho. Cada estrutura tem sua repetição de modo particular, podendo ser usada livremente de acordo com a lógica do algoritmo. É importante notar que uma estrutura de repetição pode conter, dentro dela, uma ou mais estrutura de seleção e vice-versa. Vamos conhecer as estruturas de repetição.

5.1. PARA (*FOR*)

A estrutura PARA (*FOR* em PASCAL) repete linhas de código em número de vezes definido, isto é, devemos usá-la quando sabemos exatamente quantas vezes o código irá se repetir. Geralmente, para a estrutura PARA/*FOR*, usamos uma variável auxiliar para contar a quantidade de repetições, de modo que o programa não execute nem mais, nem menos vezes do que o necessário (vamos chamar essa variável nesta apostila de "contador"). Esta variável obrigatoriamente deve ser do tipo INTEIRO/*INTEGER* Vamos a um exemplo.

Imagine que o usuário tenha que digitar 10 nomes no sistema. Caso fizessemos no modo tradicional, escreveríamos 10 vezes um comando para ler os dados que o usuário digitar. Como sabemos o número exato de repetições, podemos escrever apenas 1 uma vez o comando e usar a estrutura PARA/*FOR* para repetir 10 vezes. Acompanhe o código de exemplo abaixo.

PASCAL	PORTUGOL

`Program Repetição`
`var contador:integer;`
`  nome: string;`
 	
`Begin`
`  for contador:=1 to 10 do`
`  write('Digite um nome: ')`
`  read(nome);`
`  end`
`End`
	`Algoritmo Repetição`
	`var`
	`nome:literal`
	`contador:inteiro`
	
	`inicio`
	`  para contador de 1 ate 10 faca`
	`  escreva ("Digite um nome: ")`
	`  leia(nome)`
	`  fimpara`
	`fimalgoritmo`

Na início da estrutura, é declarada que a variável CONTADOR irá de 0 até 10. Dentro do bloco, é colocado um único código onde aparece a mensagem na tela e há a leitura da informação digitada pelo usuário. No Portugol é obrigatório finalizar a estrutura com o comando FIMPARA. No Pascal use apenas o comando *end*.

5.2. ENQUANTO (*WHILE*)

Para os casos em que não sabemos a quantidade de vezes que o código deverá ser repetido, temos a estrutura ENQUANTO (*WHILE* no PASCAL). Para repetir, testamos uma sentença e se o resultado for **verdadeiro**, o código se repete. No momento que a sentença passar a ser falsa, o laço é interrompido (o que pode ocorrer a qualquer momento, não sabemos quando). A estrutura ENQUANTO / WHILE tem uma particularidade extremamente importante de notar. O teste da sentença é feito no **início** do laço, portanto, pode ocorrer de o laço não ser executado nenhuma vez, caso a sentença não seja verdadeira logo de início. Vamos a um exemplo.

Imagine que o usuário deva adivinhar um número. Para isso deve ir tentando digitar números e quando acertar o sistema mostrará uma mensagem avisando-o do acerto. Acompanhe o código abaixo.

PASCAL	PORTUGOL

`Program Adivinhação`
`var sorteado, numero:integer;`
 	
`Begin`
`  sorteado:=23;`
`  while(sorteado<>numero) do`
`  begin`
`  write('Tente adivinhar: ')`
`  read(numero);`
`  end`
`  writeln('Parabéns');`
`End`
	`Algoritmo Adivinhação`
	`var`
	`sorteado, numero:inteiro`
	
	`inicio`
	`  sorteado <- 23`
	`  enquanto(sorteado<>numero) faca`
	`  escreva("Tente adivinhar: ")`
	`  leia(numero)`
	`  fimenquanto`
	`  escreval("Parabéns!")`
	`fimalgoritmo`

Nos dois códigos, o número sorteado é 23. O teste do ENQUANTO/WHILE é feito no início resultando em verdadeiro (23 é diferente de 0), portanto o laço é iniciado. A partir daí, o código irá se repetir eternamente nas duas linhas WRITE/ESCREVA e READ/LEIA até o número ser adivinhado. Quando isso ocorrer, a condição inicial passa a ser falsa (23 é igual a 23) e o laço é interrompido, executando por fim, a linha com a mensagem de parabéns.

5.3. REPITA (*REPEAT*)

A estrutura REPITA (*REPEAT* no PASCAL) possui o mesmo principio de funcionamento da estrutura ENQUANTO/*WHILE*, isto é, repete um trecho de código por um número de vezes indefinido, após testar uma sentença. Porém há duas principais diferenças que as distinguem. A primeira delas é que a sentença testada precisa ser **falsa** para que o código possa ser repetido, pois no momento que for verdadeira, a repetição é interrompida. A outra diferença, importantíssima, é que o teste da sentença é feito apenas no **final do código**, isto significa que

as instruções serão executadas ao menos uma vez (na estrutura ENQUANTO/*WHILE* o código corre o risco de não ser executado nenhuma vez, pois o teste é feito no início da estrutura).

Vamos refazer o mesmo exemplo anterior, utilizando a estutura REPITA/*REPEAT*. O usuário tentará adivinhar o número escolhido pelo programa.

PASCAL	PORTUGOL

Program Adivinhação
var sorteado, numero:integer;
 	
Begin
sorteado:=23;
repeat
write('Tente adivinhar: ')
read(numero);
until (numero=23)
writeln('Parabéns');
End
	Algoritmo Adivinhação
	var
	sorteado, numero:inteiro
	
	inicio
	sorteado <- 23
	repita
	escreva("Tente adivinhar: ")
	leia(numero)
	ate (numero=23)
	fimrepita
	escreval("Parabéns!")
	fimalgoritmo

Após inserir o número 23 na variável "sorteado", entramos na estrutura de repetição REPITA/*REPEAT*. Note que neste ponto não há nenhum teste de sentença, o que faz com que o código continue sendo executado normalmente. Após as linhas ESCREVA/*WRITE* e LEIA/*READ* dentro da estrutura, há o teste com o comando ATE/*UNTIL*. Se o resultado do teste for **falso**, o código irá se repetir até que esta seja verdadeira, isto é, até que o usuário acerte o número. No PORTUGOL é necessário finalizar a estrutura com o comando FIMREPITA. Ao sair da estrutura, é exibida a linha com os "parabéns".

Podemos também forçar a saída de um laço de repetição por algum motivo. Para isso usamos o comando INTERROMPA/*EXIT* que interrompe o laço e executa a próxima instrução. Note também que na linguagem PASCAL não é necessário incluir as instruções *BEGIN/END* ao utilizar a estrutura *REPEAT*, como acontece nas outras estruturas de repetição.

6. COLEÇÕES DE DADOS

Em qualquer linguagem de programação, as variáveis guardam apenas um valor. Algumas linguagem implementam o conceito de coleção, onde é possível armazenar vários valores em apenas uma variável. No PORTUGOL chamamos esta variável de VETOR, enquanto no PASCAL chamamos de *ARRAY*. Mas como localizamos os dados em uma coleção?

Em VETORES/*ARRAYS*, os dados vão sendo armazenados e organizados em uma sequencia lógica e **numerada**, como um índice de um livro. Cada informação fica armazenada em um número ordenado de acordo com a quantidade de espaços ou posições disponíveis. Imagine que desejamos armazenar vários nomes de clientes em um sistema. Ao invés de criar várias variáveis do tipo literal/*string*, podemos criar um VETOR/*ARRAY*. Veja um exemplo gráfico:

João	José	Juca	Maria	Benedita	Mário	Ana
1	2	3	4	5	6	7

Acima temos a representação gráfica de uma coleção de dados do tipo literal/*string*. Note os índices identificando cada informação que é gravada, ficando fácil para localizar ou manipular as informações dentro da coleção. Algumas linguagens de programação definem por padrão o número 0 (zero) como o primeiro índice da coleção, o que pode alterar a lógica de seu programa.

6.1. PERCORRENDO COLEÇÕES

Frequentemente, as coleções precisam ser percorridas, isto é, todos os seus espaços precisam ser conferidos e analisados. A forma mais comum de realizar essa operação é utilizando as estruturas de repetição. Como as coleções sempre possuem um número definido de espaços para guardar as informações, a estrutura PARA/*FOR* é quase sempre a mais indicada, pois precisaremos definir um o início e o fim da coleção de modo explícito. A estrutura PARA/*FOR* evita também um erro clássico de lógica que é tentar acessar um índice inexistente na coleção (algumas linguagens chamam o erro de *Null Pointer Exception*), já que a contagem dos índices é explícita. Tanto no PORTUGOL quanto no PASCAL, representamos as coleções com o símbolo de colchetes ([ ]). Veja a representação de código da declaração de uma coleção.

PASCAL	PORTUGOL
`nomedacolecao:array[x..y] of tipo;`	`nomedacolecao:vetor[x..y] of tipo`
`Ex: clients:array[1..7] of string;`	`Ex: clientes:vetor[1..7] de literal`

Primeiro definimos o nome da coleção, seguindo pela palavra VETOR/*ARRAY.* Dentro dos colchetes colocamos as dimensões da coleção, isto é o número de seus índices. Por fim, definimos o tipo de dado que a coleção irá guardar, no caso literal/*string*. Como fazemos então para percorrer a coleção?

Vamos utilizar como exemplo, a coleção de nomes anterior. Imagine que desejamos imprimir na tela os nomes contidos em todas as posições. Acompanhe o código.

PASCAL	PORTUGOL

`Program Repetição`
`var indice:integer;`
`clientes:array[1..7] of string;`
 	
`Begin`
`  for indice:=1 to 7 do`
`  write(clientes[indice])`
`  end`
`End`
	`Algoritmo Repetição`
	`var`
	`indice:inteiro`
	`clientes:vetor[1..7] de literal`
	
	`inicio`
	`  para indice de 1 ate 7 faca`
	`  escreva (clientes[indice])`
	`  fimpara`
	`fimalgoritmo`

Vamos entender o código acima. O vetor é declarado como na explicação anterior. Note que há uma variável inteira (*indice*) responsável por controlar os índices da coleção. Iniciamos então a estrutura PARA/*FOR,* contando de 1 até 7, isto é, repetindo o código 7 vezes. Para cada repetição, iremos escrever na tela o valor da coleção na posição que estiver valendo a variável *indice*, isto é, de 1 a 7, um de cada vez. Desse modo, conseguimos passar por todas as posições da coleção, imprimindo o valor na tela de cada índice.

6.2. COLEÇÕES MULTIDIMENSIONAIS

Há um tipo especial de coleção, chamado de matriz, cujos índices são multidimensionais, divididos em linhas e colunas. A forma de percorrer uma matriz é semelhante à de uma coleção unidimensional, porém com algumas adaptações. Primeiramente, vejamos a forma de declaração de uma matriz:

PASCAL	PORTUGOL
`nomedamatriz:array[x,y] of tipo;`	`nomedamatriz:vetor[x,y] of tipo`
`Ex:clientes:array[1..3,1..3] of string;`	`Ex:clientes:vetor[1..3,1..3] de literal`

Na declaração de matrizes, dentro dos colchetes especificamos os índices das linhas e os índices das colunas, separadas por vírgula. No exemplo acima, estamos criando uma matriz de 3 linhas

e 3 colunas, por isso a repetição de índices. Agora temos 9 posições para manipular. Vamos preencher as posições com nomes. Acompanhe a representação gráfica da matriz 3x3.

João	José	Juca
Maria	Benedita	Mário
Ana	Isabel	Antônio

Para acessar as posições precisamos informar a linha e a coluna, nesta ordem. Pense agora e responda. Qual a posição da informação "Mário"? Acertou se pensou [2,3]. E da informação "Ana"? Acertou se pensou [3,1].

Para percorrer as matrizes, utilizamos também a estrutura PARA/*FOR*, porém duas vezes, uma para as linhas e outra para as colunas, com as duas estruturas sendo executadas em conjunto. Para entendermos melhor, acompanhe o código abaixo e a explicação. Iremos percorrer a matriz anterior mostrando os nomes na tela.

PASCAL

```
Program Repetição
var linha, coluna:integer;
nomes:array[1..3][1..3] of string;

Begin
  for linha:=1 to 3 do
     for coluna:=1 to 3 do
      write(nomes[linha, coluna]);
     end
   end
End
```

PORTUGOL

```
Algoritmo Repetição
var
linha, coluna:inteiro
nomes:vetor[1..3][1..3] de literal

inicio
   para linha de 1 ate 3 faca
     para coluna de 1 ate 3 faca
       escreva (nomes[linha, coluna])
     fimpara
   fimpara
fimalgoritmo
```

Criamos para esse caso, duas variáveis para representar a linha e a coluna que queremos. Depois de declarada a matriz, entramos na estrutura PARA/*FOR* repetindo a variável *linha* 3 vezes e **dentro** desta mesma estrutura abrimos outro laço PARA/*FOR* para as colunas. Isto significa que na primeira repetição da variável *linha*, a variável *coluna* irá se repetir 3 vezes, passando então para a segunda repetição da linha com mais 3 repetições da coluna e assim por diante, até o final do laço. Na prática, iremos a visitar a linha 1 e depois as colunas 1, 2 e 3. Depois passamos para a linha 2 e novamente pelas colunas 1, 2 e 3. Por fim, na linha 3, ocorre o mesmo processo. Dessa forma, conseguimos passar por todas as posições da matriz.

7. FUNÇÕES E PROCEDIMENTOS

Funções e procedimentos (*functions* e *procedures* no PASCAL) podem ser vistos como blocos de código que são executados de forma separada do código principal. Normalmente são criadas para representar ações que ocorrem com frequência no sistema, em diferentes tempos, de modo a evitar que o mesmo código seja digitado mais de uma vez. Por exemplo, em um sistema, um botão é responsável por salvar o cadastro de clientes. É desnecessário digitar o código correspondente toda vez que precisarmos salvar os dados (clicando no botão). Para resolver isso, pode-se criar uma função/procedimento contendo o código e quando precisarmos acioná-la basta escrever o nome da função ou procedimento desejado.

Funções e procedimentos funcionam como um mini programa dentro do programa principal, com o objetivo de reduzir o código principal e tornar fácil o entendimento e a visualização do algoritmo como um todo. Este mini programa são escritos no início do código principal e podem ser acionados de qualquer ponto, bastando escrever o nome correspondente. A grande diferença entre os códigos é que uma função sempre retorna um valor, enquanto o procedimento nunca retorna valores.

7.1. DECLARANDO FUNÇÕES

As funções podem retornar um (e apenas um) valor para o código principal, se necessário. Uma função pode possuir suas próprias variáveis e seu próprio corpo de código, além de especificar o tipo de retorno (texto ou número). Veja abaixo a sintaxe de uma função.

PASCAL	PORTUGOL

`function nome (parâmetros):retorno`
`var`
`  <variaveis da função>`
 	
`Begin`
`  <commandos>`
`  nome:= <valor>`
`End;`
	`funcao nome (parâmetros):retorno`
	`var`
	`  <variaveis da função>`
	
	`inicio`
	`  <comandos>`
	`  retorne <valor>`
	`fimfuncao`

O nome da função segue as regras dos nomes de variáveis. Os parâmetros são variáveis vindas do programa principal que são repassadas para a função. O tipo de retorno pode ser literal/*string*, inteiro/*integer*, real ou lógico/*boolean*. Pode-se também declarar variáveis adicionais para auxiliar o processamento da função. Essas variáveis são usáveis apenas para a função, não tendo utilidade no código principal (nem sequer são reconhecidas). No código da função em PORTUGOL, temos o comando RETORNE em que é especificado o valor ou variável de retorno. No PASCAL, para especificar o retorno, basta atribuir o valor desejado ao nome da

função. No PORTUGOL, a função é finalizada com o comando FIMFUNCAO, enquanto no PASCAL colocamos um ponto e vírgula após a palavra END.

Vamos aplicar um exemplo. Criaremos uma função para somar 2 números reais, com retorno também do tipo real. Acompanhe o código.

PASCAL	PORTUGOL

`function somar (n1,n2:real):real`
`var total:real;`
 	
`Begin`
`  total:=n1+n2;`
`  somar:=total;`
`End;`
	`funcao nome (n1, n2:real):real`
	`var`
	`  total:real`
	
	`inicio`
	`  total<-n1+n2`
	`  retorne total`
	`fimfuncao`

A função recebe 2 parâmetros do tipo real, retornando também um valor real. Criamos também uma variável adicional chamada "Total", que irá ser retornada ao final do código. Note que esta variável serve apenas para a função, não podendo ser aproveitada no código principal. No código da função, somamos os dois valores recebidos e atribuímos a variável "Total" ao resultado. Logo após, há o comando de retorno. A partir daí, no código principal, podemos usar a função quantas vezes forem necessárias, bastando para isso chamá-la pelo nome, passando os valores desejados.

Ex: somar(10, 5)

Tanto no PORTUGOL quanto no PASCAL, existem funções já pré-definidas que podem ser usadas igualmente inúmeras vezes. Vamos conhecer algumas das funções.

FUNÇÃO PORTUGOL/PASCAL	DESCRIÇÃO
Maiusc/UpCase	Converte em maiúsculo. Ex: maiusc("teste")
Asc/Chr	Retorna o código ASCII. Ex: chr('A')
Int	Converte o valor para inteiro. Ex: int(2.5)
Raizq/sqrt	Retorna a raiz quadrada. Ex: sqrt(25)
Quad/sqr	Eleva ao quadrado. Ex: quad(8)
Exp	Operações de potência. Ex: exp(5,2)
Compr/length	Retorna o tamanho de um texto. Ex: compr("teste")

7.2. DECLARANDO PROCEDIMENTOS

Procedimentos são semelhantes às funções, porém não retornam valor, realizando apenas a operação simples especificada. Acompanhe a sintaxe do código abaixo.

PASCAL	PORTUGOL

procedure nome (parâmetros)
var
<variáveis do procedimento>
 	
Begin
<commandos>
End;
	procedimento nome (parâmetros)
	var
	<variáveis do procedimento>
	
	inicio
	<comandos>
	fimprocedimento

Em uma rápida análise, percebemos que a única diferença de um procedimento para uma função é a ausência de retorno. Desse modo, podemos dizer que um procedimento não gera dados para o código principal. Isso não significa que um procedimento não possa utilizar comandos de saída como ESCREVA/*WRITE*. Vamos produzir um exemplo de um procedimento que analisa e calcula o maior entre dois números utilizando a estrutura SE/*IF*, escrevendo essa informação na tela principal do programa. Acompanhe o código.

PASCAL	PORTUGOL

procedure maior (n1,n2:real)
 	
Begin
if (n1>n2) then
write('O maior é:',n1)
else
write('O maior é:',n2);
End;
	procedimento maior (n1, n2:real)
	
	inicio
	se (n1>n2) então
	escreva ("O maior é", n1)
	senao
	escreva ("O maior é", n2)
	fimse
	fimprocedimento

Note que este exemplo não utiliza variáveis adicionais. Após receber os dois números, utilizamos a estrutura condicional SE/*IF* para testar, exibindo o resultado na tela. A partir do código principal, o procedimento pode ser chamado, passando os parâmetros desejados.

Ex: maior(10, 5)

8. EXERCÍCIOS

8.1 Escrever um algoritmo que lê o nome de um funcionário, o número de horas trabalhadas, o valor que recebe por hora e o número de filhos. Com estas informações, calcular o salário deste funcionário, sabendo que para cada filho, o funcionário recebe 3% a mais, calculado sobre o salário bruto.

8.2 Escrever um algoritmo que receba a idade de uma pessoa em anos e mostre a mesma idade em dias.

8.3 Faça um algoritmo que leia o nome de um piloto, uma distância percorrida em km e o tempo que o piloto levou para percorrê-la (em horas). O programa deve calcular a velocidade média em km/h, e exibir a seguinte frase: *A velocidade média do <nome do piloto> foi <velocidade media calculada> km/h.*

8.4 Considere um terreno retangular onde será construída uma casa redonda. Escrever um algoritmo que receba as medidas do terreno e da casa, calcule a área de ambos e mostre o tamanho da área livre em m².

8.5 Escreva um algoritmo que leia um número inteiro e analise se é positivo ou negativo. Caso seja positivo, indique se é impar ou par. Caso seja negativo, escrever a mensagem "Não é positivo".

8.6 Desenvolva um algoritmo que leia o número do mês e mostre seu nome. Para exceções, exibir a mensagem "Mês inexistente!"

8.7 Escreva um algoritmo que receba o número de eleitores de uma cidade, os votos válidos, os votos brancos e os votos nulos. Mostre ao final, a porcentagem de cada um em relação ao total de eleitores.

8.8 Desenvolva um algoritmo que analise o salário de uma pessoa e um valor para financiamento de veículo. Caso o valor financiado seja menor ou igual a 5 vezes o salário, o programa deverá escrever a mensagem "FINANCIAMENTO CONCEDIDO". Caso contrário, escrever "FINANCIAMENTO NEGADO". Em qualquer caso, escrever a frase "Obrigado por nos consultar".

8.9 Faça um algoritmo que leia 10 valores e os exiba na ordem contrária em que foi digitado.

8.10 Escrever um algoritmo que leia 5 valores e mostre quantos destes são negativos.

8.11 Faça um algoritmo que receba 10 valores e mostre o quadrado de cada um deles.

8.12 Escrever um algoritmo para ler uma matriz 7x4, com valores inteiros distintos. Encontrar o menor valor e sua posição.

8.13 Crie uma matriz 2x2, preenchida com números inteiros e mostre a soma de suas diagonais.

8.14 Vamos construir uma calculadora utilizando funções ou procedimentos. No código principal, deve ser informado 2 números inteiros e o código da operação. Para código 1 faça soma, para 2 faça subtração, para 3 realize a multiplicação e para 4 faça a divisão. Exibir o resultado na tela.

8.15 Construa um algoritmo que receba um valor inteiro e informe sua tabuada do 1 ao 10.

8.16 Juberlei tem 1,10m e cresce 3 cm por ano. Pururuca tem 1,50m e cresce 2 cm por ano. Construa um algoritmo que mostre em quantos anos Juberlei será maior que Pururuca.

8.17 Um pai atrasado resolveu sair para fazer as compras de material escolar. Para ajudá-lo nos cálculos, desenvolva um algoritmo em que o pai possa digitar o nome e o valor de cada material comprado (máximo de 10 itens). Quando o pai digitar "FIM", o programa deve mostrar a lista de materiais e o preço total da compra.

8.18 Construa um algoritmo para calcular o fatorial de um número escolhido pelo usuário. O fatorial é obtido pelo produto de seus números anteriores Ex: 5! = 5x4x3x2x1

8.19 Crie uma matriz 2x2 e preencha com números inteiros distintos. Peça ao final para o usuário digitar um número e mostre uma mensagem na tela dizendo se o número está ou não na matriz.

8.20 Suponha que um caixa eletrônico trabalhe com notas de 100, 50, 20 e 10 reais. Construa um algoritmo que receba um valor de saque e indique a quantidade de notas de cada espécie devem ser emitidas. Apresentar 2 opções de saques, com maioria de notas de 100 e de 50 reais.

9. REFERÊNCIAS

NAPRO – Núcleo de Apoio à Aprendizagem de Programação – Universidade de Caxias do Sul
http://www.apoioinformatica.inf.br/
http://www.consiste.dimap.ufrn.br/~david/
http://www.inf.pucrs.br/%7Egidio/algo1/
http://www.inf.pucrs.br/~fldotti/lapro1/prfun.htm
http://pt.wikibooks.org/wiki/Pascal/Estrutura_de_repetição

==========

Apostila produzida pelo Prof. Esp. Jonas W R. Aureliano.