Jaime Evaristo
Sérgio Crespo

APRENDENDO A PROGRAMAR PROGRAMANDO NUMA LINGUAGEM ALGORÍTMICA EXECUTÁVEL (ILA)

Segunda Edição
Formato Digital

Jaime Evaristo
Professor da Universidade Federal de Alagoas
Sérgio Crespo
Professor da Universidade do Vale do Rio dos Sinos

Aprendendo a Programar

Programando numa linguagem algorítmica executável (ILA)

Segunda Edição
Formato digital
Maceió
2010

Prefácio (à primeira edição)

Mesmo nas questões menos difíceis, a programação de computadores, no seu início considerada uma arte, hoje considerada uma ciência, requer a aplicação de raciocínios heterodoxos em relação aos raciocínios utilizados nas soluções dos problemas de outras ciências. Por exemplo, os problemas elementares da Física tratam basicamente de relacionamentos entre grandezas físicas e na aplicação destes relacionamentos está a maioria das suas soluções. Em programação, a tentativa de solução de uma questão é, basicamente, a construção de uma sequências de instruções, selecionadas dentre um número finito delas, que a sua execução redunda na concretização de uma tarefa, tarefa especificada na própria questão. Em programação não se trabalha com grandezas nem com fórmulas. Escrever um programa para resolver questões que um ser humano com um conhecimento razoável é capaz de resolver facilmente, como, por exemplo, determinar o maior número de uma relação de números, requer um tipo de raciocínio não usual em outros campos do conhecimento humano. Raciocínios não usuais, porém agradáveis de serem desenvolvidos e indispensáveis para aqueles que pretendem lidar com algum ramo da computação.

Além de raciocínios específico (cujo conjunto é chamado *lógica de programação*), a programação de computadores, num nível inicial, possui uma outra característica que contribui para que a sua aprendizagem seja ímpar: o reduzido número e a simplicidade de conceitos que devam ser aprendidos.

Existem divergências com relação ao melhor modo de iniciar a aprendizagem de programação: utilizando-se uma pseudolinguagem ou através de uma linguagem de programação. Os que defendem a primeira hipótese alegam que a utilização de uma pseudolinguagem permitiria ao aluno aprender de uma forma mais informal, sem a necessidade de aprender aspectos técnicos de um sistema de computação, permitindo que todo o seu esforço fosse dedicado ao desenvolvimento da lógica de programação. Os que defendem a segunda estratégia, alegam que execução, num computador, dos programas desenvolvidos pelo aprendizando trazem um aspecto lúdico ao processo de aprendizagem que só o beneficia.

Com o *ILA* (*Interpretador de Linguagem Algorítmica*) a divergência acima referida pode diminuir significativamente. Devido à simplicidade do formato da linguagem algorítmica utilizada pelo ILA, podemos desenvolver algoritmos utilizando um português estruturado. Devido à simplicidade do interpretador, podemos desenvolver a aprendizagem de programação utilizando uma "linguagem de programação" sem se preocupar muito com os aspectos técnicos. Podemos aprender a programar programando.

O objetivo deste livro é aprender a programar programando. Com isso, o livro busca enfatizar e contextualizar os conceitos teóricos de modo a facilitar a compreensão do referencial teórico ali inserido. O livro apresenta diversos exercícios resolvidos, muitos deles bem significativos em relação ao desenvolvimento da lógica de programação. Centrado neste foco, o livro apresenta vários exercícios propostos, solicitando programas cuja confecção contribui sobremaneira para a aprendizagem.

Para atingir o objetivo explicitado acima, o livro contém nove capítulos. O capítulo 1, intitulado Introdução à Programação, trata dos seguintes assuntos: organização básica de um computador, linguagem de máquina, algoritmos, lógica de programação, linguagens de alto nível, sintaxe e semântica de um comando e sistemas de computação. O capítulo 2, Introdução à Linguagem Algorítmica, discute variáveis simples, expressões aritméticas, relações e expressões lógicas, estrutura e execução de um programa, comandos de entrada, de saída e de atribuição e funções pré-definidas.

Por seu turno, o capítulo 3, com o título Estruturas de Seleção, discorre sobre os comandos *Se* e *Faca caso*, enquanto que o capítulo 4, Estruturas de Repetição apresenta os comandos *Para proximo* e *Faca enquanto*. O capítulo 5 estuda as funções e a recursividade e o capítulo 6, as variáveis compostas.

Já os capítulos 7 e 8 estudam, respectivamente, pesquisa e ordenação e cadeia de caracteres, enquanto que, finalmente, o capítulo 9 apresenta, com alguns comentários, todos programas que

solucionam os exercícios propostos.

Naturalmente, o leitor deve resistir a tentação de consultar a solução de um problema proposto diante da primeira dificuldade em resolvê-lo. Aprende-se a programar programando e ao se escrever um programa invariavelmente se comete erros. A descoberta destes erros, a procura de formas de solucioná-los são ações que facilitam o desenvolvimento da lógica de programação. Neste sentido, é importantíssimo implementar as soluções obtidas no *ILA* (disponível para *download* em http://inf.unisinos.br/~crespo) e executar os programas correspondentes para que, estando a solução correta, tenha-se a sensação de vitória e o desejo de resolver outras questões; estando a solução incorreta, procure-se aprender com os erros cometidos.

Como também estamos aprendendo, quaisquer críticas e sugestões dirigidas a jaime@ccen.ufal.br ou a crespo@inf.puc-rio.br serão bem vindas.

Jaime Evaristo
Sérgio Crespo

Prefácio (à segunda edição)

Esta é a edição digital do livro Aprendendo a Programar Programando numa Linguagem Algorítmica Executável publicado em 2000, no formato impresso, pela Editora Book Express, baseada na cidade do Rio de Janeiro. O esgotamento da edição impressa e o encerramento da atividades da editora ensejaram-nos a oportunidade de disponibilizar o conteúdo do livro, que pode ser utilizado como livro texto em disciplinas do tipo Lógica de Programação ou para aprendizagem autodidática de programação de computadores.

Ao contrário da versão impressa, esta edição não contém as respostas dos exercícios propostos. Para obter o caderno das soluções dos exercícios basta encaminhar e-mail para jaime@ccen.ufal com assunto RESPOSTAS LIVRO ILA e contendo NOME, CATEGORIA (docente, discente, autodidata), CIDADE/ESATADO e, se for o caso, INSTITUIÇÃO/CURSO.

Os autores aproveitam a ocasião para agradecer aos vários colegas que têm referido o livro nos planos pedagógicos de suas disciplinas e aos pesquisadores que o citam nos seus trabalhos de pesquisa.

Sendo uma edição digital, correções e inclusões no texto podem ser feitas a qualquer momento. Assim, os autores rogam a participação dos leitores no sentido da melhoria do livro (inclusive, com a inclusão de novos exercícios) e prometem registrar estas intervenções. Toda e qualquer observação deve ser encaminhada para jaime@ccen.ufal.br com o assunto LIVRO ILA.

Em julho de 2010
Jaime Evaristo
Sérgio Crespo

Sumário

1 Introdução à Programação

1.1 Organização básica de um computador

Um computador é constituído de quatro unidades básicas: *unidade de entrada*, *unidade de saída*, *unidade de processamento central* e *memória*. Uma *unidade de entrada* é um dispositivo que permite que o usuário interaja com o computador, fornecendo-lhe dados e informações que serão processadas. O *teclado* o seu exemplo mais trivial. Uma *unidade de saída*, por seu turno, serve para que sejam fornecidos ao usuário do computador os resultados do processamento realizado. O *monitor de vídeo* e uma *impressora* são exemplos de unidades de saída. A *unidade central de processamento* é responsável por todo o processamento requerido, sendo muito conhecida por ***cpu***, acrossemia de ***c****entral* ***p****rocessing* ***u****nit*. Já a *memória* armazena temporariamente dados e informações que serão utilizados no processamento.

1.2 Linguagem de máquina

Linguagens de comunicação

Evidentemente, há a necessidade de que as unidades que compõem um computador se comuniquem. Por exemplo, um dado fornecido pelo teclado deve ser armazenado na memória; para a *cpu* realizar uma operação aritmética, ela vai "buscar" valores que estão armazenados na memória, e assim por diante. Para que haja comunicação entre as unidades do computador é necessário que se estabeleça uma *linguagem*.

Os seres humanos se comunicam através de várias linguagens. Entre elas estão as linguagens escrita e e oral. Uma comunicação através de uma linguagem escrita é constituída de *parágrafos*, os quais contêm *períodos*, que contêm *frases,* que são constituídas de *palavras*, sendo cada uma das palavras formadas por *letras* e esta seqüência termina aí. Assim, uma *letra* é um ente indivisível da linguagem escrita e, em função disto, é chamada *símbolo básico* desta linguagem. Este exemplo foi apresentado para que se justifique a afirmação de que toda linguagem requer a existência de *símbolos básicos,* como os *fonemas* para a linguagem falada.

A linguagem de comunicação entre as unidades

Como a comunicação entre as unidades do computador teria que ser obtida através de fenômenos físicos, os cientistas que conceberam os computadores atuais estabeleceram dois símbolos básicos para a linguagem. Esta quantidade de símbolos foi escolhida pelo fato de que através de fenômenos físicos é muito fácil obter dois estados distintos e não confundíveis, como passar corrente elétrica/não passar corrente elétrica, estar magnetizado/não estar magnetizado, etc., podendo cada um destes estados ser um dos símbolos. Assim a linguagem utilizada para comunicação interna num computador, chamada *linguagem de máquina*, possui apenas dois símbolos. Cada um destes símbolos é denominado *bit* (***bi****nary digi****t***) e eles são representados por 0 (zero) e 1 (um). Esta forma de representar os *bit's* justifica a sua denominação: *binary digit*, que significa dígito binário (além disto, *bit* em inglês significa fragmento). Portanto, as *palavras* da linguagem de máquina são seqüências de bits, ou seja, seqüências de dígitos zero e um.

O código ASCII

Para que haja a possibilidade da comunicação do homem com o computador, é necessário que as palavras da linguagem escrita sejam traduzidas para a linguagem de máquina e vice-versa. Para que isto seja possível, é necessário que se estabeleça qual a seqüência de bit's que corresponde a cada caractere usado na linguagem escrita. Ou seja, é necessário que se estabeleça uma codificação em seqüência de bit's para cada um dos caracteres. Uma codificação muito utilizada é o *código*

ASCII (*American Standard Code for Information Interchange* ou *Código Padrão Americano para Intercâmbio de Informações*), estabelecido pelo *ANSI* (*American National Standards Institute*). Nesta codificação, cada caractere é representado por uma seqüência de oito bits (normalmente, um conjunto de oito bit's é chamado *byte*). Só para exemplificar (será visto ao longo do livro que, em geral, não há necessidade de que se conheça os códigos dos caracteres), apresentamos a tabela abaixo com os códigos ASCII de alguns caracteres.

Tabela 1 Códigos ASCII de alguns caracteres

Caractere	Código ASCII
Espaço em branco	00100000
!	00100001
"	00100010
. . .	. . .
0	00110000
1	00110001
. . .	. . .
A	01000001
B	01000010
. . .	. . .
Z	01011010
. . .	. . .
a	01100001
. . .	. ..

Observe a necessidade de se haver codificado o *espaço em branco* (este "caractere" é utilizado para separar nossas palavras) e de se haver codificado diferentemente as letras maiúsculas e minúsculas, para que se possa considerá-las como coisas distintas.

Levando em conta que cada seqüência de zeros e uns pode ser vista como a representação de um número inteiro no *sistema binário de numeração* [Evaristo, J 2010], podemos, para facilitar referências, associar a cada código ASCII o inteiro correspondente, obtendo assim o que se costuma chamar de *código ASCII decimal*. Por exemplo, como 1000001 é a representação no sistema binário do inteiro 65 do sistema decimal, dizemos que o *código ASCII decimal* de A é 65.

1.3 Programas de computadores

Para que um computador tenha alguma utilidade, ele deve executar um *programa* que tenha uma finalidade específica. *Games* são programas que têm como objetivo propiciar entretenimento aos seus usuários. *Processadores de texto* são programas que permitem que textos sejam digitados, impressos e armazenados para futuras modificações ou impressões. *Planilhas eletrônicas* são programas que oferecem recursos para manipulação de tabelas de valores numéricos. *Navegadores* permitem acessos a páginas da *internet*, a rede mundial de computadores. Estes programas destinam-se a *usuários finais*, aquelas pessoas que vão utilizar o computador com um objetivo específico, usando para tal um programa que ela aprendeu a usar, não tendo nenhuma preocupação relativa ao funcionamento interno do sistema computador/programa. Por exemplo, um usuário de um processador de texto deve aprender o que fazer para que o processador destaque em **negrito** alguma parte do texto ou localize uma palavra, não havendo necessidade de saber como o programa realiza estas ações.

Na verdade, para que um processador de texto propicie ao usuário a possibilidade de que textos sejam digitados, corrigidos, gravados, inseridos em outros textos e de que palavras sejam localizadas dentro de um texto, é necessária a execução de muitas instruções com objetivos bem mais específicos e restritos. Um *programa de computador* é, na realidade, um conjunto de instruções que podem ser executadas pelo computador, de tal forma que a execução de subconjuntos destas instruções permitem a realização de ações mais genéricas.

É muito grande o número de instruções dos programas citados acima, chegando à casa dos

milhares. Rigorosamente falando, um programa dos acima citados são conjunto de programas menores, cada um deles com objetivos mais restritos, e que podem ser executados de forma integrada. É comum se utilizar a palavra oriunda do inglês *software* para designar um conjunto de programas com objetivos mais restritos que, sendo executados de forma integrada, propiciam a execução de ações bem mais genéricas.

A parte da Ciência da Computação que trata do desenvolvimento de *softwares* é denominada *Engenharia de Software*. O estudo da *Engenharia de Software* deve ser precedido da aprendizagem do desenvolvimento de programas "menores", ação que comumente é denominada de *Programação de Computadores*.

1.4 Lógica de programação

Sendo um conjunto de instruções cujas execuções redundam na realização da tarefa para a qual foi desenvolvido, o desenvolvimento de um programa requer a utilização de um raciocínio ímpar em relação aos raciocínios utilizados na solução de problemas de outros campos do saber. Por exemplo (e de forma simplificada) ao se tentar resolver um problema de Mecânica Newtoniana deve-se procurar capturar da especificação da questão as grandezas físicas envolvidas e aplicar as fórmulas que relacionam estas grandezas.

Para se desenvolver um programa que resolva um determinado problema é necessário que encontremos uma seqüência de instruções que cujas execuções resultem na solução da questão. É comum se utilizar a termo *algoritmo* para indicar uma seqüência de instruções que resolvem um dado problema, ficando, neste caso, o termo *programa* para indicar um algoritmo que pode ser executado por um computador. A *Lógica de Programação* pode ser entendida como o conjunto de raciocínios utilizados para o desenvolvimento de algoritmos (e, portanto, de programas).

Por exemplo, imagine a seguinte questão: um senhor, infelizmente bastante gordo, está numa das margens de um rio com uma raposa, uma dúzia de galinhas e um saco de milho. O senhor pretende atravessar o rio com suas cargas, num barco a remo que só comporta o senhor e uma das cargas. Evidentemente, o senhor não pode deixar em uma das margens, sozinhos, a raposa e a galinha, nem a galinha e o milho. A questão é escrever um algoritmo que oriente o senhor a realizar o seu intento. Naturalmente, na primeira viagem, ele não pode levar a raposa (neste caso, as galinhas comeriam o milho), nem o milho (caso em que a raposa devoraria as galinhas). Logo, na primeira viagem ele deve levar as galinhas. Como ele estará presente na chegada, na segunda viagem ele pode levar a raposa ou o milho. Mas, e a volta para apanhar terceira carga? A solução é ele voltar com as galinhas e, aí, atravessar o milho, já que não há problema em que a raposa e o milho fiquem juntos. Escrevendo as instruções na seqüência em que elas devem ser executadas, teremos o seguinte algoritmo.

1. Atravesse as galinhas.
2. Retorne sozinho.
3. Atravesse a raposa.
4. Retorne com as galinhas.
5. Atravesse o milho.
6. Retorne sozinho.
7. Atravesse as galinhas.

1.5 Resolução de problemas

Uma pergunta que o leitor pode estar se fazendo é: como vou "descobrir" que a primeira instrução deve ser a travessia das galinhas?

Algumas tarefas para as quais se pretende escrever um algoritmo podem ser vistas como um problema a ser resolvido. O exemplo anterior é um exemplo claro de uma tarefa com esta característica. Existem algumas técnicas que podem ser utilizadas para a resolução de problemas.

No exemplo anterior, para se definir qual seria a primeira instrução, como existem apenas três possibilidades, verifica-se o que aconteceria ao se escolher determinada instrução. Foi o que, de passagem, foi feito acima: se o homem atravessasse primeiro o milho, a raposa devoraria as galinhas; se o homem atravessasse primeiro a raposa, as galinhas comeriam o milho. Neste caso, podemos dizer que foi utilizada a técnica da exaustão: como o número de alternativas era pequeno, analisamos todas elas, uma a uma.

Esta técnica pode ser utilizada também na solução do seguinte problema: dispõe-se de três esferas idênticas na forma, sendo duas delas de mesmo peso e a terceira de peso maior. A questão é descobrir qual a esfera de peso diferente, realizando-se apenas uma pesagem numa balança de dois pratos. Para isto chamemos de A e B as esferas de mesmo peso e de C a de maior peso. Se optarmos por colocar duas esferas num dos pratos (indicando por X + Y, o fato de colocarmos as esferas X e Y num mesmo prato da balança) e a outra esfera no outro, temos as seguintes possibilidades:

a) (A+B, C).
b) (A+C, B).
c) (B+C, A).

No primeiro caso, pode acontecer qualquer coisa: a balança pode ficar equilibrada, se Peso(C) = Peso(A+B); ficar inclinada para o lado esquerdo, se Peso(C) > Peso(A+B) ou ficar inclinada para o lado direito se Peso(C) < Peso(A+B). Observe que nada pode distinguir a esfera C. Nos dois últimos casos, a balança se inclinará para a esquerda, mas, outra vez, nada distingue a esfera C. Por exaustão, resta então escolhermos duas esferas e colocarmos cada uma delas num dos pratos da balança. Agora os casos possíveis são:

a) (A, B).
b) (A, C).
c) (B, C).

No primeiro caso, a balança ficará equilibrada, o que indica que a mais pesada é aquela não escolhida; nos outros dois casos, a balança se inclinará para a direita, indicando que a esfera mais pesada é aquela que ocupa o prato respectivo. Temos então o seguinte algoritmo:

1. Escolha duas esferas.
2. Coloque cada uma das esferas escolhidas num dos pratos da balança.
3. Se a balança ficar equilibrada, forneça como resposta a esfera não escolhida; caso contrário, forneça como resposta a esfera do prato que está num nível mais baixo.

Uma outra técnica de resolução de problemas consiste em se tentar resolver casos particulares da questão ou resolver a questão para dados menores do que os dados que foram fixados. Para exemplificar, consideremos a seguinte questão: como obter exatamente 4 litros de água dispondo de dois recipientes com capacidades de 3 litros e 5 litros[1]? Como 4 = 3 + 1 ou 4 = 5 – 1 conseguiremos resolver a questão se conseguirmos obter 1 litro. Mas isto é fácil, pois 1 = 3 + 3 – 5! Temos então o seguinte algoritmo:

1. Encha o recipiente de 3 litros.
2. Transfira o conteúdo do recipiente de 3 litros para o recipiente de 5 litros.
3. Encha o recipiente de 3 litros.
4. Com o conteúdo do recipiente de 3 litros, complete o recipiente de 5 litros.
5. Esvazie o recipiente de 5 litros.
6. Transfira o conteúdo do recipiente de três litros para o recipiente de 5 litros.
7. Encha o recipiente de 3 litros.
8. Transfira o conteúdo do recipiente de 3 litros para o recipiente de 5 litros.

Para compreender o algoritmo, sejam A e B os recipientes de 3 litros e de 5 litros, respectivamente, e indiquemos por (X, n) o fato de o recipiente X conter n litros de água. No início

[1] A solução desta questão foi necessária num filme da série Duro de Matar para o protagonista desativar uma bomba.

temos (A, 0) e (B, 0) e, após a execução de cada instrução, teremos:

1. (A, 3), (B, 0).
2. (A, 0), (B, 3).
3. (A, 3), (B, 3).
4. (A, 1), (B, 5).
5. (A, 1), (B, 0).
6. (A, 0), (B, 1).
7. (A, 3), (B, 1).
8. (A, 0), (B, 4).

Outras questões que podem ser levantadas são: há outras soluções? Existe alguma solução que realize a mesma tarefa com uma quantidade menor de instruções? Para responder a estas questões talvez seja interessante lembrar que 4 = 5 – 1. Significa que, se conseguirmos tirar 1 litro do recipiente de 5 litros quando ele estiver cheio, resolveremos a questão. Para conseguir isto, basta que o recipiente de 3 litros contenha 2 litros. E para se obter 2 litros? Aí basta ver que 2 = 5 – 3. Podemos então resolver a questão com o seguinte algoritmo, constituído de apenas seis instruções:

1. Encha o recipiente de 5 litros.
2. Com o conteúdo do recipiente de 5 litros, encha o de 3 litros.
3. Esvazie o recipiente de 3 litros.
4. Transfira o conteúdo do recipiente de 5 litros para o recipiente de 3 litros.
5. Encha o recipiente de 5 litros.
6. Com o conteúdo do recipiente de 5 litros, complete o recipiente de 3 litros.

Após a execução de cada uma das instruções teremos:

1. (A, 0), (B, 5).
2. (A, 3), (B, 2).
3. (A, 0), (B, 2).
4. (A, 2), (B, 0).
5. (A, 2), (B, 5).
6. (A, 3), (B, 4).

Uma outra técnica bastante utilizada é se tentar raciocinar a partir de uma solução conhecida de uma outra questão. Para compreender isto considere as duas seguintes questões: imagine uma relação de *n* números, os quais podem ser referenciados por a_i com i = 1, 2, ..., n e queiramos somá-los com a restrição de que só sabemos efetuar somas de duas parcelas. Para resolver esta questão, podemos pensar em casos particulares: se n = 2, basta somar os dois números; se n = 3, basta somar os dois primeiros e somar esta soma com o terceiro. Naturalmente este raciocínio pode ser reproduzido para n > 3. A questão é que a soma dos dois primeiros deve estar "guardada" para que se possa somá-la com o terceiro, obtendo-se a soma dos três primeiros; esta soma deve ser "guardada" para que seja somada com o quarto e assim sucessivamente. Para isto podemos estabelecer uma *referência* à soma "atual", a qual será alterada quando a soma com o elemento seguinte for efetuada. Até para somar os dois primeiros, pode-se pensar em somar "a soma do primeiro" com o segundo.

Temos então o seguinte algoritmo:

1. Faça i = 1.
2. Faça Soma = a_1.
3. Repita n – 1 vezes as instruções 3.1 e 3.2.
   3.1. Substitua i por i + 1.
   3.2. Substitua Soma por Soma + a_i.

Por exemplo: se n = 5 e a_1 = 8, a_2 = 4, a_3 = 9, a_4 = 13 e a_5 = 7, a execução do algoritmo resultaria nas seguintes ações:

1. i = 1.
2. Soma = 8.
3.1.1. i = 2.
3.2.1. Soma = 8 + 4 = 12
3.1.2. i = 3.
3.2.2. Soma = 12 + 9 = 21.
3.1.3. i = 4.
3.2.3. Soma = 21 + 13 = 34.
3.1.4. i = 5.
3.2.4. Soma = 34 + 7 = 41.

Naturalmente, na execução acima estamos indicando por 3.1.x e 3.2.x a execução de ordem x das instruções 3.1 e 3.2.

Como veremos ao longo do livro, este algoritmo é bastante utilizado em programação, sendo que é mais comum que até o primeiro termo da relação seja "somado" dentro da repetição. Neste caso, para que o primeiro seja somado, é necessário que *Soma* seja *inicializado* com 0 (zero), ficando assim o algoritmo:

1. Faça i = 0.
2. Faça Soma = 0.
3. Repita n vezes as instruções 3.1 e 3.2.
   3.1. Substitua i por i + 1.
   3.2. Substitua Soma por Soma + a_i.

Conhecendo este algoritmo, é fácil então resolver a questão de se calcular o produto de *n* números nas mesmas condições, e aí vemos como utilizar uma solução conhecida para resolver um problema. Deve-se inicializar uma referência *Produto* com 1 e, numa repetição, multiplicar os números como foi feito no caso da soma:

1. Faça i = 0.
2. Faça Produto = 1.
3. Repita n vezes as instruções 3.1 e 3.2.
   3.1. Substitua i por i + 1.
   3.2. Substitua Produto por Produto x a_i.

1.6 Processador de um algoritmo

Obviamente, um algoritmo deve ser executado por algum agente. Este agente pode ser uma pessoa munida de certos equipamentos e utensílios ou por máquinas projetadas para executar automaticamente algumas instruções básicas. O algoritmo para a travessia do senhor gordo com as galinhas, sua raposa e seu saco de milho seria executado pelo tal senhor, que estava para tal munido do barco e de remos. O algoritmo para obtenção de quatro litros de água a partir de recipientes de conteúdos cinco litros e três litros poderia ser executado por uma pessoa que dispusesse dos dois recipientes e de água em abundância. Neste último caso, quem sabe, a pessoa poderia ser substituída por um robô.

O agente que executa um algoritmo é chamado *processador* e para que o algoritmo seja executado é necessário que o processador seja capaz de executar cada uma das suas instruções. Se o senhor gordo não souber remar ele não será capaz de atravessar o rio. Uma pessoa que não seja capaz de esvaziar um recipiente que pese cinco quilos não será capaz de executar o algoritmo dos quatro litros de água.

Alguns autores de livros com objetivos idênticos a este - facilitar a aprendizagem da programação de computadores - iniciam seus textos discorrendo exclusivamente sobre *resolução de problemas*, encarando o processador como uma "caixa preta" que recebe as instruções formuladas pelo algoritmo e fornece a solução do problema, não levando em conta o processador quando da

formulação do tal algoritmo. Entendemos que esta não é a melhor abordagem, visto que o conhecimento do que o processador pode executar pode ser definidor na elaboração do algoritmo. Por exemplo: imagine que queiramos elaborar um algoritmo para extrair o algarismo da casa das unidades de um inteiro dado (apresentaremos posteriormente uma questão bastante prática cuja solução depende deste algoritmo). Evidentemente, o algoritmo para resolver esta "grande" questão depende do processador que vai executá-lo. Se o processador for um ser humano que saiba o que é número inteiro, algarismo e casa das unidades, o algoritmo teria uma única instrução:

1. Forneça o algarismo das unidades do inteiro dado.

Porém, se o processador for um ser humano que saiba o que é número inteiro e algarismo, mas não saiba o que é casa das unidades, o algoritmo não poderia ser mais esse. Neste caso, para resolver a questão, o processador deveria conhecer mais alguma coisa, como, por exemplo, ter a noção de "mais à direita", ficando o algoritmo agora como:

1. Forneça o algarismo "mais à direita" do número dado.

E se o processador é uma máquina e não sabe o que é algarismo, casa das unidades, "mais à direita", etc.? Nesta hipótese, quem está elaborando o algoritmo deveria conhecer que instruções o processador é capaz de executar para poder escrever o seu algoritmo. Por exemplo, se a máquina é capaz de determinar o resto de uma divisão inteira, o algoritmo poderia ser:

1. Chame de n o inteiro dado;
2. Calcule o resto da divisão de n por 10;
3. Forneça este resto como o algarismo pedido.

Algumas das questões anteriores são importantes para se desenvolver o raciocínio, mas não é este tipo de questão que se pretende discutir ao longo deste livro. Estamos interessados em algoritmos para:

1. Resolver problemas matemáticos, como algoritmos para determinar a média aritmética de vários números dados, determinar as raízes de uma equação do segundo grau, encontrar o máximo divisor comum de dois números dados, totalizar as colunas de uma tabela, etc.
2. Resolver questões genéricas, como algoritmos para colocar em ordem alfabética uma relação de nomes de pessoas, atualizar o saldo de uma conta bancária na qual se fez um depósito, corrigir provas de um teste de múltipla escolha, cadastrar um novo usuário de uma locadora, etc..

Na linguagem natural, o algoritmo para o cálculo da média pode ser escrito de forma muito simples:

1. Determine a quantidade de números;
2. Some os números dados;
3. Divida esta soma pela quantidade de números.

Qualquer pessoa que saiba contar, somar e dividir números é capaz de executar este algoritmo dispondo apenas de lápis e papel. A questão que se põe é: e se a relação contiver 13.426 números? A tal pessoa é capaz de executar, porém, quanto tempo levará para fazê-lo?

Um outro aspecto a ser observado é que nem sempre a linguagem natural é eficiente para que as instruções sejam escritas. Nessa linguagem o algoritmo para determinação das raízes de uma equação do segundo grau teria uma instrução difícil de escrever e difícil de compreender como:

n. Subtraia do quadrado do segundo coeficiente o produto do número quatro pelo produto dos dois outros coeficientes.

Isto pode ser parcialmente resolvido utilizando-se uma linguagem próxima da linguagem matemática que já foi utilizada em exemplos da seção anterior. No caso da equação do segundo grau teríamos o seguinte algoritmo, que nos é ensinado nas últimas séries do ensino fundamental:

1. Chame de a, b e c os coeficientes da equação.

2. Calcule d = b² - 4ac.
3. Se d < 0 forneça como resposta a mensagem: A equação não possui raízes reais.
4. Se d ≥ 0
   4.1 Calcule x_1 = (-b + raiz(d))/2a e x_2 = (-b - raiz(d))/2a.
   4.2 Forneça x_1 e x_2 como raízes da equação.

De maneira mais ou menos evidente, *raiz(d)* está representando a raiz quadrada de *d* e a execução deste algoritmo requer que o processador seja capaz de determinar valores de expressões aritméticas, calcular raízes quadradas, efetuar comparações e que conheça a linguagem matemática.

Algoritmos para problemas genéricos são mais complicados e as linguagens utilizadas anteriormente não são adequadas (para o caso da ordenação de uma relação de nomes, foram desenvolvidos vários algoritmos e teremos oportunidade de discutir alguns deles ao longo deste livro).

1.7 Exemplos de algoritmos matemáticos

Para uma primeira discussão em termos de aprendizagem de desenvolvimento de algoritmos e utilizando a linguagem usada no exemplo da equação do segundo grau, apresentamos a seguir alguns exemplos de algoritmos que objetivam a solução de questões da matemática. Para eles supomos que o processador seja capaz de efetuar somas, subtrações e divisões decimais, de realizar comparações, de repetir a execução de um conjunto de instruções um número determinado de vezes ou enquanto uma condição seja atendida.

1. No exemplo do algoritmo para obtenção do algarismo da casa das unidades de um inteiro dado supomos que o processador seria capaz de calcular o resto de uma divisão inteira. Observando que não está suposto que o nosso processador seja capaz de determinar restos de divisões inteiras, vamos discutir um algoritmo para a determinação do quociente e do resto da divisão de dois inteiros positivos dados. Por exemplo: se o dividendo for 30 e o divisor for 7, o algoritmo deve fornecer os valores 4 para o quociente e 2 para o resto. Fomos ensinados que, para determinar o quociente, deveríamos, por tentativa, encontrar o número que multiplicado pelo divisor resultasse no maior número menor que o dividendo. No exemplo numérico citado, poderíamos tentar o 5 e teríamos 5x7 = 35 que é maior que 30; tentaríamos o 3 obtendo 3x7 = 21 que talvez seja pequeno demais em relação ao 30; aí tentaríamos o 4 obtendo 4x7 = 28, encontrando então o quociente 4. Um algoritmo para solucionar esta questão poderia ser:

1. Chame de D1 e D2 o dividendo e o divisor dados.
2. Faça I = 1.
3. repita 3.1 até IxD2 > D1.
   3.1. Substitua I por I + 1.
4. Calcule Q = I – 1.
5. Calcule R = D1 - QxD2.
6. Forneça R para o resto e Q para o quociente pedidos.

No exemplo numérico proposto, teríamos a seguinte tabela com os valores obtidos durante a execução do algoritmo:

D1	D2	I	QxI	Q	R
30	7				
		1	7		
		2	14		
		3	21		
		4	28		
		5	35		
				4	2

2. O algoritmo abaixo determina o menor divisor maior que 1 de um inteiro dado. A idéia é verificar se d = 2 é divisor e, não sendo, verificar se 3 ou 4 ou 5, etc, é divisor. A procura por um divisor vai até que um divisor seja encontrado. Naturalmente, utilizando o algoritmo anterior, o nosso processador agora sabe determinar o resto da divisão inteira de um inteiro x por outro inteiro y não nulo. Isto será indicado por Resto(x, y). Para encontrar um divisor de n basta encontrar um inteiro d tal que Resto(n, d) = 0.

1. Chame de N o inteiro dado.
2. Faça D = 2.
3. Repita 3.1 enquanto Resto(N, D) $\neq$ 0
   3.1 Substitua D por D + 1
4. Forneça D para o divisor procurado.

3. Como se depreende facilmente da sua denominação, o *máximo divisor comum* (*mdc*) de dois números dados é o maior número que os divide. Antes o *mdc* só era utilizado para simplificações de frações ordinárias; atualmente ele é utilizado na determinação de *chaves públicas* para sistemas de criptografia RSA [Evaristo, J, 2010]. Por exemplo, mdc(64, 56) = 8. De maneira óbvia, o algoritmo abaixo determina o *mdc* de dois números dados:

1. Chame de x e de y os números.
2. Determine D(x), o conjunto dos divisores de x.
3. Determine D(y), o conjunto dos divisores de y.
4. Determine I, a interseção de D(x) e D(y).
5. Determine M, o maior elemento do conjunto I.
6. Forneça M como o mdc procurado.

O cálculo de mdc(120, 84) com este algoritmo seria:

1. x = 120, y = 84.
2. D(120) = {1, 2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 24, 30, 40, 60, 120}.
3. D(84) = {1, 2, 3, 4, 6, 7, 12, 14, 21, 28, 42, 84}.
4. I = {1, 2, 3, 4, 6, 12}.
5. M = 12.

Observe que o algoritmo anterior determina o menor divisor de um inteiro não determinando todos os divisores, como necessário neste exemplo. Observe também que estamos supondo que o nosso processador é capaz de determinar a interseção de dois conjuntos.

A Matemática fornece uma outra forma de se calcular o *mdc* de dois inteiros: determina-se a *decomposição em fatores primos* dos dois inteiros e o *mdc* é o produto dos fatores primos comuns as duas decomposições com as menores multiplicidades.

Para o exemplo dado acima teríamos:

120	2
60	2
30	2
15	3
5	5
1	1

84	2
42	2
21	3
7	7
1	

o que nos dá $120 = 2^3 x 3 x 5$ e $84 = 2^2 x 3 x 7$ e, portanto, $mdc(120, 84) = 2^2 x 3 = 12$.

Vale observar que escrever este algoritmo na linguagem informal que estamos utilizando é bastante complicado.

Na há dúvida que o primeiro algoritmo para o cálculo do *mdc* apresentado é de compreensão bastante simples. Porém, comentaremos posteriormente que ele é computacionalmente bastante ineficiente no sentido de que sua execução pode, dependendo dos valores de *x* e *y*, demandar um tempo acima do razoável.

Por incrível que possa parecer, um dos algoritmos mais eficientes para o cálculo do máximo divisor comum de dois números foi desenvolvido pelo matemático grego Euclides duzentos anos Antes de Cristo. O *algoritmo de Euclides* nos é apresentado nas séries intermediárias do ensino fundamental através de um esquema como o diagrama do exemplo abaixo, cujo objetivo é determinar (de novo!) o máximo divisor comum de 120 e 84.

120	84	36	12	0
	1	2	3	

O esquema funciona da seguinte forma: divide-se 120 por 84 obtendo-se resto 36; a partir daí, repetem-se divisões até que o resto seja zero, sendo o dividendo da divisão atual o divisor da divisão anterior e o divisor da divisão atual o resto da divisão anterior. O último divisor é o máximo divisor procurado. Como se pode ver, estas instruções escritas desta forma não são nada compreensíveis, o que faz com elas sejam transmitidas oralmente nas salas do ensino fundamental. No capítulo 4 (quatro), teremos a oportunidade de discutir este algoritmo com detalhes e veremos que ele é um algoritmo bastante interessante no desenvolvimento da lógica de programação.

4. Discutiremos agora o algoritmo para o cálculo da média de uma relação contendo um número grande (digamos, 10 000) de números dados. No caso da equação do segundo grau, eram três os dados de entrada e, portanto, os chamamos de *a*, *b*, e *c*. Mas agora são 10 000 os dados de entrada! Uma solução possível é receber os números um a um, somando-os antes de receber o seguinte, conforme vimos na seção 1.5.

1. Chame de A o primeiro número dado.
2. Faça S = A.
3. Repita 9 999 vezes as instruções 3.1 e 3.2.
   3.1 Chame de A o próximo número dado.
   3.2 Substitua o valor de S por S + A.
4. Calcule M = S/10 000.
5. Forneça M para o valor da média.

Por exemplo, se a relação de números fosse {5, 3, 8, 11, ...} até a quarta execução de 3.1 e 3.2 teríamos a seguinte tabela:

A	S	M
5	5	
3	8	
8	16	
11	27	

Está fácil perceber que após 9.999ª execução das instruções 3.1 e 3.2 a variável S conterá a soma de todos os números da relação, o que justifica a instrução 4.

5. Um outro exemplo que justifica plenamente a necessidade do conhecimento do que o processador é capaz de executar é a determinação do maior número de uma relação de números. Se o processador for um aluno do ensino médio e a relação contiver poucos números, uma simples olhada na relação permitirá se identificar o maior número. Mas, e se o processador for um aluno das classes iniciais do ensino fundamental? E se a relação contiver 10 000 números? E se os números estiverem escritos em forma de fração ordinária?

Uma solução possível é supor que o maior número é o primeiro da relação e comparar este suposto maior com os demais números, alterando-o quando for encontrado um número na relação maior do que aquele que até aquele momento era o maior.

1. Chame de A o primeiro número dado.
2. Faça M = A.
3. Repita 9 999 vezes as instruções 3.1 e 3.2.
   3.1 Chame de A o próximo número dado.

3.2 Se A > M substitua o valor de M por A.

4. Forneça M para o valor do maior número.

Para exemplificar, suponha que a entrada fosse o conjunto {5, 3, 8, 11, 10...}. Até a quinta execução das instruções 3.1 e 3.2 teríamos a seguinte tabela:

A	M
5	5
3	
8	8
11	11
10	

1.8 Linguagens de alto nível

Computadores digitais foram concebidos para executarem instruções escritas em linguagem de máquina. Isto significa que um computador é capaz de executar um algoritmo escrito nesta linguagem. Um algoritmo escrito em linguagem de máquina é normalmente chamado de *programa objeto*. Nos primórdios da computação, os algoritmos que se pretendiam que fossem executados por um computador eram escritos em linguagem de máquina, o que tornava a tarefa de desenvolvimento de algoritmos muito trabalhosa, devido ao fato de que era necessário que se conhecesse qual seqüência de bits correspondia à instrução pretendida. Naturalmente, esta dificuldade acontecia pelo fato de que o ser humano não está habituado a uma linguagem com apenas dois símbolos básicos. Um grande avanço ocorreu na computação quando se conseguiu desenvolver programas que traduzissem instruções escritas originariamente numa linguagem dos seres humanos para a linguagem de máquina. O surgimento de programas para esta finalidade permitiu o desenvolvimento de algoritmos em linguagens que utilizam caracteres, palavras e expressões de um idioma, ou seja, uma linguagem cujos símbolos básicos e cujas palavras estão no cotidiano do ser humano. Uma linguagem com esta característica é chamada *linguagem de alto nível*, onde *alto nível* aí não se refere à qualidade e sim ao fato de que ela está mais próxima da linguagem do ser humano do que da linguagem da máquina (quando alguma coisa está mais próxima da máquina do que do ser humano dizemos que ela é *de baixo nível*). Como exemplo de linguagens de alto nível temos *Pascal*, *C*, *Delphi*, *Visual Basic, Java* e *C*$^{++}$. Um algoritmo escrito numa linguagem de alto nível é chamado *programa fonte* ou simplesmente *programa*

Como foi dito acima, um *programa fonte* deve ser traduzido para a linguagem de máquina. Há dois tipos de programas que fazem isto: os *interpretadores* que traduzem os comandos para a linguagem de máquina um a um e os *compiladores* que traduzem todo o programa para a linguagem de máquina. Um compilador ao receber como entrada um programa fonte fornece como saída um programa escrito em linguagem de máquina, chamado *programa objeto*. A *compilação* do programa, portanto, gera um programa que pode então ser executado. É comum nos referirmos à execução do programa fonte quando se está executando o programa objeto.

Já um interpretador traduz para a linguagem de máquina os comandos do programa um a um, executando-os em seguida. Assim a *interpretação* de um programa não gera um programa objeto.

1.9 Sintaxe e semântica de uma instrução

O que é sintaxe

Dissemos que um programa escrito em linguagem de alto nível é traduzido para a linguagem de máquina por um compilador ou cada instrução é traduzida por um interpretador. É natural se admitir que, para que o compilador consiga traduzir uma instrução escrita com caracteres de algum idioma para instruções escritas como seqüências de zeros e uns, é necessário que cada instrução seja escrita de acordo com regras preestabelecidas. Estas regras são chamadas *sintaxe* da instrução e

quando não são obedecidas dizemos que existe *erro de sintaxe*.

Se o programa fonte contém algum erro de sintaxe, o compilador não o traduz para a linguagem de máquina (isto é, o compilador não *compila* o programa) e indica qual o tipo de erro cometido e a instrução onde este erro aconteceu. Se o programa fonte for interpretado, ele é executado até a instrução que contém o erro, quando então é interrompida a sua execução e o tal erro é indicado.

O que é semântica

Naturalmente, cada instrução tem uma finalidade específica. Ou seja, a execução de um instrução resulta na realização de alguma ação, digamos *parcial*, e é a seqüência das ações parciais que redunda na realização da tarefa para a qual o programa foi escrito. A ação resultante da execução de uma instrução é chamada *semântica* da instrução. Infelizmente, um programa pode não conter erros de sintaxe (e, portanto, pode ser executado), mas a sua execução não fornecer como saída o resultado esperado para alguma entrada. Neste caso, dizemos que o programa contém *erros de lógica* que, ao contrário dos erros de sintaxe que são detectados pelo compilador ou pelo interpretador, são, às vezes, de difícil detecção.

No nosso entendimento, para aprender a programar numa determinada linguagem é necessário que se aprenda as instruções daquela linguagem (para que se conheça o que o processador é capaz de fazer), a sintaxe de cada um destes instruções e as suas semânticas. Aliado a isto, deve-se ter um bom desenvolvimento de *lógica programação* para que se escolha as instruções necessárias e a seqüência segundo a qual estas instruções devem ser escritas, para que o programa, ao ser executado, execute a tarefa pretendida. Felizmente ou infelizmente, para cada tarefa que se pretende não existe apenas uma seqüência de instruções que a realize. Ou seja, dado um problema não existe apenas um programa que o resolva. Devemos procurar o *melhor programa*, entendendo-se como *melhor programa* um programa que tenha boa *legibilidade*, cuja execução demande o menor tempo possível e que necessite, para sua execução, a utilização mínima da memória.

Existe um conjunto de instruções que é comum a todas as linguagens de alto nível e cujas semânticas permitem executar a maioria das tarefas. A aprendizagem das semânticas destas instruções e das suas sintaxes em alguma linguagem de programação (aliado ao desenvolvimento da *lógica de programação*) permite que se aprenda com facilidade outra linguagem do mesmo paradigma.

1.10 Sistemas de computação

Como foi dito anteriormente, a *cpu* de um computador é capaz de executar instruções (escritas em linguagem de máquina, permitam a repetição). Ou seja, um computador é capaz de executar programas e só para isto é que ele serve. Se um computador não estiver executando um programa ele para nada está servindo. Como foram concebidos os computadores atuais, um programa para ser executado deve estar armazenado na sua *memória*. O armazenamento dos programas (e todo o gerenciamento das interações entre as diversas unidades do computador) é feito por um programa chamado *sistema operacional*. Um dos primeiros sistemas operacionais para gerenciamento de microcomputadores foi o *DOS (Disk Operating System)*. Quando um computador é ligado, de imediato o sistema operacional é armazenado na memória e só a partir daí o computador está apto a executar outros programas. Estes programas podem ser um *game*, que transforma o "computador" num poderoso veículo de entretenimento; podem ser um *processador de texto*, que transforma o "computador" num poderoso veículo de edição de textos; podem ser uma *planilha eletrônica*, que transforma o "computador" num poderoso veículo para manipulação de tabelas numéricas, podem ser programas para gerenciar, por exemplo, o dia a dia comercial de uma farmácia e podem ser ambientes que permitam o desenvolvimento de *games* ou de programas para gerenciar o dia a dia comercial de uma farmácia. Talvez com exceção de um *game*, os programas citados acima são, na verdade, conjuntos de programas que podem ser executados de forma integrada. Um conjunto de

programas que podem ser executados de forma integrada é chamado *software*. Por seu turno, as unidades do computador, associadas a outros equipamentos chamados *periféricos*, como uma impressora, constituem o *hardware*. O que nos é útil é um conjunto *software* + *hardware*. Um conjunto deste tipo é chamado de um *sistema de computação*. De agora em diante, os nossos processadores serão *sistemas de computação*. Isto é, queremos escrever programas que sejam executado por um *sistema de computação*.

Como foi dito acima, o desenvolvimento de um programa que gerencie o dia a dia comercial de uma farmácia requer um compilador (ou um interpretador) que o traduza para a linguagem de máquina. Antigamente, as empresas que desenvolviam compiladores desenvolviam apenas estes programas, de tal sorte que o programador necessitava utilizar um processador de texto à parte para edição do programa fonte. Atualmente, os compiladores são integrados num sistema de computação que contém, entre outros:

1. *Processador de texto*, para a digitação dos programas fontes;
2. *Depurador*, que permite que o programa seja executado comando a comando, o que facilita a descoberta de erros de lógica;
3. *Help*, que descreve as sintaxes e as semânticas de todas as instruções da linguagem;
4. *Linker*, que permite que um programa utilize outros programas.

Rigorosamente falando, um sistema constituído de um compilador e os *softwares* listados acima deveria ser chamado de *ambiente de programação*; é mais comum, entretanto, chamá-lo simplesmente de *compilador*.

O ambiente de programação utilizado neste livro, denominado *ILA*, contém um *interpretador,* um *depurador* e um *help* e foi desenvolvido pelo Professor Sérgio Crespo, da Universidade do Vale do rio dos Sinos, UNISINOS. O interpretador interpreta programas desenvolvidos numa *linguagem algorítmica* na qual os comandos são escritos em português e possuem sintaxes bastante simples. Como o ILA não tem integrado um editor de texto, os programas devem editados num processador de texto que grave os textos editados em código ASCII, como, por exemplo, o *bloco de nota*s do Windows.

1.11 Exercícios propostos

1. Três índios precisam afastar de suas terras três brancos invasores. Para tal e dispondo de um barco cuja capacidade é de apenas duas pessoas, precisam atravessar um rio. Por questões de segurança, os índios não querem ficar em minoria, em nenhum momento e em nenhuma das margens. Escreva um algoritmo que oriente os índios para realizarem a travessia nas condições fixadas. (Cabe observar que, usualmente, este exercício é enunciado envolvendo três jesuítas e três canibais. A alteração feita é uma modesta contribuição para o resgate da verdadeira história dos índios).

2. O jogo conhecido como *Torre de Hanói* consiste de três torres chamadas *origem*, *destino* e *auxiliar* e um conjunto de n discos de diâmetros diferentes, colocados na torre *origem* na ordem decrescente dos seus diâmetros. O objetivo do jogo é, movendo um único disco de cada vez e não podendo colocar um disco sobre outro de diâmetro menor, transportar todos os discos para torre *destino*, podendo usar a torre *auxiliar* como passagem intermediária dos discos. Escreva algoritmos para este jogo nos casos n = 2 e n = 3.

3. Imagine que se disponha de três esferas numeradas 1, 2 e 3 iguais na forma, duas delas com pesos iguais e diferentes do peso da outra. Escreva um algoritmo que, com duas pesagens numa balança de dois pratos, determine a esfera de peso diferente e a relação entre seu peso e o peso das esferas de pesos iguais.

4. A *média geométrica* de *n* números positivos é a raiz n-ésima do produto destes números. Supondo que o processador é capaz de calcular raízes n-ésimas, escreva um algoritmo para determinar a média geométrica de *n* números dados.

5. Sabendo que o dia 01/01/1900 foi uma segunda-feira, escreva um algoritmo que determine

o dia da semana correspondente a uma data, posterior a 01/01/1900, dada. Por exemplo, se a data dada for 23/01/1900, o algoritmo deve fornecer como resposta terça-feira.

6. O show de uma banda de rock, que será realizado na margem de um rio, deve começar exatamente às 21 h. Atrasados, às 20 h 43 min, os quatro integrantes da banda estão na outra margem do rio e necessitam, para chegar ao palco, atravessar uma ponte. Há somente uma lanterna e só podem passar uma ou duas pessoas juntas pela ponte, e sempre com a lanterna. Cada integrante possui um tempo diferente para atravessar a ponte: o vocal leva 10 minutos, o guitarrista 5 minutos, o baixista 2 minutos e o baterista 1 minuto. Evidentemente, quando dois atravessam juntos, o tempo necessário é o do mais lento. Escreva um algoritmo que permita que a banda atravesse a ponte de modo que o show comece na hora marcada.

7. Resolva a questão 3 para o caso de oito esferas.

Observação

Para obter o caderno das soluções dos exercícios propostos basta encaminhar e-mail para jaime@ccen.ufal com assunto RESPOSTAS LIVRO ILA e contendo NOME, CATEGORIA (docente, discente, autodidata), CIDADE/ESATADO e, se for o caso, INSTITUIÇÃO/CURSO.

2. Introdução à Linguagem Algorítmica

2.1 Variáveis simples

Nos exemplos do capítulo anterior, utilizamos instruções do tipo *chame de a, b e c os coeficientes*. Com isto queríamos admitir que processador fosse capaz de associar cadeias de caracteres a valores numéricos. Como o nosso processador, de agora em diante, será um sistema de computação, as coisas têm que ser mais rigorosas.

Na seção 1.1 foi dito que uma das unidades básicas de um computador é a *memória*, cuja finalidade é armazenar dados e informações que serão manipulados pela *unidade central de processamento*. Na seção anterior também foi dito que os programas para serem executados devem estar armazenados na memória. Ou seja, a memória armazena programas que serão executados e dados que estes programas vão manipular. Estes dados podem ser *dados de entrada* ou dados gerados pela execução do programa. Para que a memória possa armazenar dados e programas, ela é dividida em partes, chamadas *posições de memória*, podendo o sistema operacional acessar cada uma destas posições para armazenar tais dados. O acesso do sistema operacional às posições de memória é possível porque a cada uma delas é associada uma sequência de bit's, chamada *endereço* da posição de memória.

Em programação de computadores, uma *variável simples* (ou simplesmente *variável*) é uma posição de memória à qual estão associados um *identificador* e um *tipo de dado*. O *identificador* é uma sequência de letras, dígitos e caractere para sublinhamento (sequência que deve começar com uma letra) e será utilizada no programa para se fazer referência àquela variável. Como um programa deve ser legível por outros programadores (e pelo próprio programador, em leituras posteriores), é uma boa prática se escolher um identificador de uma variável que tenha alguma relação com a sua finalidade. Se uma variável deve armazenar uma soma, um identificador muito bom para ela será *Soma*. Se uma variável vai receber números, ela poderia ser identificada por *Num* ou por *Numero*. Naturalmente, não se pode utilizar um mesmo identificador para variáveis distintas, devendo a distinção entre os identificadores ocorrer até o vigésimo caractere. Como veremos a seguir, a linguagem algorítmica do ILA fixa alguns identificadores para as sintaxes de seus comandos. Estes identificadores não podem ser utilizados nos programas, sendo conhecidos por *palavras reservadas*.

O *tipo de dado* associado a uma variável é o conjunto de elementos que podem ser nela armazenados. O ILA aceita três tipos de dados: *numerico*, *logico* e *caracter* (*numerico* e *logico* são escritos assim mesmo, sem acentos). Uma variável do tipo *numerico* (como esta denominação indica) está apta a armazenar números; uma variável do tipo *logico* está apta a armazenar os valores *falso* e *verdadeiro* e uma variável do tipo *caracter* está apta a armazenar *cadeias de caracteres* (palavras, sequências de algarismos, sequências de letras e algarismos, etc.).

Para que o sistema de computação possa reservar as posições de memória que serão utilizadas pelo programa, associar identificadores aos endereços destas posições de memória e definir a quantidade de bytes de cada posição de memória, de acordo com o tipo de dado pretendido, um programa escrito em ILA começa com a *declaração de variáveis*, que deve ser feita com a seguinte sintaxe:

```
Variaveis
    Tipo de dado  Lista de identificadores
```

com *Variaveis* escrito assim mesmo, sem acento.

Por exemplo, um programa para determinar a média de uma relação de números dados pode ter a seguinte definição:

```
Variaveis
    Numerico Num, Soma, Quant, Media
```

A ideia é que *Num* sirva para armazenar os números (um de cada vez); *Soma* para armazenar a soma dos números; *Quant* para armazenar a quantidade de números e *Media* para armazenar (aí é

evidente) a média procurada.

Considerando as observações acima a respeito dos identificadores e o fato de que o interpretador ILA não faz distinção entre letras maiúsculas e minúsculas, a definição abaixo provocaria erro

```
Variaveis
    Numerico Maior_triangulo_equilatero
    Caracter maior_triangulo_equiangulo
```

pois os vinte primeiros caracteres são (para o interpretador) iguais.

Nas seções 2.6 e 2.8 veremos com que comandos o ILA armazena dados de entrada e dados gerados pela execução do programa em variáveis. Um valor armazenado em uma variável é comumente referido como sendo o *conteúdo* da variável ou o *valor* da variável. Também é comum se referir ao identificador da variável como sendo a própria variável.

2.2 Expressões aritméticas

O ILA, como era de se esperar, é capaz de avaliar expressões aritméticas que envolvam as operações binárias de multiplicação, divisão, soma e subtração e a operação unária de *troca de sinal*. Para isto são usados os *operadores aritméticos* *****, para a multiplicação, /, para a divisão, +, para a soma, e **-**, para a subtração e troca de sinal.

Uma expressão que envolva diversas operações é avaliada de acordo com as regras de prioridade da matemática: em primeiro lugar é realizada a operação troca de sinal, em seguida são realizadas as multiplicações e divisões e, finalmente, as somas e subtrações. Por exemplo, a expressão 8 + 2*-3 é avaliada como sendo igual a 8 + (-6) = 2. Naturalmente, a prioridade pode ser alterada com a utilização de parêntesis: a expressão (8 + 2)*-3 resulta em 10*(-3) = -30. Embora, o sistema não exija, vamos sempre utilizar parêntesis para separar o operador unário para troca de sinal de algum operador binário. Assim, 8 + 2*-3 será indicada por 8 + 2*(-3). Uma expressão não parentesiada contendo operadores de mesma prioridade é avaliada da esquerda para direita. Por exemplo, 10/2*3 é igual a (10/2)*3 = 5*3 = 15.

O sistema também é capaz de calcular potências através do operador ^. Assim, o sistema avalia a expressão 2^10 como sendo 1 024. Já 3^-2 é igual a $\frac{1}{3^2}=\frac{1}{9}=0,11$ (o sistema trabalha com duas casas decimais). Preferiremos também escrever 3^(-2) no lugar de 3^-2. A potenciação tem prioridade maior que a multiplicação e que a soma. Assim, 2*3^4 = 2*(3^4) = 2*81 = 162.

Operandos podem ser conteúdos de variáveis. Neste caso, o operando é indicado pelo *identificador* da variável (é para isto que serve o identificador, para se fazer referência aos valores que na variável estão armazenados).

Através de uma *função pré-definida* (veremos o que é isto posteriormente), o ILA também é capaz de calcular restos de divisões inteiras entre dois números inteiros positivos. Isto é feito por meio da sintaxe

```
Resto(Expressão1, Expressão2)
```

onde Expressão1 e Expressão2 são expressões numéricas que resultam num valor inteiro.

Por exemplo, o ILA avalia a expressão 2^3 - Resto(6*5, 8/2) através de

2^3 - Resto(6*5, 8/2) = 8 - Resto(30, 4) = 8 - 2 = 6.

Naturalmente, Resto(7, 12) resulta no valor 7. Vale observar que se o valor de Expressão 1 ou de Expressão 2 são fracionários, Resto(Expressão 1, Expressão 2) é "calculado", mas com erros incontroláveis pelo programador.

2.3 Relações

O sistema ILA realiza *comparações* entre valores numéricos, realizadas no sentido usual da Matemática, através dos *operadores relacionais* **>** (*maior do que*), **>=** (*maior do que ou igual a*), **<** (*menor do que*), **<=** (*menor do que ou igual a*), **=** (*igual*) e **<>** (*diferente*). Uma expressão envolvendo operadores relacionais é chamada *relação* e o resultado da avaliação de uma relação é um dos resultados *falso* ou *verdadeiro*, obtido, como já foi dito, como na Matemática. Assim, 3 > 5 resulta no valor *falso*, enquanto que 7 <= 7 resulta no valor *verdadeiro*. Sendo um valor *falso* ou *verdadeiro*, o resultado da avaliação de uma relação pode ser armazenado numa variável do tipo *Logico*.

Os operandos de uma relação podem ser expressões aritméticas. Neste caso, a avaliação das expressões aritméticas tem prioridade em relação a avaliação das relações. Por exemplo, a relação 3^2 - 5 < 2*3 - 4 resulta no valor *falso*, pois 3^2 - 5 = 4 e 2*3 - 4 = 2.

2.4 Expressões lógicas

O ILA também avalia *expressões lógicas* obtidas através da aplicação dos *operadores lógicos binários e* e *ou* a duas relações ou da aplicação do *operador lógico unário nao* (assim mesmo sem acento) a uma relação. Se R_1 e R_2 são duas relações, a avaliação da aplicação dos operadores lógicos binários, de acordo com os valores de R_1 e R_2, são dados na tabela abaixo.

R_1	R_2	(R_1) e (R_2)	(R_1) ou (R_2)
verdadeiro	*verdadeiro*	*verdadeiro*	*verdadeiro*
verdadeiro	*falso*	*falso*	*verdadeiro*
falso	*verdadeiro*	*falso*	*verdadeiro*
falso	*falso*	*falso*	*falso*

Ou seja, uma expressão lógica do tipo *(R_1) e (R_2)* só recebe o valor *verdadeiro* se os valores de R_1 e de R_2 forem *verdadeiro*; uma expressão lógica do tipo *(R_1) ou (R_2)* só recebe o valor *falso* se os valores de R_1 e de R_2 forem *falso*.

A aplicação do operador unário *nao* simplesmente inverte o valor original da relação:

R_1	nao (R_1)
VERDADEIRO	*FALSO*
falso	*verdadeiro*

Como indicado acima, as relações operandos de uma expressão lógica devem estar dentro de parênteses.

Considerando que os operadores *e* e *ou* possuem o mesmo grau de prioridade, se uma expressão não parentesiada possuir mais de uma relação, ela será avaliada da esquerda para direita. O operador unário *nao* tem prioridade em relação aos operadores binários. Assim, *nao* (5 > 3) *ou* (5 < 3) tem valor *falso*, pois *nao* (5 > 3) é *falso* e 5 < 3 também é.

2.5 Estrutura/Execução de um programa em ILA

Estamos tentando aprender a escrever programas em ILA. Já vimos que se o programa necessitar manipular variáveis, estas devem ser definidas no início do programa.

Veremos no capítulo 5 que um programa pode conter *funções*. Neste caso, a definição das funções deve seguir logo a definição das variáveis. Após a definição das funções aparece a única parte do programa que é obrigatória. Esta parte é chamada *bloco principal* ou *programa principal* e contêm a sequência de comandos (e *chamadas de funções* – o que será visto no capítulo acima referido) que constitui o programa propriamente dito. Desta forma, um programa escrito em ILA terá a seguinte estrutura.

```
Variaveis
        Tipo de dado Lista de identificadores
        Funcao
        Inicio
                //comandos da função                    .
        Fim
Inicio
        //comandos do programa principal
Fim
```

O ILA não contém um editor de textos e é um sistema que é executado sob a ação do sistema operacional DOS. Desta forma, os programas (editados em um processador de textos qualquer) devem ser salvos no modo *txt* e identificados por **NomeDoArquivo.ila**, onde **NomeDoArquivo** deve ter no máximo oito caracteres e é escolhido pelo programador (como no caso dos identificadores de variáveis, é uma boa prática se escolher um **NomeDeArquivo** que tenha alguma relação com o objetivo do programa). Após a gravação do programa na pasta que contém o ILA, a sua execução pode ser solicitada através do *prompt* do DOS com o comando **ila NomeDoArquivo**.

2.6 Comando de entrada

Vimos anteriormente que a maioria dos programas manipula dados que são fornecidos pelo usuário durante a execução do programa. Estes dados, repetindo, constituem a *entrada do programa* e, naturalmente, devem ser armazenados em variáveis. O comando do ILA que permite que o usuário digite no teclado algum dado durante a execução de um programa e que armazena este dado na memória é chamado *comando de entrada* e deve ser escrito com a seguinte sintaxe:

```
Ler Identificador da variável
```

Quando da execução de um comando de entrada, todo o processamento é interrompido e o sistema fica aguardando que o usuário digite um valor. Quando um valor é digitado o sistema verifica se este é um valor do tipo de dado da variável e, em caso positivo, o armazena naquela variável. Por exemplo,

```
Variaveis
        Caracter Frase
Inicio
        Ler Frase
Fim
```

é um programa em ILA, que, simplesmente, armazenará uma cadeia de caracteres digitada no teclado na variável *Frase* (ou seja, para nada serve, pois a cadeia armazenada naquela posição de memória "evanesce" quando a execução do programa é encerrada.

Do mesmo modo, o programa

```
Variaveis
        Numerico Num
Inicio
        Ler Num
Fim
```

armazenará (por um curtíssimo intervalo de tempo) um número digitado no teclado. Este exemplo está sendo dado para que leitor seja alertado a respeito do seguinte ponto importante. Se for digitado um número inteiro pertencente ao intervalo fechado [-31768, 32767], ele será armazenado e tratado como um inteiro realmente, enquanto que se for digitado um inteiro que não pertence a este

intervalo ele será armazenado como um número fracionário (com a parte decimal igual a zero). Se um inteiro deste fosse utilizado, por exemplo, para o cálculo do resto de uma divisão ocorreria erro.

2.7 Comando de saída

De maneira natural, é necessário que o sistema disponha de mecanismos que permita que ele forneça os resultados da execução do algoritmo. Isto é feito através do *comando de saída*, que deve ser escrito com a seguinte sintaxe:

Escrever Lista de parâmetros

onde *Lista de parâmetros* pode conter (separados por vírgulas) identificadores de variáveis, expressões (expressões aritméticas e lógicas e relações) e mensagens.

No primeiro caso, o conteúdo da variável associada ao comando é exibido na tela do vídeo; no segundo caso, a expressão é avaliada e o seu resultado é exibido e no terceiro caso, a mensagem (que deve ser escrita entre aspas) é exibida. Por exemplo,

```
Inicio
    Escrever "Estou aprendendo a programar"
Fim
```

é um programa em ILA, cuja execução faz com que seja exibida na tela a mensagem *Estou aprendendo a programar*. Naturalmente, toda vez que este programa for executado será exibida a mensagem acima. Se o usuário pretende que a mensagem a ser exibida dependa da sua inspiração na hora da execução (ou do seu objetivo), o programa deveria ser modificado para

```
Variaveis
    Caracter Frase
Inicio
    Ler Frase
    Escrever Frase
Fim
```

Agora a mensagem a ser exibida será aquela que for digitada durante a execução do comando *Ler Frase*. Por exemplo, se o usuário no momento da execução do programa estiver pensando na sua amada, pode na execução do comando *Ler Frase* digitar a frase *Eu te amo* e será esta a frase que aparecerá na tela.

Para outro exemplo, considere o seguinte programa:

```
Variaveis
    Numerico x, y, z
Inicio
    Ler x
    Ler y
    Ler z
    Escrever (x + y + z)/3
Fim
```

Quando da sua execução, o sistema, para execução dos comandos Ler x, Ler y e Ler z, aguarda que sejam digitados três valores numéricos. Quando isto é feito, o sistema armazena estes três valores, respectivamente, nas variáveis x, y e z. Para execução do último comando, o ILA avalia a expressão (x + y + z)/3 e exibe o seu valor na tela. Assim, o programa fornece a média aritmética de três números dados. Para mais exemplo, observe que o programa

```
Inicio
    Escrever 5 < 3
Fim
```

exibe na tela o valor *falso*.

No nosso entendimento, não é uma boa prática se utilizar uma expressão num comando *Escrever*. Se o valor de uma expressão é útil para alguma coisa, ele deve ser armazenado em alguma variável (veremos isto na próxima seção) e esta deve ser utilizada para o fornecimento de resultados.

A possibilidade de emissão de mensagens através do comando *Escrever* permite que o próprio programa facilite a sua execução e que torne compreensíveis os resultados fornecidos. Da forma em que está escrito, a execução do programa acima que fornece a média de três números dados é dificultada pelo fato de que a execução dos comandos de leitura faz com que o sistema aguarde a digitação dos números pretendidos (o *cursor* fica simplesmente piscando na tela) e o usuário pode não saber o que está se passando. Além disto, a execução do comando Escrever exibe apenas o resultado da expressão, sem indicação do que aquele valor se refere. Assim, o programa referido ficaria muito melhor da seguinte forma

```
Variaveis
        Numerico x, y, z
Inicio
        Escrever "Digite os três números"
        Ler x
        Ler y
        Ler z
        Escrever "A média dos números ", x, " , ", y, " e ", z, " é igual a ",  (x + y + z)/3
Fim
```

Outro detalhe importante é que um comando deve ser escrito numa única linha. Assim, o leitor deve ver o tal último comando - e todos os comandos dos exemplos ao longo do livro - como se estivessem numa mesma linha.

2.8 Comando de atribuição

A seção 2.6 apresentou o comando que permite que se armazene em variáveis a entrada do programa. Agora veremos como armazenar dados gerados pela sua execução. Enfatizando o que foi dito na seção 1.5 relativo ao fato de que algoritmos podem manipular dados gerados por execuções de instruções anteriores, considere um programa para o cálculo da média de uma relação de números. Naturalmente, a quantidade de números da relação (se não foi fornecida a priori) deve ser de alguma forma determinada e armazenada em alguma variável para que possa ser utilizada no cálculo final da média pretendida.

O armazenamento de dados gerados pelo próprio programa, alterações no conteúdo de uma variável e determinações de resultados finais de um processamento são realizados através do *comando de atribuição,* que deve ser escrito com a seguinte sintaxe.

Identificador de variável = expressão

Naturalmente, a expressão do segundo membro pode se resumir a um valor constante pertencente ao tipo de dado da variável do primeiro membro. Neste caso, este valor é armazenado naquela variável. Se não for este o caso, a expressão é avaliada e, se for do mesmo tipo da variável do primeiro membro, o resultado é armazenado na variável. A expressão do segundo membro pode envolver a própria variável do primeiro membro. Neste caso, o conteúdo anterior da variável será utilizado para a avaliação da expressão e será substituído pelo valor desta expressão. Por exemplo, se i é uma variável do tipo *Numerico*, o comando i = i + 1, faz com que o seu conteúdo seja incrementado de uma unidade.

2.9 Exemplos Parte I

1. Voltando ao programa do cálculo da média de três números dados, observe que a média foi calculada e exibida, não tendo sido armazenada. Se este programa fizesse parte de um programa maior (e isto normalmente acontece! Não se usa computação para uma questão tão simples!) e esta média fosse necessária em outra parte do programa, aquele trecho teria que ser rescrito. É uma boa prática, portanto, que resultados finais de processamento sejam armazenados em variáveis, sendo então os conteúdos destas variáveis exibidos através do comando *Escrever*. Assim, o programa referido ficaria melhor escrito da seguinte forma.

```
//Programa que determina a média aritmética de três números dados
Variaveis
    Numerico x, y, z, Media
Inicio
    Escrever "Digite os três números"
    Ler x
    Ler y
    Ler z
    Media = (x + y + z)/3
    Escrever "A média dos números ", x, " , ", y, " e ", z, " é igual a ",  Media
Fim
```

Uma linha de um programa em ILA iniciada por // não é traduzida para linguagem de máquina pelo interpretador. Isto permite, como fizemos acima, incluir no texto *comentários*, que servirão para esclarecimentos sobre os objetivos do programa, finalidades de suas variáveis e das *funções* (ver capítulo 5) nele definidas. Estes esclarecimentos facilitam a compreensão do programa numa leitura posterior. É uma boa prática de programação a inclusão comentários nos textos dos programas.

2. O exemplo a seguir se justifica pelo fato de que o operador / realiza divisões decimais. Veremos posteriormente, às vezes, necessitamos obter o quociente da divisão inteira de dois inteiros dados. Por exemplo, se os inteiros dados forem 30 e 7, o quociente é igual a 4. Como o ILA calcula o resto de divisão inteira (ver seção 2.2), podemos utilizar a *identidade fundamental da divisão*

dividendo = divisor x quociente + resto

para escrever um programa que receba dois inteiros e forneça o quociente da divisão entre eles.

```
//Programa que determina o quociente de uma divisão inteira
Variaveis
    Numerico Dividendo, Divisor, Quoc, Rest
Inicio
    Escrever "Digite o dividendo e o divisor - divisor diferente de zero!"
    Ler Dividendo
    Ler Divisor
    Rest = Resto(Dividendo, Divisor)
    Quoc = (Dividendo - Rest)/Divisor
    Escrever "O quociente da divisão ", Dividendo,"/",Divisor,", é igual a ", Quoc
Fim
```

Na seção 4.4 apresentaremos um programa para calcular quocientes de divisões, supondo que o sistema não calcula restos de divisões inteiras.

3. Agora apresentaremos um programa que recebendo um número inteiro como entrada fornece o algarismo da casa das unidades deste número, questão discutida na seção 1.6. Como vimos naquela seção, o algarismo procurado é o resto da divisão do número dado por 10. Temos então o seguinte programa.

```
//Programa que determina o algarismo da casa das unidades de um inteiro dado
Variaveis
```

```
        Numerico Num, Unidade
Inicio
        Escrever "Digite um inteiro"
        Ler Num
        Unidade = Resto(Num, 10)
        Escrever "O algarismo das unidades de ", Num, " é ", Unidade
Fim
```

4. Se quiséssemos um programa para inverter um número com dois algarismos (por exemplo, se a entrada fosse 74, a saída deveria ser 47) poderíamos utilizar o seguinte fato: se x e y são os algarismos de um número (casa das dezenas e das unidades, respectivamente), então este número é x . 10 + y. Assim, a inversão seria y . 10 + x (no exemplo, 74 = 7 . 10 + 4; 47 = 4 . 10 + 7) e bastaria extrair os dois algarismos do número dado e utilizar a expressão acima. A extração do algarismo da casa das unidades foi mostrada acima. E o algarismo da casa das dezenas? Basta ver que ele é o quociente da divisão do número por 10 e o exemplo 2 apresenta uma solução para se determinar o quociente de uma divisão inteira. Temos então o seguinte programa.

```
//Programa que inverte um número com dois algarismos
Variaveis
        Numerico Num, Invertido, Unidade, Dezena
Inicio
        Escrever "Digite o numero"
        Ler Num
        Unidade = Resto(Num, 10)
        Dezena = (Num - Unidade)/10
        Invertido = Unidade*10 + Dezena
        Escrever "O invertido de ", Num, " , ", Invertido
Fim
```

Dificilmente, o caro leitor vai escrever um programa com este objetivo (para que serve inverter um número com dois algarismos?). Esta questão e algumas outras estão sendo discutidas aqui apenas como exemplos para o desenvolvimento da lógica de programação.

5. Imagine agora que queiramos um programa que determine o maior múltiplo de um inteiro dado menor que um outro inteiro dado. Por exemplo, se a entrada fosse 13 e 100, a saída deveria ser 91 (91 é o maior múltiplo de 13 que é menor que 100).

Como dividendo = divisor x quociente + resto e resto < divisor, temos que o valor da expressão dividendo - resto é o múltiplo procurado.

```
//Programa que determina o maior múltiplo de um inteiro menor que outro inteiro
Variaveis
        Numerico Num, Divisor, MaiorMultiplo
Inicio
        Escrever "Digite o numero do qual se quer o múltiplo"
        Ler Divisor
        Escrever "Digite o inteiro que deve ser maior do que o múltiplo"
        Ler Num
        MaiorMultiplo = Num - Resto(Num, Divisor)
        Escrever "O maior múltiplo de ", Divisor, " menor do que ", Num, " é ", MaiorMultiplo.
Fim
```

6. O programa a seguir, além de ser muito interessante no sentido do desenvolvimento da lógica de programação, será utilizado (a sequência de comandos do programa principal) em outros programas. O objetivo dele é permutar os conteúdos de duas variáveis. Ou seja, suponhamos que, através de comandos de entrada o programa armazenou nas variáveis x e y os valores 7 e 18 e

pretendamos que o programa faça com que o conteúdo de x passe a ser 18 e o de y passe a ser igual a 7. À primeira vista, bastaria a sequência de comandos

```
x = y
y = x
```

Ocorre que, quando o segundo comando fosse executado, o primeiro já teria sido e o conteúdo de x não seria mais o original. No nosso exemplo, teríamos a seguinte situação

x	y
17	8
8	
	8

e a permuta não teria sido feita, além do fato de que o conteúdo original de x teria sido perdido. Uma alternativa é considerar uma variável auxiliar que "guarde" o conteúdo de x antes que este seja substituído pelo conteúdo de y. Teríamos assim o seguinte programa.

```
//Programa que permuta os conteúdos de duas variáveis
Variaveis
        Numerico x, y, Aux
Inicio
        Escrever "Digite os valores iniciais"
        Ler x
        Ler y
        Escrever "Conteúdos  originais x = ", x, ", y = ", y
        Aux = x
        x = y
        y = Aux
        Escrever "Conteúdos após a permuta x = ", x, " , y = ", y
Fim
```

2.10 Funções pré-definidas

O ILA contém diversos programas com finalidades pré-determinadas e que podem ser executados durante a execução de um outro programa. Estes programas são chamados *funções pré-definidas* e as suas execuções podem ser solicitadas em expressões escritas pelo programador (a solicitação da execução de uma função é dita *ativação* ou *chamada* da função). Para isto o programador deve conhecer o *identificador* da função, quantos e de que tipo são os *argumentos* com que elas devem ser ativadas (este conjunto constitui a *sintaxe* da função) e o tipo de valor que ela *retorna* ao programa quando termina sua execução. A definição de uma função pré-definida se faz através da sintaxe.

Identificador da função(Lista de argumentos)

podendo a lista de argumentos ser vazia. A tabela a seguir apresenta algumas das funções pré-definidas do ILA, onde o caractere **-** na coluna Tipo de Argumentos indica que a Lista de Argumentos é vazia.

Algumas funções pré-definidas do ILA

Ident(Argumentos)	Tipo dos Argumento	O que retorna
Acos(x)	numerico	Arco cujo valor do cosseno é x
Aleatorio()	-	Um número aleatório entre 0 e 1
Asen(x)	numerico	Arco cujo valor do seno é x
Atan(x)	numerico	Arco cujo valor da tangente é x
Comprimento(x)	caracter	O número de caracteres de x
Cos(x)	numerico	Cosseno de x
Inteiro(x)	numerico	Parte inteira de x
Log(x)	numerico	Logaritmo decimal de x
Raiz(x, y)	numerico	Raiz índice y de x
Sen(x)	numerico	Seno de x
Tan(x)	numerico	Tangente de x

2.11 Exemplos Parte II

Podemos usar as funções pré-definidas para escrever programas que realizem outras tarefas específicas, que por si só não teriam importância, mas que, sendo escritos como *funções* (veremos isto no capítulo 5), são úteis em outros programas.

7. O exemplo a seguir fornece a parte fracionária de um número dado. Como o sistema é capaz de extrair a parte inteira do número, através da função *Inteiro*, sua parte fracionária é a diferença entre ele e esta sua parte inteira.

```
//Programa que fornece a parte fracionária de um número dado
Variaveis
        Numerico Num, Frac
Inicio
        Escrever "Digite o numero"
        Ler Num
        Frac = Num - Inteiro(Num)
        Escrever "A parte fracionaria de ", Num, " é ", Frac
Fim
```

8. O programa abaixo arredonda para inteiro um número decimal positivo de acordo com a regra usual: se a parte fracionária é maior do que ou igual a 0,5, o número é arredondado para o inteiro subsequente; caso contrário, é arredondado para o inteiro anterior. Basta observar que se a parte fracionária do número é maior do que ou igual a 0,5, pela regra do "vai um", a parte inteira do seu dobro é uma unidade maior que o dobro da sua parte inteira e se a parte fracionária é menor que 0,0, a parte inteira do seu dobro é igual ao dobro da sua parte inteira. Por exemplo, para x = 5,6, Inteiro(2*x) = Inteiro(11,2) = 11 e 2*Inteiro(x) = 2*Inteiro(5,6) = 2*5 = 10 e para y = 5,4, Inteiro(2*x) = Inteiro(10,8) = 10.

```
//Programa que efetua arredondamentos
Variaveis
        Numerico Num, Arred
Inicio
        Escrever "Digite o número"
        Ler Num
        Arred = Inteiro(2*Num) - Inteiro(Num)
        Escrever Num, " é arredondado para ", Arred
```

```
Fim
```

9. O exemplo a seguir tem objetivos diferentes dos anteriores. O primeiro objetivo é motivar o próximo capítulo e o segundo é mostrar que um programador só é capaz de escrever um programa que resolva um determinado problema se ele souber resolver o tal problema "na mão", ou seja, com a utilização apenas de lápis e papel. Trata-se de um programa que calcule a área de um triângulo, dados os comprimentos dos seus lados. Naturalmente, só é capaz de escrever este programa aquele que conhecer a fórmula abaixo, que dá área do triângulo cujos lados têm comprimentos x, y e z.

$$S = \sqrt{p.(p-x).(p-y).(p-z)}$$

onde $p = \frac{x+y+z}{2}$ é o *semiperímetro* do triângulo. Com isto, temos o seguinte programa.

```
//Programa que fornece a área de um triângulo
Variaveis
    Numerico x, y, z, p, Area
Inicio
    Escrever "Digite os comprimentos dos lados do triângulo"
    Ler x
    Ler y
    Ler z
    p = (x + y + z)/2
    Area = Raiz(p*(p - x)*(p - y)*(p - z), 2)
    Escrever "A área do triângulo de lados ", x, ", ", y, " e ", z, " é igual a ", Area
Fim
```

Se este programa for executado com entrada 3, 4 e 5 temos p = 6 e

$$Area = \sqrt{6.(6-3).(6-4).(6-5)} = \sqrt{36} = 6$$

e, como era de se esperar, a área do triângulo cujos lados têm comprimento 3, 4 e 5 unidades de comprimento é igual a 6 unidades de área.

Agora, se este programa fosse executado para entrada 1, 2 e 5 teríamos p = 4 e

$$Area = \sqrt{4.(4-1).(4-2).(4-5)} = \sqrt{-24}$$

e ocorreria erro: o sistema, como era de se esperar, não calcula raiz quadrada de número negativo.

O que acontece é que nem sempre três números podem ser comprimentos dos lados de um triângulo (a matemática prova que isto só acontece se cada um deles for menor do que a soma dos outros dois). Assim, o comando que calcula a Area só deveria ser executado se os valores digitados para x, y, e z pudessem ser comprimentos dos lados de um triângulo.

2.12 Exercícios propostos

1. Avalie cada uma das expressões abaixo.

a) (-(-9) + Raiz((-9)*(-9) - 4*3*6, 2))/(2*3).

b) (Inteiro(5,7) = 5) e (Raiz(9, 3) = 3)) ou Resto(4, 8) = 3.

2. Escreva programas para

a) Converter uma temperatura dada em graus Farenheit para graus Celsius.

b) Gerar o *invertido* de um número com três algarismos (exemplo: o *invertido* de 498 é 894).

c) Somar duas frações ordinárias, fornecendo o resultado em forma de fração.

d) Determinar o menor múltiplo de um inteiro dado maior do que um outro inteiro dado (exemplo: o menor múltiplo de 7 maior que 50 é 56).

e) Determinar o perímetro de um polígono regular inscrito numa circunferência, dados o número de lados do polígono e o raio da circunferência.

3. O exemplo 2 da seção 2.9 apresentava um programa que utilizava o resto da divisão para determinar o quociente da divisão de dois inteiros positivos dados. Reescreva o programa referido utilizando a função pré-definida Inteiro.

4. Escreva um programa que permute o conteúdo de duas variáveis sem utilizar uma variável auxiliar (ver exemplo 6 da seção 2.9).

5. Uma loja vende seus produtos no sistema entrada mais duas prestações, sendo a entrada maior do que ou igual às duas prestações, que devem ser iguais, inteiras e as maiores possíveis. Por exemplo, se o valor da mercadoria for R$ 270,00, a entrada e as duas prestações são iguais a R$ 90,00; se o valor da mercadoria for R$ 302,75, a entrada é de R$ 102,75 e as duas prestações são a iguais a R$ 100,00. Escreva um programa que receba o valor da mercadoria e forneça o valor da entrada e das duas prestações, de acordo com as regras acima. Observe que uma justificativa para a adoção desta regra é que ela facilita a confecção e o consequente pagamento dos boletos das duas prestações.

6. Um intervalo de tempo pode ser dado em horas, dias, minutos, segundos ou sequências "decrescentes" destas unidades (em dias e horas; em horas e minutos; em horas, minutos e segundos), de acordo com o interesse de quem o está manipulando. Escreva um programa que converta um intervalo de tempo dado em segundos em horas, minutos e segundos. Por exemplo, se o tempo dado for 3 850 segundos, o programa deve fornecer 1 h 4 min 10 s.

7. Escreva um programa que converta um intervalo de tempo, dado em minutos, em horas, minutos e segundos. Por exemplo, se o tempo dado for 145,87 min, o programa deve fornecer 2 h 25 min 52,2 s.

8. Um programa para gerenciar os saques de um caixa eletrônico deve possuir algum mecanismo para decidir o número de notas de cada valor que deve ser disponibilizado para o cliente que realizou o saque. Um possível critério seria o da "distribuição ótima" no sentido de que as notas de menor valor fossem distribuídas em número mínimo possível. Por exemplo, se a quantia solicitada fosse R$ 87,00, o programa deveria indicar uma nota de R$ 50,00, três notas de R$ 10,00, uma nota de R$ 5,00 e duas notas de R$ 1,00. Escreva um programa que receba o valor da quantia solicitada e retorne a distribuição das notas de acordo com o critério da distribuição ótima.

8. De acordo com a Matemática Financeira, o cálculo das prestações para amortização de um financiamento de valor F em n prestações e a uma taxa de juros i é dada pela fórmula $P = F/a_{n\text{–}i}$, onde $a_{n\text{–}i} = ((1 + i)^n - 1)/(i \,.\, (1 + i)^n)$. Escreva um programa que determine o valor das prestações para amortização de um financiamento, dados o valor do financiamento, o número de prestações para amortização e a taxa de juros.

Observação

Para obter o caderno das soluções dos exercícios propostos basta encaminhar e-mail para jaime@ccen.ufal com assunto RESPOSTAS LIVRO ILA e contendo NOME, CATEGORIA (docente, discente, autodidata), CIDADE/ESATADO e, se for o caso, INSTITUIÇÃO/CURSO.

3. Estruturas de seleção

3.1 Introdução

O último exemplo do capítulo anterior apresentava um programa para calcular a área de um triângulo, dados os comprimentos dos seus lados. Foi visto que o comando que calculava área solicitada só devia ser executado com a certeza anterior de que os valores dados como entrada poderiam ser comprimentos dos lados de um triângulo. Ou seja, o tal comando só deveria ser executado *se* $x < y + z$ *e* $y < x + z$ *e* $z < x + y$, condição que garante que os valores armazenados nas variáveis x, y e z são comprimentos dos lados de um triângulo. Assim, em algumas situações, alguns comandos só devem ser executados se alguma condição for verificada. A verificação de que uma condição é satisfeita e, a partir daí, uma determinada sequência de comandos deve ser executada é chamada de *estrutura de seleção*, *estrutura de decisão* ou *comando de seleção*.

3.2 Comando *Se*

O comando *Se* é uma estrutura de decisão que decide entre duas sequências de comandos qual aquela que será executada ou se uma dada sequência de comandos será ou não executada. No primeiro caso, sua sintaxe é a seguinte:

```
Se (Expressão Lógica) entao
        sequência de comandos 1
Senao
        sequência de comandos 2
Fim_se
```

A semântica deste comando é explicada pela própria sintaxe: se a expressão lógica for verdadeira, o sistema executará a sequência de comandos 1; se for falsa o sistema executará a sequência de comandos 2.

Por exemplo, se queremos um programa que verifique a paridade de um número dado, poderíamos verificar se o resto da divisão do número por dois é igual a 1. Se isto for verdade, o número é ímpar; se não for verdade, o número dado par. Teríamos então o seguinte programa.

```
Variaveis
        Numerico Num
Inicio
        Escrever "Digite o número"
        Ler Num
        Se Resto(Num, 2) = 1 entao
                Escrever Num, " é ímpar"
        Senao
                Escrever Num, " é par"
        Fim_se
Fim
```

No segundo caso (aquele em que o comando *Se* indica ao sistema se uma sequência de comandos deverá ou não ser executada) a sintaxe do comando é simplesmente

```
Se (Expressão Lógica) entao
        sequência de comandos
Fim_se
```

Agora, se a expressão lógica for verdadeira, a sequência de comandos é executada; se não for verdadeira, a sequência referida não é executada e o sistema passa a executar o comando seguinte ao comando S*e*.

Por exemplo, se se pretende ordenar em ordem crescente dois números dados armazenados nas variáveis x e y, só há de necessidade de se realizar alguma ação se o número armazenado em y for maior do que aquele armazenado em x. Neste caso, o que devia ser feito era a permuta dos conteúdos de x e de y. Teríamos então o seguinte programa.

```
//Programa para ordenação de dois números
Variaveis
    Numerico x, y, Aux
Inicio
    Escrever "Digite os dois números"
    Ler x
    Ler y
    Se x > y entao
        Aux = x
        x = y
        y = Aux
    Fim_se
    Escrever "Números dados escritos em ordem crescente: ", x, " , ", y
Fim
```

Observe que estamos procurando escrever os comandos da sequência vinculada à estrutura de decisão com uma tabulação diferente da tabulação em que estão postos o *Se* e o *Fim_se*. Esta forma de se editar um programa, chamada *endentação*, deve ser praticada por todo programador, pois ela é muito útil para a legibilidade dos programas.

3.3 Exemplos Parte III

1. No exemplo 9 da seção 2.11, apresentamos um programa que calculava a área de um triângulo, dados os seus lados. No final dele, mostramos que o mesmo nem sempre fornecia respostas satisfatórias para todas as entradas e comentamos que o cálculo da área deveria ser precedido da verificação de que os dados de entrada são de fato comprimentos dos lados de um triângulo. O programa referido, escrito agora de forma completa e correta, seria o seguinte.

```
//Programa que determina a área de um triângulo
Variaveis
    Numerico x, y, z, p, Area
Inicio
    Escrever "Digite os comprimentos dos lados do triângulo"
    Ler x
    Ler y
    Ler z
    Se (x < y + z) e (y < x + z) e (z < x + y) entao
        p = (x + y + z)/2
        Area = Raiz(p*(p - x)*(p - y)*(p - z), 2)
        Escrever "A área do triângulo de lados ", x, ", ", y, " e ", z, " é igual a ", Area
    Senao
        Escrever "Os valores dados não são comprimentos dos lados de um triângulo"
    Fim_se
Fim
```

2. Programas que manipulam datas (por exemplo, um programa que determine o número de dias entre duas datas dadas) contêm trechos que verificam se um ano dado é bissexto. Sabendo que um ano é bissexto se ele é múltiplo de quatro, teríamos o seguinte programa.

```
//Programa que verifica se um dado ano é bissexto
Variaveis
        Numerico Ano
Inicio
        Escrever "Digite o ano"
        Ler Ano
        Se Resto(Ano, 4) = 0 entao
                Escrever  Ano, " é bissexto"
        Senao
                Escrever Ano, " não é bissexto "
        Fim_se
Fim
```

3. O programa para ordenar dois números dados visto na seção 3.2 é um caso muito particular da questão mais geral da *ordenação* de uma relação de números ou de nomes, problema muito importante para a computação, também conhecido como *classificação*. Para a solução geral existem diversos algoritmos com este objetivo. No capítulo 7 teremos oportunidade de discutir dois destes algoritmos. Por enquanto, vejamos um programa que ordene três números dados. Além de exemplificar o comando *Se*, o programa abaixo mostra como se pode (e se deve) utilizar raciocínios anteriores para se escrever programas.

Seja então um programa que receba três números inteiros, armazene-os nas variáveis x, y e z e que ao final da sua execução deixe os conteúdos de x, de y e de z na ordem crescente. Uma idéia bem interessante é armazenar na variável x o menor dos números e em seguida ordenar os conteúdos de y e de z, que é exatamente o problema de ordenar dois números, o qual foi referido acima. Obviamente, para se executar a primeira ação pretendida (armazenar na variável x o menor dos números) só é necessário se fazer alguma coisa se o valor x já não for o menor dos números dados, ou seja, se x > y ou x > z. Nesta hipótese, o menor deles é y ou z e este menor deve ser permutado com x. Temos então o seguinte programa.

```
//Programa que ordena três números
Variaveis
        Numerico x, y, z, Aux
Inicio
        Escrever "Digite os três números"
        Ler x
        Ler y
        Ler z
        Se (x > y) ou (x > z) entao
                Se y > z entao          //z é o menor
                        Aux = x         //permuta x e z
                        x = z
                        z = Aux
                Senao                   //y é o menor
                        Aux = x         //permuta x e y
                        x = y
                        y = Aux
                Fim_se
        Fim_se
        Se y > z entao
                Aux = y                 //permuta y e z
                y = z
                z = Aux
        Fim_se
```

```
        Escrever "Números dados escritos em ordem crescente: ", x, ", ", y, " e ", z
Fim
```

Observe que se a expressão lógica do primeiro comando *Se* for verdadeira, o sistema executará outro comando *Se*. Neste caso, dizemos que os comandos estão *aninhados*. Naturalmente, deve haver um *Fim_se* para cada comando *Se*. Observe também que escrevemos no programa algumas frases explicativas das ações pretendidas. Esta frases são chamadas *comentários* e devem ser escritas precedidas de //. Quando o interpretador encontra as duas barras desconsidera tudo o que vem depois delas até o final da linha. Isto permite que o programador deixe registrado no próprio programa os comentários que ele achar conveniente. Como a edição dos programas com endentação, a prática de se colocar comentários nos programas é muito importante. Como os programas discutidos neste livro serão precedidos de explicações prévias, a utilização de comentários aqui vai se restringir a indicação do objetivo do programa.

4. Um outro exemplo que ilustra muito bem a utilização do comando *Se* é um programa para determinar as raízes de uma equação do segundo grau. Sabemos da matemática que uma equação $ax^2 + bx + c = 0$ só tem raízes reais se $b^2 - 4ac \geq 0$. Assim, um programa para encontrar as raízes reais (deixaremos o caso completo da determinação das raízes reais e complexas como exercício proposto) poderia ser o seguinte.

```
//Programa para determinação das raízes de uma equação do segundo grau
Variaveis
        Numerico a, b, c, Delta, x1, x2
Inicio
        Escrever "Digite os coeficiente da equação"
        Ler a
        Ler b
        Ler c
        Se a <> 0 entao
                Delta = b*b - 4*a*c
                Se Delta >= 0 entao
                        x1 = (-b + Raiz(Delta, 2))/(2*a)
                        x2 = (-b - Raiz(Delta, 2))/(2*a)
                        Escrever "As raízes da equação dada são ", x1, " e ", x2
                Senao
                        Escrever "A equação dada não tem raízes reais"
                Fim_se
        Senao
                Escrever "A equação dada não é do segundo grau"
        Fim_se
Fim
```

5. Imaginemos agora uma escola que adote no seu processo de avaliação a realização de quatro avaliações bimestrais e que o regime de aprovação dos alunos seja o seguinte:

i) Se a média das avaliações bimestrais for superior ou igual a 7,0, o aluno está *aprovado*, com média final igual à média das avaliações bimestrais.

ii) Se a média das avaliações bimestrais for inferior a 5,0, o aluno está *reprovado*, com média final igual à média das avaliações bimestrais.

iii) Não ocorrendo nenhum dos casos anteriores, o aluno se submete a uma *prova final* e a sua média final será a média ponderada desta prova final (com peso 4) e a média das avaliações bimestrais (com peso 6). Neste caso, o aluno estará aprovado se a sua média final for superior ou igual a 5,5.

O programa abaixo recebendo as notas das avaliações bimestrais e, se for o caso, a nota da prova final, fornece a média final do aluno e a sua condição em relação à aprovação.

```
//Programa para verificar aprovação de um aluno
Variaveis
    Numerico Bim1, Bim2, Bim3, Bim4, MedBim, PrFinal, MedFinal
Inicio
    Escrever "Digite as notas bimestrais"
    Ler Bim1
    Ler Bim2
    Ler Bim3
    Ler Bim4
    MedBim = (Bim1 + Bim2 + Bim3 + Bim4)/4
    MedFinal = MedBim
    Se (MedBim >= 5) e (MedBin < 7) entao
        Escrever "Digite a nota da prova final"
        Ler PrFinal
        MedFinal = (MedBim*6 + PrFinal*4)/10
    Fim_se
    Se MedFinal >= 5.5 entao
        Escrever "Aluno aprovado com média final igual a ", MedFinal
    Senao
        Escrever "Aluno reprovado com média final igual a ", MedFinal
    Fim_se
Fim
```

3.4 Comando *Faca caso*

Suponhamos que uma empresa decidiu dar um aumento escalonado a seus funcionários de acordo com a seguinte regra: 13% para os salários inferiores ou iguais a R$ 200,00; 11% para os salários situados entre R$ 200,0 e R$ 400,00 (inclusive); 9 % para os salários entre R$ 400,00 e R$ 800,00 (inclusive) e 7% para os demais salários. Naturalmente, pode-se escrever um programa com o comando *Se* que receba o salário atual de um funcionário desta empresa e retorne o valor do seu novo salário.

```
//Programa para atualizar salários
Variaveis
    Numerico SalarioAtual, SalarioNovo, Indice
Inicio
    Escrever "Digite o salário atual"
    Ler SalarioAtual
    Se SalarioAtual <= 200 entao
        Indice = 1.13
    Senao
        Se (SalarioAtual > 200) e (SalarioAtual <= 400) entao
            Indice = 1.11
        Senao
            Se (SalarioAtual > 400) e (SalarioAtual <= 800) entao
                Indice = 1.09
            Senao
                Indice = 1.07
            Fim_se
        Fim_se
    Fim_se
    SalarioNovo = SalarioAtual * Indice
```

```
        Escrever "Salario Atual = ", SalarioAtual
        Escrever "Salario Novo = ", SalarioNovo
Fim
```

Observe que os diversos comandos *Se* aninhados dificultam sobremaneira a legibilidade do programa. Observe também que entre as quatro opções de "sequências" de comandos *Indice = 1.13*; *Indice = 1.11*; *Indice = 1.09*; *Indice = 1.07*; somente uma será executada.

Com uma melhor legibilidade, o comando *Faca caso* seleciona entre várias sequências de comandos qual aquela que deve ser executada ou se nenhuma delas deve ser executada. Sua sintaxe é:

```
Faca caso
        Caso Expressão lógica 1 :
                sequência de comandos 1
        Caso Expressão lógica 2 :
                sequência de comando 2
        . . .
        Caso Expressão lógica n :
                sequência de comando n
        Outro_caso :
                sequência de comando x
Fim_caso
```

A semântica deste comando é bem simples. Se a *expressão lógica k*, para $k = 1, 2, ..., n$, for verdadeira, a *sequência de comandos k* é executada. Se para todo $k = 1, 2, ..., n$ a *expressão lógica k* for falsa, a *sequência de comandos x* (aquela "vinculada" ao *Outro_caso*) é executada. Esta opção, *Outro_caso*, é facultativa. Se ela não existir e todas as expressões lógicas forem falsas, nada é executado e o processamento passa para o comando que segue o *Fim_caso*.

Uma aplicação usual do comando *Faca caso* é em programas que contêm *menu de opções*. Isto acontece quando um programa é capaz de realizar de modo independente várias tarefas. Por exemplo, um programa que gerencie um caixa eletrônico de um banco deve oferecer ao usuário algumas opções em relação à ação que ele pretende realizar na sua conta como, por exemplo, a emissão do saldo atual, a emissão de um extrato, a realização de um saque e a realização de um depósito. Se fôssemos escrever um programa em ILA com este propósito, ele poderia ter a seguinte estrutura:

```
Variaveis
        Numerico Op, ...
        . . .
Inicio
        Escrever "1-Saldo"
        Escrever "2-Extrato"
        Escrever "3-Saque"
        Escrever "4-Depósito"
        Escrever "Digite sua opção:
        Ler Op
        Faca caso
                Caso Op = 1:
                        Sequência de comandos para emissão do saldo
                Caso Op = 2
                        Sequência de comandos para emissão do saldo
                Caso Op = 3:
                        Sequência de comandos para realização do saque
                Caso Op = 4:
```

```
                Sequência de comandos para realização do depósito
        Fim_caso
Fim
```

3.5 Exercícios propostos

1. De um modo geral, as ligações telefônicas são cobradas pelas suas durações. O sistema registra os instantes em que a ligação foi iniciada e concluída e é acionado um programa que determina o intervalo de tempo decorrido entre aqueles dois instantes dados. Escreva um programa receba dois instantes dados em horas e minutos e determine o intervalo de tempo (em horas e minutos) decorrido entre eles.

2. Escreva um programa que realize arredondamentos de números utilizando a regra usual da matemática: se a parte fracionária for maior do que ou igual a 0,5, o número é arredondado para o inteiro imediatamente superior, caso contrário, é arredondado para o inteiro imediatamente inferior.

3. Escreva um programa para verificar se um inteiro dado é um quadrado perfeito, exibindo, nos casos afirmativos, sua raiz quadrada.

4. Escreva um programa para determinar o maior de três números dados.

5. Escreva um programa para classificar um triângulo de lados de comprimentos dados em *escaleno* (os três lados de comprimentos diferentes), *isósceles* (dois lados de comprimentos iguais) ou *equilátero* (os três lados de comprimentos iguais).

6. Escreva um programa para verificar se um triângulo de lados de comprimentos dados é *retângulo*, exibindo, nos casos afirmativos, sua *hipotenusa* e seus *catetos*.

7. Escreva um programa para determinar as raízes reais ou complexas de uma equação do segundo grau, dados os seus coeficientes.

8. Escreva um programa para determinar a idade de uma pessoa, em anos meses e dias, dadas a data (dia, mês e ano) do seu nascimento e a data (dia, mês e ano) atual.

9. Escreva um programa que, recebendo as quatro notas bimestrais de um aluno da escola referida no exemplo 5 da seção 3.3, forneça a nota mínima que ele deve obter na prova final para que ele seja aprovado.

10. Em relação ao exemplo 2 da seção 3.3 e rigorosamente falando, há anos múltiplos de quatro que não são bissextos. São aqueles múltiplos de 100 que não são múltiplos de 400. Por exemplo, o ano 2000 foi um ano bissexto, mas o ano de 2100 não será. Reescreva o exemplo referido para que o programa detecte estas exceções.

Observação

Para obter o caderno das soluções dos exercícios propostos basta encaminhar e-mail para jaime@ccen.ufal com assunto RESPOSTAS LIVRO ILA e contendo NOME, CATEGORIA (docente, discente, autodidata), CIDADE/ESATADO e, se for o caso, INSTITUIÇÃO/CURSO.

4. Estruturas de repetição

4.1 Introdução

Um locutor brasileiro ao narrar um jogo de basquete ou de futebol americano nos Estados Unidos recebe a informação sobre a temperatura local medida em graus Fahrenheit. Naturalmente, ele deve fornecer aos brasileiros a temperatura em graus Celsius. Para isto, o locutor poderia executar o programa solicitado no primeiro item do segundo exercício da seção 2.12, fornecendo ao programa a temperatura em graus Fahrenheit e recebendo a correspondente temperatura em graus Celsius. Mais prático, talvez, seria a produção disponibilizar para o locutor uma tabela contendo as temperaturas possíveis em graus Fahrenheit e as correspondentes em graus Celsius. A confecção desta tabela poderia ser feita através de execuções sucessivas do programa referido, porém com a inconveniência da digitação de todas as entradas. Isto seria contornado se pudéssemos repetir a execução dos comandos que gerariam as temperaturas em graus Fahrenheit e as correspondentes em graus Celsius. O *ILA* possui os comandos *Para proximo* e *Faca enquanto*, chamados *estruturas de repetição*, cuja execução redunda na repetição da execução de uma determinada sequência de comandos.

4.2 Comando *Para proximo*

O comando *Para proximo* é uma estrutura de repetição que repete a execução de uma dada sequência de comandos um número de vezes determinado pelo próprio programa, devendo ser escrito com a seguinte sintaxe:

```
Para  Variável = Exp. 1 ate Exp. 2 Passo Exp. 3
      Sequência de comandos
Proximo
```

Aí *Exp 1*, *Exp 2* e *Exp 3* são expressões numéricas que são avaliadas quando da execução do comando. O valor de *Exp. 1* é então armazenado na variável *Variável* (chamada *variável de controle* da estrutura) e a *Sequência de comandos* é executada. Em seguida, a variável de controle é incrementada ou decrementada do valor de *Exp. 3* e o seu conteúdo é comparado com o valor de *Exp 2*. Se o valor de *Exp 3* for *positivo* (respectivamente, *negativo*) e o conteúdo da variável de controle for *maior* que o valor de *Exp 2* (respectivamente, *menor*) a execução da estrutura é encerrada; caso contrário, a *Sequência de comandos* é novamente executada e a comparação supra é novamente realizada.

Por exemplo, o programa

```
Variaveis
      Numerico i
Inicio
      Para i = 1 ate 8 passo 2
            Escrever i
      Proximo
Fim
```

exibe na tela os números 1, 3, 5, e 7, enquanto que o programa

```
Variaveis
      Numerico i
Inicio
      Para i = 7 ate 3 passo -1
            Escrever i
      Proximo
Fim
```

exibe na tela os números 7, 6, 5, 4, 3. Por seu turno, o programa

```
Variaveis
        Numerico i
Inicio
        Para i = 1 ate 8 passo -2
                Escrever i
        Proximo
Fim
```

exibe na tela, apenas, o número 1.

Vale observar que a indicação *Passo Exp. 3* é facultativa. Se ela for omitida o sistema assume o valor 1 para *Exp. 3* e, portanto, neste caso, a variável de controle é incrementada de uma unidade. Assim, e para mais um exemplo, a confecção da tabela de conversão citada na seção 4.1 poderia ser feita através da execução do seguinte programa.

```
//Programa que gera uma tabela de conversão de temperaturas dadas em graus Fahrenheit para graus Celsius
Variaveis
        Numerico Fahrenheit, Celsius
Inicio
        Para Fahrenheit = 32 ate 110
                Celsius = 5*(Fahrenheit - 32)/9
                Escrever Fahrenheit, "      ", Celsius
        Proximo
Fim
```

4.3 Comando *Faca enquanto*

Para introduzir uma nova estrutura de repetição e cotejá-la com o comando *Faca proximo*, considere um programa para encontrar um *divisor próprio* de um inteiro dado (um *divisor próprio* de um inteiro n é um divisor de n menor que n e diferente de 1). Com a utilização do comando *Para proximo* teríamos a seguinte solução.

```
//Programa que determina um divisor próprio de um inteiro positivo dado
Variaveis
        Numerico Num, i, Divisor
Inicio
        Escrever "Digite o número"
        Ler Num
        Divisor = Num
        Para i = 2 ate Num - 1
                Se Resto(Num, i) = 0 entao
                        Divisor = i
                Fim_se
        Proximo
        Se Divisor < Num entao
                Escrever  Divisor , " é divisor próprio de ", Num
        Senao
                Escrever Num, "não tem divisores próprios"
        Fim_se
Fim
```

Um problema com este programa é que ele retorna sempre, se existir, o maior divisor próprio.

Isto significa que se a entrada for um número par, por exemplo, a estrutura de repetição não é interrompida quando o divisor 2 é encontrado, o que, evidentemente, vai prejudicar a performance do programa. Isto pode ser contornado no *ILA*, pois este sistema admite que a variável de controle do comando *Para proximo* tenha o seu conteúdo alterado dentro do próprio comando. Com isto, o programa acima ficaria da seguinte forma.

```
//Programa para a determinação de  um divisor próprio de um inteiro positivo dado
Variaveis
        Numerico Num, i, Divisor
Inicio
        Escrever "Digite o número"
        Ler Num
        Divisor = Num
        Para i = 2 ate Num - 1
                Se Resto(Num, i) = 0 entao
                        Divisor = i
                        i = Num
                Fim_se
        Proximo
        Se Divisor < Num entao
                Escrever  Divisor , " é divisor próprio de ", Num
        Senao
                Escrever Num, "não tem divisores próprios"
        Fim_se
Fim
```

Nesta versão, quando o primeiro divisor próprio é encontrado, o comando i = Num faz com que a execução do comando *Para proximo* seja interrompida. A prática de alterar o conteúdo da variável de controle não será aqui incentivada pelo fato de que há situações em que não existe um número máximo de repetições possíveis: o placar de um set de voleibol, por exemplo. Na verdade, a questão central é que o comando *Para proximo* deve ser utilizado quando o número de repetições de execução de uma sequência de comandos é conhecido a priori. Quando isto não acontece (que é o caso do exemplo anterior: não se sabe a priori se e quando um divisor próprio vai ser encontrado), deve-se usar o comando *Faca enquanto*, que possui a seguinte sintaxe:

```
Faca enquanto Expressão lógica
        Sequência de comandos
Fim_enquanto
```

A semântica deste comando é óbvia: *enquanto* a expressão lógica é verdadeira a sequência de comandos é executada. Naturalmente, pode ocorrer que a sequência de comandos não seja executada nenhuma vez, isto ocorrendo se a *Expressão lógica* for falsa quando da "primeira" execução do comando. Por outro lado, é necessário que um dos comandos da sequência de comandos altere conteúdos de variáveis que aparecem na *Expressão lógica* de modo que em algum instante ela se torne falsa. Do contrário, a sequência de comandos terá sua execução repetida indefinidamente, o programa nunca termina e, evidentemente, não executa a tarefa para a qual foi desenvolvido. Quando isto acontece é comum se dizer que o programa está em *loop.*

Com o *Faca enquanto* as questões levantadas acima sobre o programa para determinar um divisor próprio de um inteiro dado são resolvidas e temos o seguinte programa:

```
//Programa que determina um divisor próprio de um inteiro positivo dado
Variaveis
        Numerico Num, i, Met
Inicio
```

```
        Escrever "Digite o número"
        Ler Num
        i = 2
        Met = Num/2
        Faca enquanto (Resto(Num, i) <> 0) e (i < Met)
                i = i + 1
        Fim_enquanto
        Se (i <= Met) entao
                Escrever i, " é divisor de ", Num
        Senao
                Escrever Num, " não tem divisores"
        Fim_se
Fim
```

Observe que a utilização da variável Met é para melhorar a performance do programa já que se um inteiro não possui um divisor próprio menor do que sua "metade", então ele não possui divisores próprios. O programa pode ser melhorado substancialmente, utilizando-se o fato discutido em [Evaristo, J 2010] de que se um inteiro não possui um divisor próprio menor do que sua raiz quadrada, ele não possuirá divisores próprios. Levando em conta este fato, teríamos o seguinte programa.

```
//Programa que determina um divisor próprio de um inteiro positivo dado
Variaveis
        Numerico Num, i, r
Inicio
        Escrever "Digite o número"
        Ler Num
        i = 2
        r = Raiz(Num, 2)
        Faca enquanto (Resto(Num, i) <> 0) e  (i <= r)
                i = i + 1
        Fim_enquanto
        Se (i <= r) entao
                Escrever i, " é divisor de ", Num
        Senao
                Escrever Num, " não tem divisores próprios"
        Fim_se
Fim
```

Um número inteiro que não tem divisores próprios é chamado *número primo*. Assim, o programa acima verifica se um dado número é primo. No próximo capítulo apresentaremos uma *função* que verifica especificamente se um inteiro dado é primo.

Uma outra aplicação importante do comando *Faca enquanto* diz respeito a aplicações sucessivas de um programa. O leitor deve ter observado que os programas anteriores são executados apenas para uma entrada. Se quisermos a sua execução para outra entrada precisamos começar tudo de novo. Por exemplo, se o programa anterior foi gravado na unidade C com o nome *Divis.ila* e quisermos obter um divisor próprio de 5 111 teríamos de, a partir do *prompt do DOS*, comandar

```
C:\>Ila  Divis
```

O programa começaria a ser executado e a tela ficaria com a seguinte configuração.

```
C:\>Ila  Divis
Digite o número
```

Neste momento o número 5 111 deveria ser digitado e então o resto do programa seria

executado, ficando agora a tela assim:

```
C:\>Ila  Divis
Digite o número
5111
19 é divisor de 5111
```

Aí a execução do programa é encerrada e, novamente, aparece o *prompt do DOS*, ficando a tela da seguinte forma:

```
C:\>Ila  Divis
Digite o número
5111
19 é divisor de 5111
C:\>
```

Se quiséssemos agora um divisor próprio de 5 501, teríamos de repetir a execução do programa, digitando novamente

```
C:\>Ila  Divis
```

Pode-se repetir a execução de um programa quantas vezes se queira, colocando-o numa estrutura definida por um comando *Faca enquanto* controlada pelo valor da entrada ou por uma pergunta ao usuário do tipo "deseja continuar?". No primeiro caso, o valor que encerra a execução pode ser informado dentro da mensagem que indica a necessidade da digitação da entrada e no segundo caso utiliza-se uma variável do tipo caractere para "responder" a pergunta citada acima. O programa anterior poderia ser então escrito de uma das seguintes formas.

```
//Programa que determina um divisor próprio de um inteiro positivo dado
Variaveis
        Numerico Num, i, r
Inicio
        Num = 1
        Faca enquanto Num <> 0
                Escrever "Digite o número (zero para encerrar)"
                Ler Num
                i = 2
                r = Raiz(Num, 2)
                Faca enquanto (Resto(Num, i) <> 0) e  (i <= r)
                        i = i + 1
                Fim_enquanto
                Se (i <= r) entao
                        Escrever i, " é divisor de ", Num
                Senao
                        Escrever Num, " não tem divisores próprios"
                Fim_se
        Fim_enquanto
Fim
```

ou

```
//Programa que determina um divisor próprio de um inteiro positivo
Variaveis
        Numerico Num, i, r
        Caracter Resp
Inicio
        Resp = "S"
```

```
        Faca enquanto (Resp = "S") ou (Resp = "s")
                Escrever "Digite o número"
                Ler Num
                i = 2
                r = Raiz(Num, 2)
                Faca enquanto (Resto(Num, i) <> 0) e  (i <= r)
                        i = i + 1
                Fim_enquanto
                Se (i <= r) entao
                        Escrever i, " é divisor de ", Num
                Senao
                        Escrever Num, " é primo "
                Fim_se
                Escrever "Deseja continuar (S/N)?
                Ler Resp
        Fim_enquanto
Fim
```

Uma outra aplicação interessante do comando *Faca enquanto* é na verificação da *consistência dos dados de entrada*. Esta ação consiste em que um programa deve recusar dados incompatíveis com a entrada do programa. Por exemplo, se a entrada é o número correspondente a um mês do ano, este número não pode ser menor do que 1 nem maior do que 12. A consistência dos dados pode ser feita colocando-se o comando de entrada *Ler* dentro de um comando *Faca enquanto* controlado pela expressão lógica que garante a consistência esperada. No exemplo do número correspondente a um mês do ano poderíamos ter o seguinte trecho (considerando definida uma variável do tipo *Numerico* Mes):

```
Escrever "Digite o número correspondente ao mês"
Ler Mes
Faca enquanto (Mes < 1) ou (Mes > 12)
        Se (Mes < 1) ou (Mes > 12) então
                Escrever "Mês inválido (digite o mês corretamente")
        Fim_se
        Ler Mes
Fim_enquanto
```

4.4 Exemplos Parte III

1. Consideremos um programa para determinar a soma dos *n* primeiros números ímpares, *n* dado. Por exemplo, se for fornecido para *n* o valor 6, o programa deve retornar 36, pois 1 + 3 + 5 + 7 + 9 + 11 = 36. Naturalmente, o sistema pode gerar os números ímpares que se pretende somar, através do comando Impar = 1 e da repetição do comando Impar = Impar + 2. Naturalmente, também, para que o sistema gere o próximo ímpar, o anterior já deve ter sido somado. Isto pode ser feito através do comando Soma = 0 e da repetição do comando Soma = Soma + Impar. Temos então o seguinte programa.

```
//Programa que determina a soma dos n primeiros números ímpares
Variaveis
        Numerico n, i, Impar, Soma
Inicio
        n = 1
        Faca enquanto n <> 0
                Escrever "Digite a quantidade ímpares (zero para encerrar)"
```

```
            Ler n
            Impar = 1
            Soma = 0
            Para i = 1 ate n
                  Soma = Soma + Impar
                  Impar = Impar + 2
            Proximo
            Se n <> 0 entao
                  Escrever "A soma dos ", n, " primeiros números ímpares é ", Soma
            Fim_se
      Fim_enquanto
Fim
```

Observe que os comandos Impar = 1 e Soma = 0 atribuem um valor inicial às variáveis para que estes valores iniciais possam ser utilizados nas primeiras execuções dos comandos Soma = Soma + Impar e Impar = Impar + 2. Normalmente, nos referimos a comandos que atribuem valores iniciais a variáveis para que estes valores possam ser utilizados na primeira execução de um comando que terá sua execução repetida como *inicialização da variável.*

Uma outra observação interessante é que, como existe uma fórmula que dá o i-ésimo número ímpar ($a_i = 2i - 1$), o programa acima poderia ser escrito de uma forma mais elegante, prescindindo, inclusive, da variável ímpar.

```
//Programa que determina a soma dos n primeiros números ímpares
Variaveis
      Numerico n, i, Soma
Inicio
      Escrever "Digite a quantidade ímpares (zero para encerrar)"
      n = 1
      Faca enquanto n <> 0
            Ler n
            Soma = 0
            Para i = 1 ate n
                  Soma = Soma + 2*i - 1
            Proximo
            Se n <> 0 entao
                  Escrever "A soma dos ", n, " primeiros números ímpares é ", Soma
            Fim_se
      Fim_enquanto
Fim
```

Optamos por apresentar a primeira versão pelo fato de que nem sempre a fórmula para gerar os termos da sequência que se pretende somar é tão simples ou é muito conhecida. Por exemplo, o exercício número 2 da seção 4.5 pede para somar os quadrados dos n primeiros números naturais e, neste caso, embora a fórmula exista, ela não é tão conhecida.

2. Um dos exemplos da seção anterior apresentava um programa que determinava, se existisse, um divisor próprio de um n inteiro dado. Imaginemos agora que queiramos um programa que apresente a lista de todos os divisores de um inteiro n dado. Neste caso, o programa pode percorrer todos os inteiros de um até a metade de n verificando se cada um deles é um seu divisor. Temos então o seguinte programa.

```
//Programa que lista os divisores de um inteiro dado
Variaveis
      Numerico n, i, Metade
Inicio
```

```
        Escrever "Digite o número (zero para encerrar)"
        n = 1
        Faca enquanto n <> 0
                Ler n
                Metade = Inteiro(n/2)
                Escrever "Os divisores de ", n, " são "
                Para i = 1 ate Metade
                        Se Resto(n, i) = 0 entao
                                Escrever i
                        Fim_se
                Proximo
                Escrever n
        Fim_enquanto
Fim
```

Vale observar que, ao contrário do que foi dito na seção 2.9, os valores de saída deste programa não estão sendo armazenados. O que acontece é que ainda não temos condições de armazenar uma quantidade indefinida de elementos. Este problema será resolvido no capítulo 6. Vale observar também que, como a variável de controle do comando *Para próximo* pode receber valores fracionários, o cálculo de Inteiro(n/2) é desnecessário, podendo-se usar diretamente Metade = n/2. A função inteiro foi colocada ali pelo fato de que a maioria dos sistemas faz diferença entre números inteiros e números fracionários e, em alguns casos, a variável de controle de uma estrutura do tipo P*ara próximo* só pode receber valores inteiros.

3. Na seção 1.5 discutimos um algoritmo que determinava o quociente e o resto da divisão entre dois inteiros positivos dados. Embora o ILA possua uma função pré-definida que calcula o resto de uma divisão inteira e já mostramos como calcular o quociente, utilizando esta função, vamos apresentar a implementação do algoritmo referido.

```
//Programa que retorna o quociente e o resto da divisão de dois inteiros dados
Variaveis
        Numerico Dividendo, Divisor, Rest, Quoc
        Caracter r
Inicio
        r = "S"
        Faca enquanto (r = "S") ou (r = "s")
                Escrever "Digite o dividendo e o divisor"
                Ler Dividendo
                Ler Divisor
                Faca enquanto Divisor = 0
                        Se Divisor = 0 entao
                                Escrever "O divisor tem que ser não nulo"
                        Fim_se
                        Ler Divisor
                Fim_enquanto
                Quoc = 1
                Faca enquanto Quoc*Divisor <= Dividendo
                        Quoc = Quoc + 1
                Fim_enquanto
                Quoc = Quoc - 1
                Rest = Dividendo - Quoc*Divisor
                Escrever "O quociente e o resto da divisão de ", Dividendo, " por ", Divisor, " são
", Quoc, " e ", Rest
                Escrever "Deseja continuar (S/N)?"
```

```
            Ler r
        Fim_enquanto
Fim
```

4. Em muitos casos há necessidade de que um dos comandos da sequência que terá sua execução repetida através de uma estrutura de repetição seja uma outra estrutura de repetição, caso em que dizemos que as estruturas estão *aninhadas*. Por exemplo, seja um programa que pretenda exibir a lista de todos os *pares ordenados* cujas componentes pertençam ao conjunto {1, 2, 3, 4, 5}. Observe que, para isto, para cada valor da primeira componente o programa deve gerar todas as segundas componentes. Portanto, devemos ter uma estrutura de repetição para gerar as primeiras componentes e uma outra, vinculada a cada valor da primeira componente, para gerar as segundas.

```
//Programa para gerar os pares ordenados de elementos de um conjunto
Variaveis
        Numerico i, j
Inicio
        Para i = 1 ate 5
                Para j = 1 ate 5
                        Escrever "(",i,", ",j,")"
                Proximo
        Proximo
Fim
```

5. É interessante observar que a variável de controle da estrutura *interna* pode depender da variável de controle da estrutura *externa*. Por exemplo, se ao invés dos pares ordenados, quiséssemos os subconjuntos do conjunto {1, 2, 3, 4, 5, 6} com dois elementos, o programa não deveria exibir o subconjunto, por exemplo, {1, 1}, que possui um só elemento, e deveria exibir apenas um dos subconjuntos {1, 2} e {2, 1} já que eles são iguais. Isto pode ser obtido, inicializando j com uma unidade maior do que o valor de i.

```
Variaveis
        Numerico i, j
Inicio
        Para i = 1 ate 5
                Para j = i + 1 ate 6
                        Escrever "{",i,", ",j,"}"
                Proximo
        Proximo
Fim
```

6. Seja um programa para o cálculo da média de uma dada quantidade de números. Na seção 1.5 discutimos um algoritmo para determinar a média de 10 000 números dados. Na ocasião discutimos que utilizaríamos uma única variável para receber os números sendo que um valor subsequente só seria solicitado depois que o anterior fosse "processado". A diferença agora é que a quantidade de números será um dado de entrada, o que torna o programa de aplicação mais variada. Como a quantidade de números será dada, pode-se utilizar uma estrutura *Para próximo* para receber e somar os números.

```
//Programa que determina a média aritmética de n números dados
Variaveis
        Numerico Num, Soma Media, n, i
Inicio
        Escrever “Digite a quantidade de números”
        Ler n
        Soma = 0
```

```
        Escrever “Digite os números”
        Para i = 1 até n
                Ler Num
                Soma = Soma + Num
        Proximo
        Media = Soma/n
        Escrever “Média dos números dados: “, Media
Fim
```

7. O exemplo acima tem o inconveniente de que sua execução exige que se saiba anteriormente a quantidade de números. Naturalmente, isto não ocorre na maioria dos casos. Vejamos então um programa para determinar a média de uma relação de números dados, sem que se conheça previamente a quantidade deles. Neste caso, não podemos utilizar o comando *Para proximo*, pois não sabemos o número de repetições! Assim, o comando *Faca enquanto* deve ser utilizado e uma pergunta deve ser formulada: qual a expressão lógica que controlará a estrutura? A solução é "acrescentar" à relação um valor sabidamente diferente dos valores da relação e utilizar este valor para controlar a repetição. Este valor aqui referido é conhecido como *flag*. Como dito logo acima, deve-se ter certeza que o *flag* não consta da relação. Isto não é complicado, pois ao se escrever um programa se tem conhecimento de que valores o programa vai manipular e a escolha do *flag* fica facilitada. Por exemplo, se o programa vai manipular números positivos pode-se usar -1 para o *flag*. Além do *flag*, o programa necessita de uma variável que determine a quantidade de números da relação, valor este que será utilizado no cálculo da média.

```
//Programa que determina a média aritmética de uma relação de números dados
Variaveis
        Numerico Num, Soma, Media, Quant
Inicio
        Soma = 0
        Quant = 1
        Escrever "Digite os números (-1 para encerrar)"
        Ler Num
        Faca enquanto Num <> -1
                Soma = Soma + Num
                Quant = Quant + 1
                Ler Num
        Fim_enquanto
        Quant = Quant - 1
        Media = Soma/Quant
        Escrever "Média dos números dados: ", Media
Fim
```

Vale observar que o comando Quant = Quant - 1 é necessário pelo fato de que o *flag* foi computado como elemento da relação. Isto poderia ser evitado, inicializando-se a variável Quant com zero. Porém, quando estivermos no capítulo 6 estudando *vetores* a inicialização aqui adotada será importante.

8. Na seção 1.6 apresentamos o *algoritmo de Euclides* para a determinação do *máximo divisor comum* de dois números dados. Para relembrar, vejamos como calcular o máximo divisor comum de 204 e 84.

204	84	36	12	0
	2	2	3	

O algoritmo é o seguinte: divide-se 204 por 84 obtendo-se resto 36; a partir daí repete-se divisões até que o resto seja zero, sendo o dividendo da divisão atual o divisor da divisão anterior e

o divisor da divisão atual o resto da divisão anterior. O último divisor é o máximo divisor procurado.

Escrever este algoritmo numa linguagem de programação é muito simples, pois uma estrutura de repetição e comandos de atribuição permitem que se obtenha facilmente a sequência de divisões desejadas.

```
//Programa para a determinação do máximo divisor comum de dois inteiros positivos
Variaveis
    Numerico x, y, m, n, Mdc, Rest
Inicio
    Escrever "Digite os dois números"
    Ler m
    Ler n
    x = m
    y = n
    Rest = Resto(x, y)
    Faca enquanto Rest <> 0
        x = y
        y = Rest
        Rest = Resto(x, y)
    Fim_enquanto
    Mdc = y
    Escrever "Mdc(",m,", ", n,") = ", Mdc
Fim
```

Note a necessidade da utilização das variáveis *x* e *y*. Elas são variáveis utilizadas no processamento e terão os seus conteúdos alterados durante a execução do programa. Se usássemos as variáveis m e n, os valores dos dados de entrada seriam perdidos, o que, evidentemente, não deve ocorrer.

À primeira vista, o programa deveria inicialmente determinar o maior dos números m e n, armazenando-o em x. O quadro seguinte mostra que isto não é necessário, apresentando a simulação da execução do programa para m = 68 e n = 148.

m	n	x	y	Rest	Mdc
68	148	68	148	68	
		148	68	12	
		68	12	8	
		12	8	4	
		8	4	0	
					4

9. Um outro algoritmo matemático cuja implementação em linguagem de programação apresenta um bom exemplo do uso de estruturas de repetição é o algoritmo para a determinação do *mínimo múltiplo comum* (mmc) de dois números dados. Como indica a própria denominação, o *mínimo múltiplo comum* de dois números é o menor número que é divisível pelos dois números. A matemática prova que o *mmc* de dois números é o produto dos divisores primos dos dois números, comuns ou não, ambos com as suas multiplicidades. Para se obter os divisores primos, realiza-se divisões sucessivas pelos primos que são divisores de pelo menos um dos números.

O quadro da página seguinte mostra o cálculo do mínimo múltiplo comum dos números 360 e 420, como nos é ensinado no ensino fundamental.

360,	420	2
180,	210	2
90,	105	2
45,	105	3
15,	35	3
5,	35	5
1,	7	7
1,	1	MMC = 2∗2∗2∗3∗3∗5∗7 = 2 520

Observe que, quando um divisor primo é encontrado, repete-se a divisão, com o quociente no lugar do dividendo, até se obter um número que não é múltiplo daquele divisor. Aí, incrementa-se o tal divisor. Isto é feito até que ambos os quocientes sejam iguais a 1. Temos o seguinte programa.

```
//Programa para determinação do mínimo múltiplo comum de dois inteiros positivos
Variaveis
    Numerico m, n, x, y,  Div, Mmc
Inicio
    Escrever "Digite os dois números"
    Ler m
    Ler n
    x = m
    y = n
    Mmc = 1
    Div = 2
    Faca enquanto (x <> 1) ou (y <> 1)
        Faca enquanto (Resto(x, Div) = 0) ou (Resto(y, Div) = 0)
            Mmc = Mmc*Div
            Se Resto(x, Div) = 0 entao
                x = Inteiro(x/Div)
            Fim_se
            Se Resto(y, Div) = 0 entao
                y = Inteiro(y/Div)
            Fim_se
        Fim_enquanto
        Div = Div + 1
    Fim_enquanto
    Escrever "mmc(", m,", ", n,") = ", Mmc
Fim
```

10. A questão do *mmc* é muito interessante como exemplo para a aprendizagem de programação pelo fato de que podemos apresentar um outro algoritmo de compreensão bem mais simples que o anterior.

A idéia é a seguinte: se formos multiplicando m sucessivamente por 1, 2, 3,..., obteremos múltiplos de *m*. Para se obter o mínimo múltiplo comum basta que se tome o primeiro destes produtos que seja múltiplo também de *n*.

```
//Programa para a determinação do mínimo múltiplo comum de dois inteiros positivos
Variaveis
    Numerico m, n, x, y,  i, Mmc
Inicio
    Escrever "Digite os dois números"
    Ler m
    Ler n
    Se m > n entao
```

```
            x = m
            y = n
        Senao
            x = n
            y = m
        Fim_se
        i = 1
        Faca enquanto Resto(x*i, y) <> 0
            i = i + 1
        Fim_enquanto
        Mmc = x*i
        Escrever "mmc(", m,", ", n,") = ", Mmc
Fim
```

4.5 Exercícios propostos

1. Mostre a configuração da tela após a execução do programa

```
Variaveis
    Numerico i, a, q, Termo
Inicio
    Para i = 5 ate 1 passo -1
        a = i
        q = 3
        Termo = a
        Faca enquanto Termo <= 9*a
            Escrever Termo
            Termo = Termo*q
        Fim_enquanto
    Proximo
Fim
```

2. Escreva um programa que determine a soma dos quadrados dos n primeiros números naturais, n dado.

3. Escreva um programa para calcular a soma dos n primeiros termos das sequências abaixo, n dado.

a) $\left(\frac{1}{2}, \frac{3}{5}, \frac{5}{8}, \ldots\right)$

b) $\left(1, -\frac{1}{2}, \frac{1}{3}, -\frac{1}{4}, \ldots\right)$

4. Escreva um programa que gere uma tabela de conversão de temperaturas em graus Fahrenheit para graus Celsius, com as temperaturas em graus Fahrenheit cobrindo a faixa de 0 a 150 graus, com variação de meio grau.

5. Um número inteiro é dito *perfeito* se ele é igual à metade da soma de todos os seus divisores. Por exemplo, como os divisores de 6 são 1, 2, 3 e 6 e 1 + 2 + 3 + 6 = 12, 6 é perfeito. A Matemática ainda não sabe se a quantidade de números perfeitos é ou não finita. Escreva um programa que liste todos os números perfeitos menores que um inteiro n dado.

6. O número 3 025 possui a seguinte característica: 30 + 25 = 55 e 55^2 = 3 025. Escreva um programa que escreva todos os números com quatro algarismos que possuem a citada característica.

7. Escreva um programa que escreva todos os pares de números de dois algarismos que apresentam a seguinte propriedade: o produto dos números não se altera se os dígitos são invertidos. Por exemplo, 93x13 = 39x31 = 1 209.

8. Escreva um programa para determinar o número de algarismos de um número inteiro positivo dado.

9. Escreva um programa que verifique se um dado número inteiro positivo é o produto de dois números primos. Por exemplo, 15 é o produto de dois primos, pois 15 = 3 x 5; 20 não é o produto de dois primos, pois 20 = 2 x 10 e 10 não é primo.

10. Quando um número não é produto de dois números primos, a matemática prova que ele pode ser escrito de maneira única como um produto de potências de números primos distintos. Este produto é chamado de *decomposição em fatores primos* do número e os expoentes são chamados de *multiplicidade* do primo respectivo. Por exemplo, $360 = 2^3x3^2x5$. Escreva um programa que obtenha a decomposição em fatores primos de um inteiro dado.

11. Escreva um programa que transforme o computador numa *urna eletrônica* para eleição, em segundo turno, para presidente de um certo país, às quais concorrem os candidatos 83-Alibabá e 93-Alcapone. Cada voto deve ser dado pelo número do candidato, permitindo-se ainda os votos 00 para *voto em branco* e 22 para *voto nulo*, devendo o eleitor ser consultado quanto à confirmação do seu voto. No final da eleição o programa deve emitir um relatório contendo a votação de cada candidato, a quantidade votos em branco, a quantidade de votos nulos e o candidato eleito.

12. A *sequência de Fibbonaci* é a sequência (1, 1, 2, 3, 5, 8, 13, ...) definida por

$$a_n = \begin{cases} 1, n = 1 \, ou \, n = 2 \\ a_{n-1} + a_{n-2}, se \, n > 2 \end{cases}$$

Escreva um programa que determine o n-ésimo termo desta sequência, n dado.

13. Os sistemas de computação que gerenciam caixas de lojas e supermercados fornecem ao operador, após a informação do valor do pagamento, o troco, em números decimais, que ele deve dar ao cliente. Talvez fosse interessante que, para otimizar a utilização das notas e das moedas de menor valor, visando minimizar o problema da "falta de troco", o sistema fornecesse ao operador as quantidades de cada nota e de cada moeda para um "troco ótimo". Admitindo que o supermercado forneça também troco para pagamentos em cheque de qualquer valor, escreva um programa que, recebendo o valor da compra e o valor do pagamento, forneça o "troco ótimo" no sentido comentado acima.

14. A *série harmônica* $S = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} + \ldots$ é *divergente*. Isto significa que dado qualquer real k existe n_0 tal que $1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n_0} > k$. Escreva um programa que dado um real k determine o menor inteiro n_0 tal que S > k. Por exemplo se k = 2, o programa deve fornecer $n_0 = 4$, pois $1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} = 2,083\ldots$ e $1 + \frac{1}{2} + \frac{1}{3} = 1,8333\ldots$

15. Escreva um programa que escreva todos os subconjuntos com três elementos de um conjunto que tem *n* elementos, *n* dado.

16. Dois números inteiros são ditos ***amigos*** se a soma dos divisores de cada um deles (menores que eles) é igual ao outro. Por exemplo, os divisores de 220 são 1, 2, 4, 5, 10, 11, 20, 22, 44, 55 e 110 e 1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 = 284 e os divisores de 284 são 1, 2, 4, 71 e 142 e 1 + 2 + 4 + 71 + 142 = 220. Escreva um programa que determine todos os pares de inteiros amigos menores que um inteiro dado.

17. Um inteiro positivo x é dito uma ***potência prima*** se existem dois inteiros positivos p e k, com p primo, tais que $x = p^k$. Escreva uma função que receba um inteiro e verifique se ele é uma potência prima.

18. Um inteiro positivo x é dito uma ***potência perfeita de base z e expoente y*** se existem dois inteiros positivos z e y tais que $x = z^y$. Escreva uma função que receba um inteiro e verifique se ele é uma potência perfeita.

Observação

Para obter o caderno das soluções dos exercícios propostos basta encaminhar e-mail para jaime@ccen.ufal com assunto RESPOSTAS LIVRO ILA e contendo NOME, CATEGORIA (docente, discente, autodidata), CIDADE/ESATADO e, se for o caso, INSTITUIÇÃO/CURSO.

5. Funções

5.1 Introdução

O exercício 7 da seção 3.5 solicitava que fosse feito um programa para a determinação das raízes reais ou complexas de uma equação do segundo grau. Com os conhecimentos adquiridos até a referida seção, um possível programa seria o seguinte.

```
//Programa para a determinação das raízes de uma equação do segundo grau
Variaveis
    Numerico a, b, c, x1, x2, Delta, Real, Imag
Inicio
    Escrever "Digite os coeficientes"
    Ler a
    Ler b
    Ler c
    Se a <> 0 entao
        Real = -b/(2*a)
        Delta = b^2 - 4*a*c
        Se Delta >= 0 entao
            Imag = Raiz(Delta, 2)/(2*a)
            x1 = Real + Imag
            x2 = real - Imag
            Escrever "Raízes: ", x1, " e ", x2
        Senao
            Imag = Raiz(-Delta, 2)/(2*a)
            Escrever "Raízes: ", Real, " + ",Imag,"i e ", Real, " - ",Imag,"i"
        Fim_se
    Senao
        Escrever "A equação não e do segundo grau"
    Fim_se
Fim
```

Observe que os comandos Imag = Raiz(Delta, 2)/(2*a) e Imag = Raiz(-Delta, 2)/(2*a) são basicamente idênticos e são necessários para que o radicando seja positivo. O sistema ILA permite que uma ação deste tipo possa ser realizada "à parte do programa" e que este "chame" a execução desta ação quando necessário. Estas ações que são executadas "fora" do programa propriamente dito são realizadas através de *funções* que, como dito na seção 2.5, devem ser definidas logo após às definições das variáveis com a seguinte sintaxe:

```
Funcao Identificador(Lista de parâmetros)
Inicio
    //Sequência de comandos
Fim
```

Nesta definição *Lista de parâmetros* é um conjunto de variáveis (e, portanto, devem ser declaradas na área para tal) que receberão os valores para os quais a execução da função é solicitada, atuando como se fosse a "entrada" da função. Estes valores estes que são chamados *argumentos* da execução da função Como uma entrada de um programa, a lista de parâmetros de uma função pode ser vazia.

A execução de uma função pode ser solicitada com a colocação do identificador da função (e a lista dos *argumentos*) dentro de uma expressão ou numa linha do programa, como se faz com um comando. Para que uma função possa ser *ativada* (ou seja, tenha a sua execução solicitada) dentro de uma expressão é necessário que um dos seus comandos seja um comando cuja sintaxe é

Retornar Expressão

sendo o valor de *Expressão* utilizado para a avaliação da expressão que contém a *chamada* da função.

5.2 Exemplos Parte IV

1. Com o uso de funções, o programa da equação do segundo grau poderia ser melhorado com a utilização de uma função que calculasse o valor absoluto de Delta, permitindo que fosse calculada "a parte imaginária" da raiz, quer Delta seja positivo ou negativo.

```
//Programa para determinação das raízes de uma equação do segundo grau.
Variaveis
    Numerico a, b, c, x1, x2, Delta, Real, Imag, z, x
    //Função que retorna o valor absoluto de um número
    Funcao Abs(x)
    Inicio
        Se x < 0 entao
            Retornar -x
        Senao
            Retornar x
        Fim_se
    Fim
Inicio //Programa principal
    Escrever "Digite os coeficientes"
    Ler a
    Ler b
    Ler c
    Se a <> 0 entao
        Real = -b/(2*a)
        Delta = b^2 - 4*a*c
        z = Abs(Delta)
        Imag = Raiz(z, 2)/(2*a)
        Se Delta >= 0 entao
            x1 = Real + Imag
            x2 = real - Imag
            Escrever "Raízes: ", x1, " e ", x2
        Senao
            Escrever "Raízes: ", Real, " + ",Imag,"i e ", Real, " - ",Imag,"i"
        Fim_se
    Senao
        Escrever "A equação não é do segundo grau"
    Fim_se
Fim
```

2. De um modo geral, os sistemas de computação não trabalham com números racionais na forma de fração ordinária. A manipulação de frações ordinárias é feita considerando-se separadamente os termos da fração. Um programa que pretendesse simplificar uma fração ordinária poderia ter uma função que retornasse o máximo divisor comum dos termos da fração e a simplificação poderia ser feita dividindo-se os termos da fração por este máximo divisor comum.

```
//Programa para simplificar frações ordinárias
Variaveis
    Numerico Num, Den, x, y, r, Mdc, NovoNum, NovoDen
```

```
        //Função que retorna o máximo divisor comum de dois números dados
        Funcao MaxDivCom(x, y)
        Inicio
                r = Resto(x, y)
                Faca enquanto r <> 0
                        x = y
                        y = r
                        r = Resto(x, y)
                Fim_enquanto
                Retornar y
        Fim
Inicio //Programa principal
        Escrever "Digite os termos da fração"
        Ler Num
        Ler Den
        Mdc = MaxDivCom(Num, Den)
        NovoNum = Num/Mdc
        NovoDen = Den/Mdc
        Escrever "A fração ", Num, "/", Den, " simplificada e: ", NovoNum, "/", NovoDen
Fim
```

3. Um programa para listar todos os números primos menores do que um inteiro dado poderia ter uma *função lógica* (ou seja, uma função que retorna um valor *falso* ou um valor *verdadeiro*) que recebendo como argumento um número inteiro verificasse se este inteiro é ou não primo.

```
//Programa para listar todos os primos menores que um inteiro positivo dado
Variaveis
        Numerico Num, x, i, j
        //Função lógica que verifica se um número inteiro é primo
        Funcao Primo(x)
        Inicio
                i = 2
                Faca enquanto (Resto(x, i) <> 0) e (i < x/2)
                        i = i + 1
                Fim_enquanto
                Se (Resto(x, i) = 0) e (x <> 2) entao
                        Retornar falso
                Senao
                        Retornar verdadeiro
                Fim_se
        Fim
Inicio //Programa principal
        Escrever "Digite o número"
        Ler Num
        Escrever "Os primos menores que ", Num," são: "
        Para j = 2 ate Num - 1
                Se Primo(j) = verdadeiro entao
                        Escrever j
                Fim_se
        Proximo
Fim
```

4. Se quisermos um programa que gere uma tabela de *fatoriais*, podemos escrever uma função

que recebendo um inteiro positivo retorna o fatorial deste inteiro. Lembrando que n! = 1 . 2 . ... . n e que 0! = 1, a função deve inicializar uma variável f com o valor 1 e numa estrutura de repetição de i = 1 até n calcular f = f*i. Ao fim da estrutura de repetição f armazenará o fatorial procurado.

```
//Programa que gera uma tabela de fatoriais
Variaveis
    Numerico i, n, j, f, Num
    //Funcao que retorna o fatorial de um inteiro dado
    Funcao Fat(n)
    Inicio
        f = 1
        Para i = 1 ate n
            f = f*i
        Proximo
        Retornar f
    Fim
Inicio //Programa principal
    Escrever "Digite o valor máximo da tabela"
    Ler Num
    Para j = 0 ate Num
        Escrever j,"! =  ", Fat(j)
    Proximo
Fim
```

5. O exemplo 7 da seção 2.11 apresentou um programa que fornecia a parte fracionária de um inteiro dado, programa este que utilizava a função pré-definida Inteiro. Uma função que retorna a parte fracionária de um número positivo, sem usar nenhuma função pré-definida, poderia encontrar o maior inteiro menor que o número dado e retornar a diferença entre este número dado e o maior inteiro determinado.

```
//Função que determina a parte fracionária de um número e não utiliza funções pré-definidas.
Variaveis
    Numerico Num, x, i
Funcao Frac(x)
Inicio
    i = 0
    Faca enquanto i <= x
        i = i + 1
    Fim_enquanto
        i = i - 1
    Retornar x - i
Fim
```

6. As funções dos exemplos acima retornam, através do comando *Retornar*, um valor e elas são ativadas dentro de uma expressão. Pode-se definir funções que executem alguma tarefa, mas não retorne nenhum valor específico. Neste caso, a função é ativada com a referência ao seu identificador seguido, se for o caso, da lista de argumentos. A construção de um *menu de opções* de um programa que realize diversas tarefas é um exemplo deste tipo de função. Por exemplo, um programa que gerencie o acervo de uma biblioteca poderia conter a seguinte função.

```
Funcao Menu()
Inicio
    Escrever "1-Cadastrar usuário"
    Escrever "2-Cadastrar livro"
```

```
        Escrever "3-Empréstimo"
        Escrever "4-Devolução"
        Escrever "5-Encerrar"
        Escrever “Digite sua opção”
        Ler Opcao
Fim
```

Neste caso, um dos primeiros comandos do programa principal seria a ativação da função através de

```
Menu()
```

como se escreve um comando. Observe que esta função também exemplifica uma função cuja lista de parâmetros é vazia.

7. A maior vantagem do uso de funções é a possibilidade de que um programa seja escrito em *módulos*, o que facilita a *legibilidade* do programa, a *manutenção* do programa (no sentido de que alterações no sistema são realizadas simplesmente alterando-se algumas funções) e permite que vários programadores desenvolvam um mesmo programa, cada um deles ficando responsável por módulos específicos. Para exemplificar, suponhamos que pretendemos desenvolver um *software matemático* para a manipulação algébrica de *números complexos*. Em ILA um programa como este poderia ter uma função que exibisse um menu de opções e funções para cada uma das tarefas que se pretende que o programa seja capaz de executar. Algo como

```
//Programa para álgebra dos números complexos
Variaveis
        Numerico a, b, c, d, r, i, x, y, Mod, Ang
        Caracter Opc
        Funcao Menu()
        Inicio
                Escrever "1-Módulo de um complexo"
                Escrever "2-Soma de dois complexos"
                Escrever "3-Produto de dois complexos"
                Escrever "4-Forma polar de um complexo"
                Escrever  "5-Encerra o programa"
                Escrever  "Digite sua opção"
                Ler Opc
        Fim
        Funcao LerDoisComplexos()
        Inicio
                Escrever "Digite os complexos"
                Ler a
                Ler b
                Ler c
                Ler d
        Fim
        Funcao Modulo(r, i)
        Inicio
                Retornar Raiz(a^2 + b^2, 2)
        Fim
        Funcao SomaComplexos(a, b, c, d)
        Inicio
                r = a + c
                i = b + d
        Fim
```

```
        Funcao MultiplicaComplexos(a, b, c, d)
        Inicio
                r = a*c - b*d
                i = a*d + b*c
        Fim
        Funcao FormaPolar(a, b)
        Inicio
                Mod = Modulo(a, b)
                Se a <> 0 entao
                        Ang = Atan(b/a)
                        Escrever a," + ",b , "i = ", Mod,".(cos(",Ang, ") + i.sen(", Ang,")"
                Senao
                        Escrever b , "i = ", Mod,".(cos(Pi/2) + i.sen(Pi/2)"
                Fim_se
        Fim
//Programa principal
Inicio
        Faca enquanto Opc <> "5"
        Menu()
        Faca caso
                Caso Opc = "1":
                        Escrever "Digite o complexo"
                        Ler a
                        Ler b
                        Escrever "|",a,"+",b,"i| = ", Modulo(a, b)
                Caso Opc = "2":
                        LerDoisComplexos()
                        SomaComplexos(a, b, c, d)
                        Escrever "(", a, " + ",b, "i) + (", c, " + ", d, "i) = ", r, " + ",i, "i"
                Caso Opc = "3":
                        LerDoisComplexos()
                        MultiplicaComplexos(a, b, c, d)
                        Escrever "(", a, " + ",b, "i) . (", c, " + ", d, "i) = ", r, " + ",i, "i"
                Caso Opc = "4":
                        Escrever "Digite o complexo"
                        Ler a
                        Ler b
                        FormaPolar(a, b)
                Fim_caso
        Fim_enquanto
Fim
```

Observe que este exemplo mostra que uma função pode chamar a ativação de outra função definida anteriormente, como a função FormaPolar chamou a função Modulo.

5.3 Recursividade

O exercício 12 da seção 4.5 definiu a *sequência de Fibbonaci* como sendo a sequência (a_n) definida por

$$a_n = \begin{cases} 1, n = 1 \, ou \, n = 2 \\ a_{n-1} + a_{n-2}, se \, n > 2 \end{cases}$$

Observe que o termo de ordem n é definido a partir de termos anteriores. Isto significa que para o cálculo de um determinado termo há necessidade de que se *recorra* a valores de todos os termos anteriores. Por exemplo, para a determinação de a_5 necessitamos conhecer a_4 e a_3; para a determinação destes dois, necessitamos conhecer a_2 e a_1. Uma definição com estas características é dita uma definição por *recorrência* ou uma definição *recursiva*.

Outro ente matemático que pode ser definido por recorrência é o *fatorial* de um número natural, objeto de discussão do exemplo 4 da seção anterior. Embora se possa definir fatorial de um natural n por n! = 1 . 2 . 3 . ... . n, é matematicamente mais elegante definir por

$$n! = \begin{cases} 1, n = 0 \; ou \; n = 1 \\ n \,.\, (n-1)!, se \; n > 1 \end{cases}$$

Por exemplo, 4! = 4 . 3! = 4 . (3 . 2!) = (4 . 3) . (2 . 1!) = 4 . 3 . 2 . 1 = 24.

Naturalmente, uma definição recursiva necessita conter uma condição que interrompa a *recorrência*. Esta condição é chamada *condição de escape*. No caso do fatorial a condição de escape é n = 0 ou n = 1. A expressão que realiza propriamente a recorrência pode ser chamada *expressão de recorrência*

O que é surpreendente é que a maioria dos sistemas para desenvolvimento de programas permite que sejam definidas funções recursivas praticamente da mesma maneira que elas são escritas em matemática. Como o objetivo do lLA é apenas a aprendizagem da lógica de programação, os pesquisadores que o desenvolveram não incluíram recursos sofisticados de recursão. O ILA permite apenas a implementação de funções recursivas quando a expressão de recorrência contém uma única referência àquilo que se está definindo. Este é o caso, por exemplo, da definição recursiva do fatorial, cuja implementação em ILA é, simplesmente,

```
// Funcao recursiva que calcula o fatorial de um numero natural
Funcao FatRec(n)
Inicio
        Se (n = 0) ou (n = 1) entao
                Retornar 1
        Senao
                Retornar  n*FatRec(n-1)
        Fim_se
Fim
```

Se esta função for ativada para n = 4, por exemplo, ela *retorna* 4*FatRec(3). Isto significa que a função é ativada novamente para n = 3, ficando a expressão 4*FatRec(3) na *pilha de recursão*, aguardando o retorno de FatRec(3). Esta ativação *retorna* 3*FatRec(2) e a expressão na pilha, agora 4*3*FatRec(2), fica aguardando o *retorno* de FatRec(2), que é a expressão 2*FatRec(1). A expressão na tal *pilha de recursão* fica agora 4*3*2*FatRec(1), aguardando o *retorno* de FatRec(1). Como pela definição da função, FatRec(1) retorna 1, a expressão 4*3*2*1 é finalmente calculada e o seu valor, 24, é retornado para o comando que ativou a função para n = 4.

Devido à necessidade do empilhamento das expressões que calculam a função recursiva e das sucessivas ativações da mesma função, é fácil perceber que a utilização de uma função recursiva demanda um maior tempo computacional do que uma função não recursiva (chamada, geralmente, *função iterativa*). Assim, sempre que possível, devemos optar pela solução iterativa. No capítulo 7, porém, apresentaremos um exemplo (não executável) onde a solução recursiva seria tão eficiente quanto a iterativa, sendo bem mais elegante.

5.4 Exercícios propostos

1. Escreva uma função que retorne o k-ésimo dígito (da direita para esquerda) de um inteiro n, k e n dados. Por exemplo, K_esimoDigito(2845, 3) = 8.

2. O *fatorial ímpar* de um número n ímpar positivo é o produto de todos os números ímpares positivos menores do que ou iguais a n. Indicando o *fatorial ímpar* de n por n| temos, n| = 1 . 3. 5 . ... . n. Por exemplo, 7| = 1 . 3 . 5 . 7 = 105. Escreva funções, iterativa e recursiva, para a determinação do fatorial ímpar de um inteiro ímpar dado.

3. Como na questão anterior, o *fatorial primo* ou *primorial* de um número primo positivo é o produto de todos os primos positivos menores do que ou iguais a ele: p# = 2 . 3 . 5 . 7 . ... . p. Por exemplo, 7# = 2 . 3 . 5 . 7 = 210. Escreva um programa que determine o primorial de um número primo dado.

4. Escreva funções, iterativa e recursiva, que retorne a soma dos algarismos de um inteiro positivo dado.

5. O exemplo 5 da seção 5.2, apresentou uma função que, sem usar funções pré-definidas, retornava a parte fracionária de um número positivo dado. Reescreva esta função sem a exigência da positividade do argumento.

6. Escreva uma função recursiva que determine o *mínimo múltiplo comum* de dois inteiros dados.

7. Escreva funções, recursiva e iterativa, que implementem o operador ^.

8. Escreva uma função recursiva que determine o *máximo divisor comum* de dois inteiros dados.

Observação

Para obter o caderno das soluções dos exercícios propostos basta encaminhar e-mail para jaime@ccen.ufal com assunto RESPOSTAS LIVRO ILA e contendo NOME, CATEGORIA (docente, discente, autodidata), CIDADE/ESATADO e, se for o caso, INSTITUIÇÃO/CURSO.

6. Variáveis compostas

6.1 Introdução

Nos exemplos 6 e 7 da seção 4.4 discutimos programas para a determinação da média de uma relação de números dados. Para tal, utilizamos uma variável simples para receber os números, sendo que cada vez que um número, a partir do segundo, era recebido o anterior era "perdido". Ou seja, a relação de números não era armazenada. Imagine que a relação fosse uma relação de notas escolares e além da média se quisesse também saber a quantidade de alunos que obtiveram nota acima da média ou uma outra medida estatística (*desvio padrão*, por exemplo) que dependesse da média. Neste caso, haveria a necessidade de que a relação fosse redigitada, o que, além da duplicidade do trabalho, facilitaria os erros de digitação. É importante então que exista uma "variável" capaz de armazenar vários valores simultaneamente de tal forma que se possa acessar cada um deles independentemente de se acessar os demais.

Um outro exemplo que justifica plenamente a necessidade de uma *variável composta* é o caso do exemplo 2 da seção 4.4. Lá queríamos a relação dos divisores de um inteiro dado e estes divisores eram apenas exibidos, não sendo armazenados, como recomendado na seção 2.7.

Uma *variável composta* é um conjunto de variáveis simples do tipo *Numerico*, identificadas pela concatenação de índices entre colchetes ao identificador da variável composta. Como as variáveis simples, as compostas também devem ser definidas no início do programa com a seguinte sintaxe:

Matriz numerico Identificador[Expr 1, Expr 2, ..., Expr n]

onde Expr 1, Expr 2, ..., Expr n são expressões numéricas cujo valor é um número inteiro. Comumente o valor de *n* é referido como sendo a *dimensão* da variável composta e cada variável simples da variável composta é chamada de *componente* da variável composta. Em relação aos valores de *n*, costuma-se denominar a variável composta das seguintes formas: quando $n = 1$, temos um *vetor*; quando $n = 2$, uma *matriz*; quando $n = 3$, uma *matriz tridimensional* e assim sucessivamente.

Para exemplificar um vetor, a declaração

Matriz numerico Vet[1000]

define um conjunto de mil variáveis do tipo numerico Vet[1], Vet[2], ..., Vet[1000] e podemos armazenar em Vet uma relação com até mil números. Já para exemplificar uma matriz, definição

Matriz numerico Mat[20, 30]

define um conjunto de 600 variáveis Mat[1, 1], ..., Mat[1, 30], Mat[2, 1], ..., Mat[2, 30], ..., Mat[20, 1], ..., Mat[20, 30]. Naturalmente, matrizes bidimensionais, como Mat acima, são utilizadas para armazenar *tabelas de dupla entrada*. Para um valor fixo de *i*, dizemos que Mat[i, 1], Mat[i, 2], ... constituem a *linha i* da matriz enquanto que para um valor fixo de *j* Mat[1, j], Mat[2, j], ... constituem uma *coluna j*. O número de linhas e o número de colunas constituem a *ordem* da matriz.

Uma limitação do sistema ILA para a manipulação de variáveis compostas é que as componentes destas variáveis não podem ser argumentos de um comando *Ler*. A solução é definir uma variável simples do tipo *Numerico* para ser passada para o comando *Ler* e em seguida usar um comando de atribuição para armazenar o valor digitado na componente pretendida. Se queremos armazenar um dado de entrada na primeira componente do vetor Vet, utilizamos uma variável Aux e a sequência de comandos

```
Ler Aux
Vet[1] = Aux
```

Feita esta observação, o armazenamento de uma relação de números num *vetor* depende do fato de que seja ou não conhecida a quantidade de números da relação. Se esta quantidade é

conhecida antecipadamente pode-se usar uma estrutura *Para proximo*:

```
Escrever "Digite os elementos da relação"
Para i = 1 ate n
        Ler Aux
        Vet[i] = Aux
Proximo
```

onde *n* é a quantidade de elementos da relação.

Se tal número não é conhecido, pode-se utilizar um *flag*, como foi discutido no exemplo 7 da seção 4.4, e uma estrutura *Faca enquanto*:

```
Escrever "Digite os números (-1 para encerrar)"
Ler Aux
Vet[1] = Aux
i = 1
Faca enquanto Vet[i] <> -1
        i = i + 1
        Ler Aux
        Vet[i] = Aux
Fim_enquanto
i = i - 1
```

Observe que, a partir daí, a variável *i* é importante para o resto do programa pois armazena a quantidade elementos da relação ou o *tamanho do vetor*. Obviamente, o comando i = i - 1 é para "retirar" do vetor o *flag*.

Como uma componente de um vetor pode ser argumento de um comando *Escrever*, a exibição dos elementos de uma relação de números armazenada num vetor é feita de maneira óbvia, bastando "percorrer" o vetor com uma estrutura *Para proximo*.

```
Para i = 1 ate Quant
        Escrever V[i]
Proximo
```

Para o armazenamento de uma matriz é comum se exigir o conhecimento prévio da sua ordem. Com isto pode-se utilizar duas estruturas *Para proximo* aninhadas, a externa para controlar as linhas e a interna para controlar as colunas.

```
Escrever "Digite a ordem da matriz"
Ler m
Ler n
Escrever "Digite (por linha) os elementos da matriz"
Para i = 1 ate m
        Para j = 1 ate n
                Ler Aux
                Mat[i, j] = Aux
        Proximo
Proximo
```

A exibição dos elementos de uma matriz também é feita com duas estruturas *Para proximo* aninhadas.

```
Para i = 1 ate m
        Para i = 1 ate n
                Escrever Mat[i, j]
        Proximo
Proximo
```

6.2 Exemplos Parte V

1. Um vetor pode ser gerado pelo próprio sistema. Por exemplo, o programa abaixo armazena num vetor os quadrados dos n primeiros números naturais, n dado.

```
Variaveis
        Numerico i, n
        Matriz Numerico Vet[100]
Inicio
        Escrever "Digite n"
        Ler n
        Para i = 1 ate n
                Vet[i] = i^2
        Proximo
        Escrever "Os quadrados dos ", n, "primeiros números naturais são "
        Para i = 1 ate n
                Escrever Vet[i]
        Proximo
Fim
```

2. Seja um programa para determinar a média das notas de uma avaliação de uma turma de uma escola e, em seguida, determinar quantos alunos obtiveram nota maior que a média. Podemos armazenar as notas num vetor, calcular a média destas notas e depois "percorrer" o vetor quantificando o número de componentes maiores que a média. Como foi dito na seção anterior, pode-se percorrer um vetor utilizando-se uma estrutura *Para proximo*.

```
//Programa para determinar a média de uma relação de números e a quantidade de números da relação maiores que a média.
Variaveis
        Numerico Aux, i, j, Media, Soma, Quant
        Matriz numerico Vet[50]
Inicio
        //Entrada dos dados e cálculo da média
        Escrever "Digite os elementos da relação (-1 para encerrar)"
        Ler Aux
        Vet[1] = Aux
        i = 1
        Soma = 0
        Faca enquanto Vet[i] <> -1
                Soma = Soma + Aux
                Ler Aux
                i = i + 1
                Vet[i] = Aux
        Fim_enquanto
        i = i - 1
        Media = Soma/i
        //Determinação do número de componentes maiores que a média
        Para j = 1 ate i
                Se Vet[j] > Media entao
                        Quant = Quant + 1
                Fim_se
        Proximo
        Escrever "Média da relação: ", Media, "; Maiores que a média: ", Quant
```

```
Fim
```

3. Para um programa que determine o maior elemento de uma relação armazenada num vetor pode-se usar o seguinte algoritmo. Supõe-se que o maior elemento é a primeira componente do vetor e, em seguida, percorre-se todo o vetor verificando se alguma componente é maior do que aquele valor que, até o momento, é o maior. Encontrando-se uma componente de valor maior, troca-se o maior valor.

```
//Programa para determinar a maior componente de um vetor
Variaveis
        Numerico Aux, i, j, Maior
        Matriz numerico Vet[50]
Inicio
        Escrever "Digite os elementos da relação (-1 para encerrar)"
        Ler Aux
        Vet[1] = Aux
        i = 1
        Faca enquanto Vet[i] <> -1
                Ler Aux
                i = i + 1
                Vet[i] = Aux
        Fim_enquanto
        i = i - 1
        Maior = Vet[1]
        Para j = 2 ate i
                Se Vet[j] > Maior entao
                        Maior = Vet[j]
                Fim_se
        Proximo
        Escrever "O maior elemento da relação é ", Maior
Fim
```

4. O programa a seguir exemplifica a possibilidade de que os índices das componentes de um vetor sejam dados através de expressões. O seu objetivo é obter uma relação de elementos "intercalando" os elementos de duas relações dadas. Ou seja, dadas duas relações de números, o programa deve gerar uma terceira relação onde os elementos de ordem ímpar são os elementos da primeira relação e os de ordem par os da segunda. Por exemplo, se as relações são (1, 3, 6, 12) e (2, 5, 10, 15) o programa deve gerar a relação (1, 2, 3, 5, 6, 10, 12, 15). Assim, se as relações dadas forem armazenadas nos vetores V e W e queremos gerar a terceira relação em Vet, deveremos ter Vet[1] = V[1], Vet[2] = W[1], Vet[3] = V[2], Vet[4] = W[2], e assim sucessivamente. Observe que se i é ímpar Vet[i] é a componente de V de ordem $\frac{i+1}{2}$ e se i é par Vet[i] é a componente de W de ordem $\frac{i}{2}$.

```
Variaveis
        Numerico Aux, i, n, m
        Matriz numerico V[250]
        Matriz numerico W[250]
        Matriz numerico Vet[500]
Inicio
        Escrever "Digite o número de elementos das relações"
        Ler n
        Escrever "Digite os elementos da primeira relação"
```

```
        Para i = 1 ate n
                Ler Aux
                V[i] = Aux
        Proximo
        Escrever "Digite os elementos da segunda  relação"
        Para i = 1 ate n
                Ler Aux
                W[i] = Aux
        Proximo
        m = 2*n
        Para i = 1 ate m
                Se Resto(i, 2) = 1 entao
                        Vet[i] = V[(i+2)/2]
                Senao
                        Vet[i] = W[i/2]
                Fim_se
        Proximo
        Para i = 1 ate m
                Escrever Vet[i]
        Proximo
Fim
```

Este exemplo mostra também que a declaração de variáveis compostas exige que cada uma delas seja definida isoladamente. O ILA não aceitaria a definição

```
Variaveis
        Matriz Numerico V[250], W[250], Vet[500]
```

5. Como no exemplo 1, o sistema pode gerar uma matriz. Para exemplificar isto, apresentaremos um programa que gera a *matriz identidade de ordem n*. Para um inteiro positivo dado, a *matriz identidade de ordem n* é a matriz $I_n = (i_{rs})$, de ordem nxn, dada por $i_{rs} = 1$, se r = s, e $i_{rs} = 0$, se $r \neq s$. Esta matriz é muito importante no estudo das matrizes sendo utilizada, por exemplo, para a determinação da *matriz inversa* de uma matriz inversível. Por exemplo, se *n* = 3, temos

$$I_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

```
//Programa para gerar a matriz unidade de ordem n
Variaveis
        Numerico k, j, n
        Matriz numerico I[30, 40]
Inicio
        Escrever "Digite a ordem da matriz unidade"
        Ler n
        Para k = 1 ate n
                Para j = 1 ate n
                        Se k = j entao
                                i[k,j] = 1
                        Senao
                                i[k, j] = 0
                        Fim_se
                Proximo
        Proximo
```

```
        Escrever "Matriz unidade de ordem ", n
        Para k = 1 ate n
                Para j = 1 ate n
                        Escrever i[k, j]
                Proximo
        Proximo
Fim
```

6. Quando, como no exemplo anterior, o número de linhas é igual ao número de colunas a matriz é dita *matriz quadrada*. Neste caso, os elementos de índices iguais constituem a *diagonal principal*. A soma dos elementos da diagonal principal de uma matriz quadrada é o *traço* da matriz. Como mais um exemplo de programas que manipulem matrizes, o programa abaixo determina o *traço* de uma matriz quadrada dada. Observe que para percorrer a diagonal principal não há necessidade de dupla estrutura *Para proximo*.

```
//Programa para determinar o traço de uma matriz quadrada
Variaveis
        Numerico Aux, i, j, n, Traco
        Matriz numerico A[30, 30]
Inicio
        Escrever "Digite a ordem da  matriz"
        Ler n
        Escrever "Digite os elementos da matriz"
        Para i = 1 ate n
                Para j = 1 ate n
                        Ler Aux
                        A[i,j] = Aux
                Proximo
        Proximo
        //Determinação do traço da matriz
        Traco = 0
        Para i = 1 ate n
                Traco = Traco + A[i, i]
        Proximo
        Escrever "O traço da matriz dada e ", Traco
Fim
```

7. Naturalmente, o armazenamento de uma matriz que possui alguma propriedade específica pode ser facilitado se esta propriedade for levada em conta no programa. É o caso, por exemplo, de uma *matriz simétrica* (uma matriz quadrada $A = (a_{ij})$ é dita *simétrica* se $a_{ij} = a_{ji}$, quaisquer que sejam i e j). Se Mat[10, 10] é uma variável do tipo *Matriz numerico*, para armazenar em Mat uma matriz simétrica basta digitar os elementos situados acima da diagonal principal e os elementos desta diagonal.

```
//Programa para armazenar e exibir uma matriz simétrica
Variaveis
        Numerico i, j,  n, Aux, x, y
        Matriz numerico Mat[10, 10]
Inicio
        Escrever "Digite a ordem da matriz"
        Ler n
        Escrever "Digite os elementos da matriz, acima da diagonal"
        Para i = 1 ate n
                Para j = i ate n
```

```
                    Ler Aux
                    Mat[i, j] = Aux
                    Mat[j, i] = Aux
                Proximo
            Proximo
        Fim
```

8. Como se determina a soma de duas matrizes somando-se os seus elementos de mesmos índices, um programa que receba duas matrizes e determine a soma delas é muito simples.

```
        Variaveis
            Numerico Aux, i, j, m1, n1, m2, n2
            Matriz numerico A[30, 40]
            Matriz numerico B[30, 40]
            Matriz numerico Soma[30, 40]
        Inicio
            Escrever "Digite a ordem da primeira matriz"
            Ler m1
            Ler n1
            Escrever "Digite os elementos da primeira matriz"
            Para i = 1 ate m1
                Para j = 1 ate n1
                    Ler Aux
                    A[i,j] = Aux
                Proximo
            Proximo
            Escrever "Digite a ordem da segunda matriz"
            Ler m2
            Ler n2
            Escrever "Digite os elementos da segunda matriz"
            Para i = 1 ate m2
                Para j = 1 ate n2
                    Ler Aux
                    B[i, j] = Aux
                Proximo
            Proximo
            //Determinação da matriz soma
            Se (m1 = m2) e (n1 = n2) entao
                Para i = 1 ate m1
                    Para j = 1 ate n1
                        Soma[i, j] = A[i, j] + B[i, j]
                    Proximo
                Proximo
                Para i = 1 ate m1
                    Para j = 1 ate n1
                        Escrever Soma[i, j]
                    Proximo
                Proximo
            Senao
                Escrever "A soma das matrizes não esta definida"
            Fim_se
        Fim
```

9. Já um programa para multiplicação de matrizes não é tão simples. Na verdade é um programa que é muito útil para o desenvolvimento da lógica de programação. Se $A = (a_{ij})_{mxn}$ e $B = (b_{ij})_{rxs}$, a matriz produto só está definida se n = r e, neste caso, se P = A . B, então $p_{ij} = a_{i1} . b_{1j} + a_{i2} . b_{2j} + ... + a_{in} . b_{rj}$.

Ou seja, o elemento ij da matriz produto é a soma dos produtos dos elementos da linha i da matriz A pelos elementos da coluna j da matriz B, o que exigirá uma terceira estrutura *Para proximo*.

```
Variaveis
    Numerico Aux, i, j, k, m1, n1, m2, n2
    Matriz numerico A[30, 40]
    Matriz numerico B[30, 40]
    Matriz numerico M[30, 40]
Inicio
    Escrever "Digite a ordem da primeira matriz"
    Ler m1
    Ler n1
    Escrever "Digite os elementos da primeira matriz"
    Para i = 1 ate m1
        Para j = 1 ate n1
            Ler Aux
            A[i,j] = Aux
        Proximo
    Proximo
    Escrever "Digite a ordem da segunda matriz"
    Ler m2
    Ler n2
    Escrever "Digite os elementos da segunda matriz"
    Para i = 1 ate m2
        Para j = 1 ate n2
            Ler Aux
            B[i,j] = Aux
        Proximo
    Proximo
    //Determinação da matriz produto
    Se n1 = m2 entao
        Para i = 1 ate m1
            Para j = 1 ate n2
                M[i, j] = 0
                Para k = 1 ate n1
                    M[i, j] = M[i, j] + A[i, k]*B[k, j]
                Proximo
            Proximo
        Proximo
        Para i = 1 ate m1
            Para j = 1 ate n2
                Escrever M[i, j]
            Proximo
        Proximo
    Senao
        Escrever "O produto das matrizes não esta definido"
    Fim_se
```

```
Fim
```

6.3 Formatação da saída/Interface com o usuário

Mesmo não fazendo da parte da lógica de programação, é interessante que quem está aprendendo a programar saiba que os sistemas para desenvolvimento de programas oferecem recursos para que a saída do programa seja exibida de forma elegante e agradável de se ver e para que o sistema ofereça ao usuário *interfaces* que facilitem a entrada de dados. Isto é obtido através de funções (ou, em alguns sistemas, *procedimentos*) pré-definidos, sendo que, quanto mais sofisticado seja o sistema mais recursos ele oferece. Os sistemas *visuais*, como Visual Basic e Delphi, são os melhores exemplos de sistemas que oferecem recursos espetaculares para a criação de *interfaces* elegantes entre o sistema e o usuário.

Quando um comando *Escrever* é executado no ILA, o seu argumento é exibido e o *cursor* (traço intermitente que indica a posição onde o próximo caractere a ser exibido, sê-lo-á) vai para a linha e para a coluna seguintes àquelas da posição anterior. Por exemplo, se o vetor (3, 5, 8, 9, 12) estiver armazenado na variável Vet, a estrutura

```
Para i = 1 ate 5
        Escrever Vet[i]
Proximo
```

exibirá na tela

```
3
 5
  8
   9
    12
```

Naturalmente, um vetor não deve ser exibido desta forma! O ILA possui o comando *Posicionar* que, como o seu nome indica, posiciona o cursor num ponto específico da tela do vídeo. Sua sintaxe é:

```
Posicionar Variável 1, Variável 2
```

onde os conteúdos das variáveis 1 e 2 devem se números inteiros, que indicarão, respectivamente, a linha e a coluna em que o cursor será posicionado.

No exemplo anterior, a sequência de comandos

```
Para j = 1 ate 5
        x = x + i
        Posicionar 10, x
        Se j = 1 entao
                Escrever "(", Vet[j], ","
        Senao
                Se j < 5 entao
                        Escrever Vet[j], ","
                Senao
                        Escrever Vet[j], ")"
                Fim_se
        Fim_se
Proximo
```

exibe na tela, exatamente, (3, 5, 8, 9, 12).

O caso de uma matriz é mais grave. Se a matriz

$$\begin{pmatrix} 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \\ 4 & 5 & 6 & 7 \end{pmatrix}$$

estiver armazenada na variável composta Mat, a estrutura

```
Para i = 1 ate 4
    Para j = 1 ate 3
        Escrever Mat[i, j]
    Proximo
Proximo
```

exibirá na tela os elementos da matriz de seguinte forma:

```
2
 3
  4
   5
    3
     4
      5
       6
        4
         5
          6
           7
```

o que, naturalmente, não é interessante, pois uma matriz só tem interesse se ela estiver escrita em linhas e colunas. Para exibir uma variável composta bidimensional em forma de matriz podemos usar a seguinte estrutura

```
y = 1
Para i = 1 ate m
    x = 1
    Para j = 1 ate n
        Posicionar y, x
        Escrever Mat[i, j]
        x = x + 4
    Proximo
    y = y + 1
Proximo
```

Para a formatação da saída e para a criação de interfaces, o ILA ainda disponibiliza um comando cuja execução limpa uma área da tela do vídeo. Este comando é ativado através da seguinte sintaxe:

```
Limpar Expr1, Expr2, Expr3, Expr4
```

onde Expr1, Expr2, Expr3 e Expr4 são expressões que resultam valores inteiros e indicam as coordenadas do canto esquerdo superior (Expr1, Expr2) e do canto direito inferior (Expr3, Expr4) da área a ser limpa. Se Expr1, Erpr2, Expr3, Expr4 forem omitidos, o sistema adota os valores padrões 1, 1, 24 e 80, que são as coordenadas do canto esquerdo superior e as do canto direito inferior da tela. Ou seja, o comando *Limpar* equivale a *Limpar* 1, 1, 24, 80. Aproveitando o ensejo, valores tomados como padrão por um sistema são chamados valores *default*.

Outra instrução útil para a formatação da saída é o comando *Janela* que desenha uma moldura numa posição da tela fixado pelo programador. Este comando é ativado através da seguinte sintaxe:

```
Janela Expr1, Expr2, Expr3, Expr4
```

onde, como no comando *Limpar*, Expr1, Expr2, Expr3 e Expr4 são expressões que resultam valores inteiros e indicam as coordenadas do canto esquerdo superior (Expr1, Expr2) e do canto direito inferior (Expr3, Expr4) da moldura.

Finalmente, o comando *Cor* que permite se fixar cores distintas para a "frente" do vídeo (textos, molduras, etc.) e para o "fundo" do vídeo. A sua sintaxe é, simplesmente,

Cor Expr1, Expr2

onde Expr1 e Expr2 podem ser variáveis do tipo *Numerico* ou do tipo *Caracter* ou, ainda, valores constante destes tipos de dados, de acordo com a tabela a seguir.

Numerico	Caracter	Numerico	Caracter
0	preto	9	azul_intenso
1	azul	10	verde_intenso
2	verde	11	ciano_intenso
3	ciano	12	vermelho_intenso
4	vermelho	13	magenta_intenso
5	magenta	14	marrom_intenso
6	marrom	15	amarelo
7	cinza	16	branco
8	preto_intenso		

6.4 Exercícios propostos

1. Escreva um programa que armazene uma relação de números na ordem inversa da ordem original. Por exemplo, se a relação dada for (3, 6, 8, 9) o programa deve armazenar (9, 8, 6, 3).

2. Escreva um programa que receba um vetor e o decomponha em dois outros vetores, um contendo as componentes de ordem ímpar e o outro contendo as componentes de ordem par. Por exemplo, se o vetor dado for (3, 5, 6, 8, 1, 4, 2, 3, 7), o vetor deve gerar os vetores (3, 6, 1, 2, 7) e (5, 8, 4, 3).

3. Escreva um programa que receba um vetor de números inteiros e o decomponha em dois outros vetores, um contendo as componentes de valor ímpar e o outro contendo as componentes de valor par. Por exemplo, se o vetor dado for (3, 5, 6, 8, 1, 4, 2, 3, 7) o programa deve gerar os vetores (3, 5, 1, 3, 7) e (6, 8, 4, 2).

4. Um *vetor* do R^n é uma n-upla de números reais (x_1, x_2, ..., x_n), sendo cada x_i chamado de *componente*. A *norma* de um vetor (x_1, x_2, ..., x_n) é definida por $\sqrt{x_1^2+x_2^2+\ldots+x_n^2}$. Escreva um programa que receba um vetor do R^n, n dado, e forneça sua norma.

5. O *produto escalar* de dois vetores do R^n é a soma dos produtos das componentes correspondentes. Isto e, se u = (x_1, x_2, ..., x_n) e v = (y_1, y_2, ..., y_n), o *produto escalar* é $x_1.y_1 + x_2.y_2 \ldots + x_n.y_n$. Escreva um programa que receba dois vetores do R^n, n dado, e forneça o produto escalar deles.

6. A *amplitude* de uma relação de números reais é a diferença entre o maior e o menor valores da relação. Por exemplo, a *amplitude* da relação 5, 7, 15, 2, 23 21, 3, 6 é 23 - 2 = 21. Escreva um programa que receba uma relação de números e forneça sua *amplitude*.

7. Os *desvios* de uma relação de números reais são as diferenças entre cada número e a média aritmética da relação. O *desvio médio* de uma relação de números reais é a média aritmética dos valores absolutos dos *desvios*. Escreva um programa que receba uma relação de números reais e forneça o seu *desvio médio*.

8. O *desvio padrão* de uma relação de números reais é a raiz quadrada da média aritmética dos quadrados dos desvios. Escreva um programa que receba uma relação de números reais e forneça o seu *desvio padrão*.

9. Escreva um programa que forneça as componentes distintas de um vetor dado. Por

exemplo, se o vetor dado for (3, 2, 1, 3, 4, 1, 5, 5, 2) o programa deve fornecer (3, 2, 1, 4, 5).

10. O exemplo 3 da seção 2.9 pedia um programa para extrair o algarismo da casa das unidades de um inteiro dado. Aparentemente esta questão não tem interesse prático. Vejamos um problema cuja solução depende desta questão. Algumas empresas que realizam sorteios de prêmios entre seus clientes o fazem através dos sorteios da loteria federal, sendo ganhador o número formado pelos algarismos das casas das unidades dos números sorteados no cinco prêmios da referida loteria. Por exemplo, se o sorteio da loteria federal deu como resultado os números 2345**1**, 0023**4**, 1123**6**, 0123**5** e 2345**2**, o prêmio da tal empresa seria dado ao cliente que possuísse o bilhete de número **14652**. Escreva um programa que receba os números sorteados pela loteria federal e forneça o número que ganhará o prêmio de acordo com as regras acima (vale observar que o programa não funcionará corretamente se um dos números sorteados for maior que 32767, conforme observação final da seção 2.6. Não se preocupe com isto! O que nos interessa aqui não é o programa e sim a aprendizagem da lógica de programação).

11. Escreva um programa que insira um valor dado num vetor dado numa posição dada. Por exemplo se o vetor dado for (3, 8, 5, 9, 12, 3), o valor dado for 10 e a posição dada for 4, o programa deve fornecer (3, 8, 5, **10**, 9, 12, 3).

12. Escreva um programa que insira um valor dado num vetor ordenado dado de modo que o vetor continue ordenado. Por exemplo, se o vetor dado for (2, 5, 7, 10, 12, 13) e o valor dado for 6, o programa deve fornecer o vetor (2, 5, **6**, 7, 10, 12, 13).

13. Escreva um programa que delete uma componente de ordem dada de um vetor dado. Por exemplo, se o vetor dado for (2, 5, 7, 10, 12, 13) e a componente a ser deletada for a de ordem 4, programa deve fornecer o vetor (2, 5, 7, 12, 13).

14. Escreva um programa que, dadas duas relações de números, cada uma delas com números distintos, forneça os números que aparecem nas duas listas. Por exemplo. se as relações forem (9, 32, 45, 21, 56, 67, 42, 55) e (24, 42, 32, 12, 45, 11, 67, 66, 78), o programa deve fornecer o vetor (32, 45, 67, 42).

15. Escreva um programa que, dado um vetor ordenado, forneça a maior diferença entre duas componentes consecutivas, fornecendo também as ordens das componentes que geraram esta maior diferença. Por exemplo, se o vetor dado for (3, 5, 9, 16, 17, 20, 26, 31), o programa deve fornecer como maior diferença o valor 7 (16 - 9), e as ordens 4 e 3.

16. Uma avaliação escolar consiste de 50 questões objetivas, cada uma delas com 5 opções, (1, 2, 3, 4 e 5), sendo apenas uma delas verdadeira. Escreva um programa que receba a sequência de respostas corretas, o *gabarito*, e corrija um cartão-resposta dado.

17. Escreva um programa que forneça o valor numérico de um polinômio P(x) dado, para um valor de x dado. Por exemplo, se o polinômio dado for $P(x) = x^3 + 2x - 1$ e o valor de x dado for 2, o programa deve fornecer $P(2) = 2^3 + 2x2 - 1 = 11$.

18. O(s) valor(es) de maior frequência de uma relação de valores numéricos é(são) chamado(s) *moda* da relação. Escreva um programa que receba uma relação de notas escolares maiores do que zero e menores do que ou iguais a 10, com uma casa decimal, e forneça a(s) moda(s) desta relação. Por exemplo, se a relação de notas for (8,0; 3,5, 4,5; 8,0; 6,0; 4,5; 6,0; 3,5; 2,5; 6,0; 9,0) o programa deve fornecer o valor 6,0 (frequência 3).

19. Escreva um programa que receba um número inteiro n e forneça o número formado pelos algarismos de n escritos na ordem inversa. Por exemplo se o número dado for 3876, o programa deve fornecer 6783.

20. A matemática prova que a conversão de um número do sistema decimal para o sistema binário pode ser feita através de divisões sucessivas do número e dos quocientes sucessivamente obtidos por 2, sendo então o número binário dado pela sequência iniciada por 1 e seguida pelos restos obtidos nas divisões sucessivas, na ordem inversa em que são obtidos. Por exemplo, para se converter 22 do sistema decimal para o sistema binário temos: Resto(22, 2) = 0; Resto(11, 2) = 1; Resto(5, 2) = 1; Resto(2, 2) = 0 e, portanto, $22s = (10110)_2$. Escreva um programa que converta um número positivo dado no sistema decimal de numeração para o sistema binário, usando o algoritmo

acima.

21. O exercício 10 da seção 4.5 solicitava um programa que determinasse a *decomposição em fatores primos*, fornecendo os fatores primitivos e suas respectivas *multiplicidades*. Na ocasião os fatores primos e suas multiplicidades eram apenas exibidos não sendo armazenados. Modifique o programa referido para que os fatores primos e as suas multiplicidades sejam armazenados, antes de serem exibidos.

22. A Universidade Federal de Alagoas adota o sistema de verificação de aprendizagem listado no exemplo 5 da seção 3.3, com o adendo de que terá direito a uma *reavaliação* um aluno que obtiver uma nota inferior a 7,0 em algum bimestre. Neste caso, a nota obtida na reavaliação substitui a menor das notas bimestrais obtidas. Escreva um programa que, recebendo as notas das avaliações bimestrais e, se for o caso, a nota da reavaliação e, se for o caso, a nota da prova final, forneça a média final de um aluno da UFAL e a sua condição em relação à aprovação.

23. Escreva um programa que forneça a *transposta* de uma matriz dada.

24. Um dos métodos para a se estudar as soluções de um *sistema linear de n equações a n incógnitas* aplica *operações elementares sobre as linhas da matriz dos coeficientes*, sendo a permuta de duas linhas uma destas operações elementares. Escreva um programa que permute as posições de duas linhas de uma matriz dadas.

25. Uma matriz quadrada é dita *triangular* se os elementos situados acima de sua diagonal principal são todos nulos. Escreva um programa que receba uma matriz quadrada e verifique se ela é *triangular.*

26. O exemplo 7 da seção 6.2 apresentou um programa para armazenar uma matriz simétrica. Este exercício quer algo contrário: escreva um programa que verifique se uma matriz dada é simétrica.

27. Escreva um programa que receba uma matriz e totalize suas colunas. Por exemplo, se a matriz dada for $\begin{pmatrix} 2 & 6 \\ 5 & 8 \\ 6 & 9 \end{pmatrix}$, o programa deve fornecer a matriz $\begin{pmatrix} 2 & 6 \\ 5 & 8 \\ 6 & 9 \\ 13 & 23 \end{pmatrix}$. Naturalmente um programa deste seria utilizado para totalizar as colunas de uma tabela de valores numéricos.

28. Escreva um programa que determine as médias de cada uma das linhas de uma matriz. Por exemplo, se a matriz dada for

$$\begin{pmatrix} 3 & 7 & 4 & 6 \\ 5 & 2 & 3 & 4 \\ 2 & 6 & 3 & 1 \end{pmatrix},$$

o programa deve fornecer a matriz

$$\begin{pmatrix} 3 & 7 & 4 & 6 & 5{,}0 \\ 5 & 4 & 5 & 4 & 4{,}5 \\ 2 & 6 & 5 & 1 & 3{,}5 \end{pmatrix}$$

29. Escreva um programa que determine o menor valor de cada uma das linhas de uma matriz dada, fornecendo o índice da coluna que contém este menor valor. Por exemplo, se a matriz dada for $\begin{pmatrix} 3 & 7 & 4 & 6 \\ 5 & 2 & 3 & 4 \\ 2 & 6 & 3 & 1 \end{pmatrix}$, o programa deve fornecer uma tabela do tipo

Linha	Menor valor	Coluna
1	3	1
2	2	2
3	1	4

Um programa como este poderia receber os preços de diversos produtos praticados por vários supermercados e forneceria, para cada produto, o menor preço e o supermercado que pratica este melhor preço.

30. No exemplo 7 da seção anterior vimos como armazenar uma matriz simétrica. Na prática, uma matriz deste tipo ocorre, por exemplo, numa tabela de distâncias entre cidades, como a

seguinte tabela que dá as distâncias aéreas, em Km, entre as capitais dos estados nordestinos (Aracaju, Fortaleza, João Pessoa, Maceió, Natal, Recife, Salvador, São Luís, Teresina).

	A	F	JP	M	N	R	S	SL	T
A	0	812	438	210	550	398	267	1218	1272
F	812	0	562	730	444	640	1018	640	432
JP	418	562	0	284	144	110	758	1208	987
M	210	730	294	0	423	191	464	1220	1126
N	550	414	144	423	0	252	852	1064	843
R	398	640	118	191	252	0	654	1197	935
S	267	1018	758	464	852	654	0	1319	1000
SL	1218	640	1208	1220	1064	1197	1319	0	320
T	1272	432	987	1126	843	935	1000	320	0

Imagine que uma companhia de transporte aéreo estabeleça que uma viagem entre duas cidades que distem mais de 400 Km deve ter uma escala. Escreva um programa que armazene uma tabela das distâncias aéreas entre n cidades e dadas duas cidades determine, se for o caso, a cidade em deve se realizar uma escala para que o percurso seja o menor possível. Por exemplo, nas condições estabelecidas, a viagem entre Maceió e São Luís deve ter uma escala em Fortaleza (o percurso Maceió/Fortaleza/São Luís é de 1370 Km; o percurso, por exemplo, Maceió/Recife/São Luís é de 1388 Km)

31. Esta questão não envolve variáveis compostas. Ela se encontra neste capítulo para se exercitar formatação de saída. Escreva um programa para gerar uma tabuada para multiplicação, exibindo-a na forma usual de tabuadas.

Observação

Para obter o caderno das soluções dos exercícios propostos basta encaminhar e-mail para jaime@ccen.ufal com assunto RESPOSTAS LIVRO ILA e contendo NOME, CATEGORIA (docente, discente, autodidata), CIDADE/ESATADO e, se for o caso, INSTITUIÇÃO/CURSO.

7. Pesquisa e Ordenação

7.1 Introdução

Embora os problemas que serão abordados neste capítulo sejam exercícios de manipulação de vetores, optamos por destacá-los numa seção específica devido à importância deles para a computação como um todo e, em particular, para o desenvolvimento da lógica de programação.

Uma *pesquisa* (ou busca ou *consulta)* consiste em se verificar se um dado valor é uma componente de uma variável composta. Para exemplificar, consideremos o Registro Nacional de Veículos Automotores (RENAVAM). O RENAVAM contém dados de todos os veículos registrados nos Departamentos Estaduais de Trânsito, dados que incluem, entre outros, a placa do veículo, o número do motor, o nome do proprietário, o CPF do proprietário. Evidentemente, pode ser necessário se verificar se uma determinada placa é de um veículo registrado ou se um dado CPF é de alguém que é proprietário de algum veículo.

A questão da *ordenação* (também chamada *classificação*) foi introduzida na seção 3.3 e consiste em se colocar numa ordem preestabelecida uma relação de valores. Na seção referida, mostramos como ordenar uma relação contendo três valores. Neste capítulo, apresentaremos algoritmos para ordenar uma relação com qualquer números de valores. A ordenação de uma relação é realizada para que a leitura dos resultados seja facilitada ou para que, como veremos a seguir, pesquisas sejam realizadas com mais eficiência.

7.2 Pesquisa sequencial

A *pesquisa sequencial* é o método mais simples e mais óbvio (embora, o mais ineficiente) de se realizar uma busca. No nosso caso, em que realizaremos pesquisas num vetor, a pesquisa sequencial consiste em se percorrer o vetor comparando cada componente com o valor a ser pesquisado.

```
Variaveis
        Numerico Aux, i, j, x, Pos
        Matriz numerico Vet[50]
Inicio
        Escrever "Digite os elementos da relação (-1 para encerrar)"
        Ler Aux
        Vet[1] = Aux
        i = 1
        Faca enquanto Vet[i] <> -1
                Ler Aux
                i = i + 1
                Vet[i] = Aux
        Fim_enquanto
        i = i - 1
        Escrever "Digite o valor a ser pesquisado"
        Ler x
        j = 1
        Faca enquanto (Vet[j] <> x) e (j < i)
                j = j + 1
        Fim_enquanto
        Se Vet[j] = x entao
                Pos = j
                Escrever x, " encontrado na posição ", Pos
        Senao
```

```
            Escrever x, " não encontrado"
        Fim_se
Fim
```

7.3 Pesquisa binária

É fácil perceber que a pesquisa sequencial é muito ineficiente. Imagine que se queira pesquisar num dicionário o significado da palavra *zumbaia*. Pelo método anterior, começaríamos na letra *a* e iríamos de palavra em palavra até encontrar a palavra cujo significado estamos procurando (a propósito, ***zumbaia*** significa cortesia exagerada; cumprimento ruidoso e servil).

Existe um método, chamado *pesquisa binária*, que se aplica quando o vetor possui seus elementos ordenados (que é o caso de um dicionário) e é bem mais eficiente que a pesquisa sequencial. Este método consiste em se comparar o valor a ser pesquisado *x* com o valor da componente que ocupa a posição central do vetor. Se *x* for igual a esta componente a pesquisa se encerra; se *x* for menor que esta componente, repete-se o raciocínio com a "metade inferior" do vetor; se *x* for maior que a componente central, repete-se o raciocínio com a "metade superior" do vetor. Isto se repete até que *x* seja encontrado ou que se tenha condições de se afirmar que *x* não está no vetor.

Por exemplo, se se pretende pesquisar x = 16 no vetor Vet = (1, 2, 4, 7, 9, 11, 13, 14, 16, 18, 19, 22, 24), compara-se 16 com o termo central do vetor (13); como *x* é maior que este termo central, repete-se o raciocínio com o vetor Vet1 = (14, 16, 18, 19, 22, 24). Assim compara-se 16 com o novo "termo central" 18; como *x* é menor, raciocina-se com o vetor Vet2 = (14, 16). Agora o termo central é 14 e *x* é maior que ele. Passa-se então para o vetor Vet3 = {16} e aí a pesquisa se encerra com sucesso. O interessante da implementação deste método é que não há necessidade da criação destes vetores. Eles são criados abstratamente através de duas variáveis *Prim* e *Ult* que estabelecem quais são as primeira e última componentes do vetor que se está analisando.

```
//Programa para pesquisa pelo método da pesquisa binária
Variaveis
    Numerico Aux, i, x, Prim, Ult, Cent
    Matriz numerico Vet[50]
Inicio
    Escrever "Digite os elementos da relação (-1 para encerrar)"
    Ler Aux
    Vet[1] = Aux
    i = 1
    Faca enquanto Vet[i] <> -1
        Ler Aux
        i = i + 1
        Vet[i] = Aux
    Fim_enquanto
    i = i - 1
    //Comandos para a pesquisa binária
    Escrever "Digite o valor a ser pesquisado"
    Ler x
    Prim = 1
    Ult = i
    Cent = Inteiro((Prim + Ult)/2)
    Faca enquanto (Vet[Cent] <> x) e (Prim < Ult)
        Se Vet[Cent] > x entao
            Ult = Cent - 1
        Senao
```

```
                Prim = Cent + 1
            Fim_se
            Cent = Inteiro((Prim + Ult)/2)
        Fim_enquanto
        Se Vet[Cent] = x entao
            Escrever x, " encontrado na posição ", Cent
        Senao
            Escrever x, " não encontrado"
        Fim_se
Fim
```

A *pesquisa binária* também permite (nos sistemas que admitem variáveis compostas como parâmetros de funções) um bom exemplo de uma *função recursiva*. Embora o ILA não aceite variáveis compostas como parâmetros, vamos apresentar a solução recursiva da pesquisa binária que poderia ser implementada neste sistema, caso o impedimento anterior não existisse.

```
//Função recursiva para a pesquisa binária (não executável)
Funcao PesqBinRec(V, Prim, Ult, x)
//V seria um vetor, Prim, Ult e x variáveis do tipo Numerico
Inicio
    Cent = Inteiro((Prim + Ult)/2)
    Se (V[Cent] = x) ou (Prim > Ult) entao
        Se V[Cent] = x entao
            Escrever x , " encontrado na posição ", Cent
        Senao
            Escrever x, " não encontrado"
        Fim_se
    Senao
        Se V[Cent] > x entao
            Retornar PesqBinRec(V, Prim, Cent - 1)
        Senao
            Retornar PesqBinRec(V, Cent + 1, Ult)
        Fim_se
    Fim_se
Fim
```

7.4 Ordenação pelo algoritmo SelectSort

O algoritmo *SelectSort* consiste em se selecionar, sucessivamente, o maior elemento, o segundo maior elemento, o terceiro maior elemento, etc., e, após cada seleção, armazenar o valor selecionado num vetor auxiliar na posição que mantém o tal vetor auxiliar ordenado. Por exemplo, se se pretende a ordenação em ordem crescente, o "primeiro" maior valor é armazenado na última posição do vetor auxiliar; o segundo maior valor é armazenado na penúltima posição do vetor auxiliar e assim sucessivamente. Para que se obtenha, por exemplo, o segundo maior valor do vetor, excluímos o "primeiro" maior valor atribuindo a esta componente um valor sabidamente menor do que todos os valores armazenados no vetor. Por exemplo, se os valores do vetor são positivos pode-se atribuir -1 a cada componente já selecionada e já armazenada no vetor auxiliar.

Para exemplificar o método, vamos ordenar o vetor Vet = (5, 2, 7, 1, 8). Basta percorrer o vetor 5 vezes selecionando sucessivamente 8, 7, 5, 2 e 1 e realizando as seguintes atribuições:

1. Aux = (, , , , 8)

Vet = (5, 2, 7, 1, -1)
2. Aux = (, , , 7, 8)
Vet = (5, 2, -1, 1, -1)
3. Aux = (, , 5, 7, 8)
Vet = (-1, 2, -1, 1, -1)
4. Aux = (, 2, 5, 7, 8)
Vet = (-1, -1, -1, 1, -1)
5. Aux = (1, 2, 5, 7, 8)
Vet = (-1, -1, -1, -1, -1),

Para finalizar, basta armazenar nas componentes de Vet as componentes de Aux.

```
//Ordenação de um vetor pelo algoritmo SelectSort
Variaveis
        Numerico Aux, i, j, k, x, Maior
        Matriz numerico Vet[50]
        Matriz numerico Aux[50]
Inicio
        Escrever "Digite os elementos da relação (-1 para encerrar)"
        Ler Aux
        Vet[1] = Aux
        i = 1
        Faca enquanto Vet[i] <> -1
                Ler Aux
                i = i + 1
                Vet[i] = Aux
        Fim_enquanto
        i = i - 1
        //Comandos do SelectSort
        Para j = 1 ate i
                Maior = Vet[1]
                x  = 1
                Para k = 1 ate i
                        Se Vet[k] > Maior entao
                                Maior = Vet[k]
                                x = k
                        Fim_se
                Proximo
                Aux[i - j + 1] = Maior
                Vet[x] = -1
        Proximo
        Para j = 1 ate i
                Vet[j] = Aux[j]
        Proximo
        Para j = 1 ate i
                Escrever Vet[j]
        Proximo
Fim
```

7.5 Ordenação pelo algoritmo BubbleSort

O algoritmo *BubbleSort* consiste em se percorrer o vetor a ser ordenado várias vezes, comparando-se cada elemento com o seguinte, permutando suas posições se eles não estiverem na

ordem pretendida. Assim, cada vez que o vetor é percorrido, o maior elemento ainda não ordenado é colocado na sua posição de ordenação definitiva. Naturalmente, o vetor será percorrido até que não haja mais trocas a se fazer, quando então ele estará ordenado. Por exemplo, se o vetor a ser ordenado em ordem crescente fosse Vet = (5, 1, 9, 3, 7, 2), teríamos as seguintes configurações para Vet, de acordo com a ordem de percurso:

Percurso	Vet
0	(5, 1, 9, 3, 7, 2)
1	(1, 5, 9, 3, 7, 2)
	(1, 5, 3, 9, 7, 2)
	(1, 5, 3, 7, 9, 2)
	(1, 5, 3, 7, 2, 9)
2	(1, 3, 5, 7, 2, 9)
	(1, 3, 5, 2, 7, 9)
3	(1, 3, 2, 5, 7, 9)
4	(1, 2, 3, 5, 7, 9)

```
 //Ordenação de um vetor por BubbleSort
Variaveis
        Numerico Aux, i, j, k, x
        Matriz numerico Vet[50]
        Logico Troca
Inicio
        Escrever "Digite os elementos da relação (-1 para encerrar)"
        Ler Aux
        Vet[1] = Aux
        i = 1
        Faca enquanto Vet[i] <> -1
                Ler Aux
                i = i + 1
                Vet[i] = Aux
        Fim_enquanto
        i = i - 1
        //Comandos do BubbleSort
        j = i
        Troca = verdadeiro
        Faca enquanto Troca = verdadeiro
                j = j - 1
                Troca = falso
                Para k = 1 ate j
                        Se Vet[k] > Vet[k + 1] entao
                                Aux = Vet[k]
                                Vet[k] = Vet[k + 1]
                                Vet[k + 1] = Aux
                                Troca = verdadeiro
                        Fim_se
                Proximo
        Fim_enquanto
        Para j = 1 ate i
                Escrever Vet[j]
        Proximo
Fim
```

Observe a utilização da variável do tipo *Logico* Troca. É ela que verifica se houve alguma troca, para que se faça um novo percurso no vetor, através do comando *Troca = verdadeiro*. Quando não acontece mais nenhuma troca, o vetor já está ordenado, e o valor da variável Troca se mantém *falso*. Assim, a estrutura *Faca enquanto Troca = verdadeiro* se encerra.

Observe também que o comando j = j - 1 é para que um novo percurso não compare os elementos que já estão ordenados.

7.6 Exercícios propostos

1. A maioria das pessoas acham que são azaradas quando procuram uma ficha numa pilha, sempre tendo receio que a ficha procurada seja uma das últimas da pilha. Uma pessoa que acredite ser assim azarada pode pesquisar a tal ficha pesquisando, sucessivamente, a parte superior e a parte inferior da pilha. Assim, verifica a primeira ficha, em seguida, a última, em seguida, a segunda ficha, em seguida, a penúltima e assim sucessivamente. Escreva um programa que implemente este método de pesquisa.

2. Uma versão melhorada do *SelectSort* para um vetor que possui i componentes consiste em se comparar a maior dentre as i - 1 primeiras componentes com a última componente, permutando-se suas posições se aquela maior componente for menor do que esta última componente, colocando-se então o maior elemento na posição desejada. Este raciocínio é repetido no vetor das i - 1 primeiras componentes e assim sucessivamente. Escreva um programa que implemente esta versão do *SelectSort*.

3. A algoritmo *InsertSort* para ordenação de um vetor Vet consiste em se tomar um vetor auxiliar Aux, contendo uma única componente Vet[1], Aux = (Vet[1]). Em seguida, insere-se as demais componentes de Vet, uma a uma, em Aux de modo que Aux se mantenha ordenado, como solicitado no exercício 12 da seção 6.4. Escreva um programa que implemente o *InsertSort*.

Observação

Para obter o caderno das soluções dos exercícios propostos basta encaminhar e-mail para jaime@ccen.ufal com assunto RESPOSTAS LIVRO ILA e contendo NOME, CATEGORIA (docente, discente, autodidata), CIDADE/ESATADO e, se for o caso, INSTITUIÇÃO/CURSO.

8. Cadeias de caracteres

8.1 Introdução

O estudo específico do tipo de dados *Caracter* propicia a oportunidade de discutir várias questões importantes para a prática da computação e para o desenvolvimento da lógica de programação. De um modo geral, os sistemas de computação para desenvolvimento de programas possuem várias funções pré-definidas para a manipulação de cadeias caracteres (nestes sistemas chamadas *strings*). No ILA estão definidas duas funções pré-definidas, além de um operador, com o objetivo aqui discutido. Para o objetivo deste livro, este número pequeno de funções pré-definidas do ILA é muito útil, pois permite que sejam discutidos os algoritmos que implementam as outras funções, o que é bem interessante para o desenvolvimento da lógica de programação.

Dentre as funções para manipulação de caracteres que estão definidas nos vários ambientes de programação estão funções que deletam um dado número de caracteres a partir de uma posição dada, que inserem uma cadeia de caracteres em outra cadeia de caracteres e que transformam um dado valor numérico numa cadeia de caracteres.

8.2 O operador *concatenação* e as funções *Parte* e *Valor*

O operador *concatenação* (símbolo +) atua em operadores do tipo cadeia caracteres anexando a segunda cadeia à primeira. Por exemplo, a execução da sequência de comandos

```
s1 = "ABA"
s2 = "CATE"
s = s1 +  s2
Escrever s
```

escreverá na tela a palavra ABACATE.

A função *Valor* converte uma cadeia de caracteres em um número. Para que a conversão seja possível, os caracteres da cadeia argumento devem ser dígitos, podendo o primeiro deles ser um dos caracteres + ou - e um dos caracteres a partir do segundo ser um ponto final, Se o primeiro caractere não for um dos permitidos, a função retorna zero e se um dos outros caracteres não for um dígito ou existirem dois pontos finais adjacentes, a função retorna o número correspondente à subcadeia que permitiu a conversão. A sintaxe da função *Valor* é

```
Valor(s)
```

onde s é um valor do tipo caractere.

Por exemplo, a execução da sequência de comandos

```
s1 = "3245"
s2 = "4.897"
s3 = "23x45"
s4 = "x1"
x = Valor(s1)
y = Valor(s2)
z = Valor(s3)
w = Valor(s4)
Escrever x,"    ", y, "     ", z, "     ", w
```

escreve na tela 3245 4.90 23 0. Naturalmente, x, y, z e w são variáveis numéricas e uma diferença importante entre x e s1 é que com x se pode realizar operações aritméticas.

É de se esperar que alguém que está aprendendo a programar se pergunte: para que transformar uma cadeia de caracteres em um número? Uma aplicação que necessita esta transformação é a manipulação de matrículas de funcionários de uma empresa ou de estudantes de

uma escola, números de contas bancarias, inscrições no *Cadastro Geral de Contribuintes*, etc.. Quase sempre estes identificadores são dados por cadeias de caracteres que envolvem, além de dígitos, traços, barras e, até, letras. Por exemplo, houve um tempo que as matrículas dos alunos da Universidade Federal de Alagoas tinham o formato 1999G24D008V9, onde 1999 indicava o ano de ingresso, G indicava o nível do curso (no exemplo, *graduação*), 24 indicava o código do curso, D referia o turno (no exemplo, *diurno*), 008 indicava um número de série dentre todos os alunos do curso diurno de graduação 24 ingressantes em 1999, V referia a forma de ingresso (no exemplo, ingresso por *vestibular*) e 9 era um *dígito verificador* que serve para detectar possíveis erros de digitação. Como veremos no exemplo 8 da seção 8.4, a determinação do *digito verificador* envolve operações aritméticas com os números correspondentes aos dígitos que compõem a matrícula, a conta ou a inscrição. Assim, necessitaremos transformar estes caracteres em números para que possamos realizar as operações exigidas.

Outra aplicação importante da transformação de cadeia de caracteres em números ocorre nos sistemas visuais de programação. Normalmente, estes sistemas não possuem um comando do tipo *Ler*, sendo a entrada dos dados feita através de *boxes de edição*, oferecidos pelo sistema para a criação de interfaces. Nestes boxes de edição o usuário digita os dados de entrada, mesmo os numéricos, como uma cadeia de caracteres havendo então a necessidade da transformação.

A outra função do ILA para manipulação de cadeias de caracteres é a função *Parte* que retorna uma subcadeia da cadeia que lhe é passada como argumento. Esta função possui três parâmetros: a cadeia da qual se quer retornar uma subcadeia, um inteiro que indica a posição a partir da qual a subcadeia pretendida ocorre na cadeia e um outro inteiro que indica quantos caracteres tem a subcadeia pretendida. A função *Parte* possui a seguinte sintaxe

```
Parte(s, p, n)
```

onde, como foi dito acima, s é uma cadeia de caracteres, p e n são inteiros, p indicando a posição a partir da qual está a subcadeia pretendida e n o número de caracteres da tal subcadeia.

Por exemplo, a execução da sequência de comandos

```
s1 = "abacate"
s2 = Parte(s1, 4, 2)
Escrever s2
```

escreve na tela a palavra *ca*. Vale observar que se o número de caracteres pretendido for maior que o número de caracteres que existem a partir da posição p, a função retornará toda a subcadeia a partir da posição p. Por exemplo, se o segundo comando da sequência acima fosse s2 = Parte(s1, 4, 10) seria exibido na tela a palavra *cate*.

8.3 Exemplos Parte VII

Nesta seção, apresentaremos as implementações em ILA de várias funções disponibilizadas pelos diversos sistemas de computação para manipulação de cadeias de caracteres.

1. Vejamos, como primeiro exemplo, uma função que delete um dado número de caracteres de uma cadeia dada, a partir de uma posição dada. Para isto, a função terá parâmetros idênticos aos parâmetros da função *Parte*: a cadeia de caracteres (s), a posição a partir da qual ocorrerá a exclusão (p) e o número de caracteres a serem excluídos (n). A idéia da função é bem simples (c é o comprimento de s): se $p + n \leq c$, a exclusão se dará com a concatenação dos p - 1 primeiros caracteres com os c - (p + n) + 1 últimos caracteres; se $p + n > s$, basta se tomar os p - 1 primeiros caracteres.

```
Variaveis
      Caracter  s
      Numerico c, p, n
Funcao Delete(s, p, n)
```

```
Inicio
        c = Comprimento(s)
        Se p + n <= c entao
                s = Parte(s, 1, p - 1) + Parte(s, p + n, c - p - n + 1)
        Senao
                s = Parte(s, 1, p - 1)
        Fim_se
        Retornar s
Fim
```

2. Vejamos agora uma função que insere uma cadeia de caracteres em outra cadeia, a partir de uma posição dada. Por exemplo, queremos uma função *Insere* que sendo ativada por Insere("abate", "ca", 4) retorne "abacate". Definiremos então uma função com três parâmetros: a cadeia que receberá a inserção (s1), a cadeia que será inserida (s2) e a posição de inserção (p). Naturalmente, se c = Comprimento(s1), a inserção só pode ser realizada se p <= c + 1. Neste caso, se p = c + 1, a inserção é obtida, simplesmente, concatenando-se s1 e s2; se p < c + 1, a inserção é obtida concatenando-se os p - 1 primeiros caracteres de s1 com s2 e com os c - (p -1) últimos caracteres de s2.

```
Variaveis
        Caracter s1, s2
        Numerico c, p
Funcao Insere(s1, s2, p)
Inicio
        c = Comprimento(s1)
        Se p > c + 1 entao
                Escrever "Inserção impossível"
        Senao
                Se p = c + 1 entao
                        s1 = s1 + s2
                Senao
                        s1 = Parte(s1, 1, p - 1) + s2 + Parte(s1, p, c - p + 1)
                Fim_se
        Retornar s1
        Fim_se
Fim
```

3. Agora uma função cuja ação é inversa da ação função *Valor*: recebe um número e o armazene como uma cadeia de caracteres. Se o número é um número inteiro, a função é simples. Basta extrair os números correspondentes aos algarismos do número, transformá-los nos respectivos algarismos e concatená-os em seguida. Para esta concatenação, inicializamos uma cadeia de caracteres com a cadeia contendo um espaço em branco e concatenamos os tais algarismos, um a um.

```
Variaveis
        Numerico x, r, n
        Caracter s
Funcao IntParaCarac (n)
Inicio
        s = " "
        Se n >= 0 entao
                x = n
        Senao
                x = -n
```

```
        Fim_se
        Faca enquanto (x > 0)
                r = Resto(x, 10)
                Faca caso
                        Caso r = 0:
                                s =  "0" + s
                        Caso r = 1:
                                s =  "1" + s
                        Caso r = 2:
                                s =  "2" + s
                        Caso r = 3:
                                s =  "3" + s
                        Caso r = 4:
                                s =  "4" + s
                        Caso r = 5:
                                s =  "5" + s
                        Caso r = 6:
                                s =  "6" + s
                        Caso r = 7:
                                s =  "7" + s
                        Caso r = 8:
                                s =  "8" + s
                        Caso r = 9:
                                s =  "9" + s
                Fim_caso
                x = Inteiro(x/10)
        Fim_enquanto
        s = Parte(s, 1, Comprimento(s) - 1)
        Se n < 0 entao
                s = "-" + s
        Senao
                Se n = 0 entao
                        s = "0"
                Fim_se
        Fim_se
        Retornar s
Fim
```

Observe que o comando s = Parte(s, 1, Comprimento(s) - 1) é para eliminar o espaço em branco que inicializou *s* (isto não seria necessário se o sistema aceitasse inicializar uma cadeia de caracteres com a *cadeia vazia*, ""). Observe também que a seleção *Se n < 0 ...* é para tratar o caso dos números negativos e a seleção *Se n = 0 ...* é para tratar o caso n = 0, já que o ILA não trata bem a operação Resto(0, 10). Vale observar também que o comando *Faca caso* que concatena os caracteres e transforma em caracteres números positivos menores que dez não seria necessário em sistemas de objetivos mais gerais: estes sistemas possuem funções pré-definidas para realizar estas ações.

Para o caso dos números decimais (chamados também de números de *ponto flutuante*), podemos separar o número nas suas partes inteira e fracionária e aplicar a função acima a cada uma destas partes. Para transformar num inteiro, como o sistema trabalha apenas com duas casas decimais, multiplicamos a parte fracionária por 100. Se o número a ser convertido está armazenado na variável Num, a sequência de comandos abaixo realiza a conversão aqui discutida.

```
Int = Inteiro(Num)
```

```
f = 100*(Num - Int)
f = Inteiro(f)
Se f < 0 entao
        f = - f
Fim_se
s = IntParaCarac (Int) + "." + IntParaCarac (f)
Escrever s
```

4. Finalmente, uma função que verifique se uma cadeia de caracteres é subcadeia de outra cadeia de caracteres. Esta função, em processadores de texto, serve para localizar cadeias de caracteres dentro de um texto.

```
Variaveis
        Caracter s1, s2, s
        Numerico c1, c2, i, p
Funcao Pos(s1, s2)
Inicio
        p = 0
        c1 = Comprimento(s1)
        c2 = Comprimento(s2)
        i = 1
        s = Parte(s1, i, c2)
        Faca enquanto (s <> s2) e (i < c1 - c2 + 1)
                i = i + 1
                s = Parte(s1, i, c2)
        Fim_enquanto
        Se s = s2 entao
                p = i
        Fim_se
        Retornar p
Fim
```

8.4 Exemplos Parte VIII

Agora apresentaremos programas bastante usuais em computação e que usam as funções para manipulação de cadeias de caracteres pré-definidas do sistema e as funções definidas acima.

5. O primeiro programa tem o objetivo de converter uma data dada no formato americano mm/dd/aaaa para o formato brasileiro dd/mm/aaaa.

```
Variaveis
        Caracter Dia, Usa, Br, s1, s2
        Numerico c, p, n
Funcao Insere(s1, s2, p)
Inicio
        c = Comprimento(s1)
        Se p > c + 1 entao
                Escrever "Inserção impossível"
        Senao
                Se p = c + 1 entao
                        s1 = s1 + s2
                Senao
                        s1 = Parte(s1, 1, p - 1) + s2 + Parte(s1, p, c - p + 1)
                Fim_se
```

```
            Retornar s1
        Fim_se
    Fim
    Funcao Delete(s1, p, n)
    Inicio
        c = Comprimento(s1)
        Se p + n <= c entao
            s1 = Parte(s1, 1, p - 1) + Parte(s1, p + n, c - p - n + 1)
        Senao
            s1 = Parte(s1, 1, p - 1)
        Fim_se
        Retornar s1
    Fim
    Inicio //Programa principal
        Escrever "Digite a data no formato mm/dd/aaaa"
        Ler Usa
        Br = Usa
        Dia = Parte(Usa, 4, 3)
        Br = Delete(Usa, 4, 3)
        Br = Insere(Br, Dia, 1)
        Escrever "Data convertida ao formato dd/mm/aaaa:  ", Br
    Fim
```

6. O programa anterior poderia possuir uma função lógica que verificasse se a data dada era válida. Isto não ocorreria se o valor do mês fosse maior que 12 ou que, por exemplo, se o mês fosse junho e o dia fosse 31.

```
    Variaveis
        Caracter Data, d, s
        Numerico Dia, Mes, Ano
        Logico Verifica, v
    Funcao VerificaData(d)
    Inicio
        Verifica = verdadeiro
        s = Parte(d, 1, 2)
        Mes = Valor(s)
        Se Mes <= 12 entao
            s = Parte(d, 4, 2)
            Dia = Valor(s)
            Se ((Mes = 4) ou (Mes = 6) ou (Mes = 9) ou (Mes = 11)) e (Dia > 30) entao
                Verifica = falso
            Senao
                Se Mes = 2 entao
                    s = Parte(d, 7, 2)
                    Ano = Valor(s)
                    Se (Resto(Ano, 4) = 0) e (Dia > 29) entao
                        Verifica = falso
                    Senao
                        Se Dia > 28 entao
                            Verifica = falso
                        Fim_se
                    Fim_se
                Senao
```

```
                        Se Dia > 31 entao
                            Verifica = falso
                        Fim_se
                    Fim_se
                Fim_se
            Senao
                Verifica = falso
            Fim_se
            Retornar Verifica
Fim
```

7. Imagine agora a seguinte questão. O interpretador ILA ao interpretar um comando *Faca enquanto* espera encontrar depois da palavra *Faca* um espaço em branco e a palavra *enquanto*. Desta forma se o digitador digitasse mais de um espaço em branco entre as duas palavras, o interpretador acusaria erro. Como é possível que o digitador possa, inadvertidamente, digitar mais de um espaço em branco entre duas palavras, é interessante que, antes da interpretação, os espaços em branco em excesso sejam suprimidos. O programa abaixo realiza esta ação e, para tal, faz uso da função *Delete* do exemplo 1 desta seção.

```
//Programa que exclui espaços em branco em excesso entre duas palavras de uma frase dada.
Variaveis
    Caracter s
    Numerico i, c, p, n, j
Inicio //Programa principal
    Escrever "Digite a cadeia"
    Ler s
    c = Comprimento(s)
    i = 1
    Faca enquanto i < c
        j = i
        Faca enquanto (Parte(s, i, 1) = " ") e (Parte(s, i + 1, 1) = " ")
            i = i + 1
        Fim_enquanto
        Se i > j entao
            i = i - j
            s = Delete(s, j + 1, i)
            c = Comprimento(s)
            i = i + 1
        Fim_se
        i = i + 1
    Fim_enquanto
    Escrever s
Fim
```

8. A questão a seguir é bem interessante. Trata de um programa que determine o *dígito verificador* de um número de uma conta corrente, de um número de matrícula de um estudante de uma escola, etc.. Este dígito verificador, como já foi dito acima, serve para a prevenção de possíveis erros de digitação. Por exemplo, se a matrícula 30245-7 fosse digitada erroneamente como 39245-7, o erro seria detectado pois o dígito verificador da conta 39245 seria 6 e não 7. Existem vários métodos para a determinação do dígito verificador. Um deles é dado pelo seguinte algoritmo.

1. Multiplica-se os números correspondentes aos dígitos da conta, da direita para esquerda, por 2, por 3, etc..

2. Soma-se os produtos obtidos no item 1.

3. Determina-se o resto da divisão da soma obtida no item 2 por 11.
4. Subtrai-se de 11 o resto obtido no item 3
5. Se o valor obtido no item 4 for 10 ou 11 o dígito verificado é igual a zero; senão, o dígito é o valor obtido no item anterior.

Por exemplo, se o número da conta for 30245, temos

1. 5x2 = 10, 4x3 = 12, 2x4 = 8, 0x5 = 0, 3x6 = 18
2. 10 + 12 + 8 + 0 + 18 = 48
3. Resto(48, 11) = 4
4. 11 - 4 = 7
5. Dígito verificador = 7.

A implementação deste algoritmo em ILA redundaria no seguinte programa.

```
Variaveis
        Caracter Mat, s, Digito
        Matriz numerico Digitos[15]
        Numerico Dv, i, c, Aux, j, n
//Função que armazena num vetor os números correspondentes aos dígitos de um inteiro dado
Funcao ExtraiDigitos(s)
Inicio
        c = Comprimento(s)
        Para i = 1 ate c
        Digito = Parte(s, i, 1)
        Digitos[i] = Valor(Digito)
        Proximo
Fim
//Função que transforma um número de um algarismo no algarismo correspondente.
Funcao NumeroParaDigito(n)
Inicio
        Faca caso
        Caso n = 0:
        Digito =  "0"
        Caso n = 1:
        Digito =  "1"
        Caso n = 2:
        Digito =  "2"
        Caso n = 3:
        Digito =  "3"
        Caso n = 4:
        Digito =  "4"
        Caso n = 5:
        Digito =  "5"
        Caso n = 6:
        Digito =  "6"
        Caso n = 7:
        Digito =  "7"
        Caso n = 8:
        Digito =  "8"
        Caso n = 9:
        Digito =  "9"
        Fim_caso
        Retornar Digito
```

```
Fim
Inicio //Programa principal
        Escrever "Digite a matricula"
        Ler Mat
        ExtraiDigitos(Mat)
        Aux = 0
        j = 2
        Para i = c ate 1 passo -1
                Aux = Aux + Digitos[i] * j
                j = j + 1
        Proximo
        Aux = Resto(Aux, 11)
        Dv = 11 - Aux
        Se (Dv = 10) ou (Dv = 11) entao
                Dv = 0
        Fim_se
        Mat = Mat + "-" + NumeroParaDigito(Dv)
        Escrever "Matricula = ", Mat
Fim
```

Observe que poderíamos ter usado a função *IntParaCarac* construída no exemplo 3 da seção 8.3. Optamos por escrever uma nova função pelo fato de que os inteiros a serem transformados aqui têm apenas um algarismo.

8.5 Exercícios propostos

1. Uma palavra é *palíndroma* se ela não se altera quando lida da direita para esquerda. Por exemplo *raiar* é *palíndroma*. Escreva um programa que verifique se uma palavra dada é palíndroma.

2. Um dos recursos disponibilizados pelos editores de texto mais modernos é a determinação do número de palavras de um texto. Escreva um programa que determine o número de palavras de uma frase dada.

3. O exercício 20 da seção 6.4 solicitava um programa que convertesse um número dado no sistema decimal para o sistema binário. Pela limitação do sistema em tratar números inteiros, uma solução que tratasse a conversão como sendo do tipo *Numerico* seria limitada pelo fato de que, por exemplo, 32 no sistema binário é 100000, que não pode ser tratado como inteiro. Escreva um programa para a conversão citada, tratando o valor em binário como uma cadeia de caracteres.

4. Escreva um programa que converta um número do sistema binário, dado como uma cadeia de zeros e uns, para o sistema decimal de numeração.

5. Reescreva o programa apresentado no exemplo 8 da seção 8.4 de tal modo que ele possa, além de gerar dígitos verificadores, verificar se uma conta dada (incluindo o dígito verificador) foi digitada incorretamente, incorreção esta detectada pelo dígito verificador.

6. Os editores de texto possuem um recurso que permite o usuário substituir uma subcadeia de um texto por outra cadeia de caracteres. Escreva um programa que realize esta ação numa frase dada.

7. As companhias de transportes aéreos costumam representar os nomes dos passageiros no formato *último sobrenome/nome*. Por exemplo, o passageiro Carlos Drumond de Andrade seria indicado por Andrade/Carlos. Escreva um programa que receba um nome e o escreva no formato acima.

8. As normas para a exibição da bibliografia de um artigo científico, de uma monografia, de um livro texto, etc., exigem que o nome do autor seja escrito no formato *último sobrenome, sequência das primeiras letras do nome e dos demais sobrenomes, seguidas de ponto final*. Por

exemplo, Antônio Carlos Jobim seria referido por Jobim, A. C.. Escreva um programa que receba um nome e o escreva no formato de bibliografia.

9. É muito comum que os títulos de documentos como avisos, declarações, atestados, etc., apareçam em letras maiúsculas separadas por um espaço em branco. Escreva uma função que receba uma palavra e a retorne no formato acima.

10. Escreva uma função que gere *logins* para usuários de um sistema de computação, sendo o *login* composto pelas letras iniciais do nome do usuário.

Observação

Para obter o caderno das soluções dos exercícios propostos basta encaminhar e-mail para jaime@ccen.ufal com assunto RESPOSTAS LIVRO ILA e contendo NOME, CATEGORIA (docente, discente, autodidata), CIDADE/ESATADO e, se for o caso, INSTITUIÇÃO/CURSO.

Índice remissivo

Bibliografia

Dijkstra, E. W., *A Discipline of Programiming*. Prentice-Hall. New Jersey, 1975.

Evaristo, J., *Programando com Pascal*, Edição Digital, Maceió, disponível em www.ic.ufal.br/professor/jaime.

Evaristo, J., *Aprendendo a Programar Programando em Linguagem C*. Edição Digital, Maceió, disponível em www.ic.ufal.br/professor/jaime. .

Evaristo, J., Perdigão, E., *Introdução `a Álgebra Abstrata*. Editora da Universidade Federal de Alagoas (EDUFAL). Alagoas, 2002.

Knuth, D. E., *The Art of Computer Programming*, volume 2, *Seminumerical Algorithms* Addison-Wesley Publishing Company. USA, 1988.

Kowaltowski, T. & Lucchesi, C., *Conceitos Fundamentais e Teoria da Computação*. Anais do II WEI. Minas Gerais, 1994

Norton, P., *Introdução à Informática*. Makron Books. Sãp Paulo, 1996.

Rangel, J. L., *Os Programas de Graduação em Linguagens de Programação*. Anais do IIWEI. Minas Gerais, 1994.

Szwarcfiter, J. L. & Markenzon, *Estruturas de Dados e seus Algoritmos*. LTC Editora. Rio de Janeiro, 1994.

Wirth, N., *Algorithms & Data Structures*. Prentice-Hall. New-Jersey, 1986.