

ALGORITMOS:
APRENDA A PROGRAMAR

PRINCÍPIOS DA **PROGRAMAÇÃO**

JORGE LUIZ SURIAN



LISTA DE FIGURAS

Figura 2.1 – Ábaco	6
Figura 2.2 – Ábaco	7
Figura 2.3 – Pascaline	8
Figura 2.4 – Ábaco	9
Figura 2.5 – Ábaco	10
Figura 2.6 – Charles Babbage	11
Figura 2.7 – Lady Lovelace	11
Figura 2.8 – George Boole	12
Figura 2.9 – Tabuleiro do censo	13
Figura 2.10 – Válvula de Lee De Forest	14
Figura 2.11 – Mark I	14
Figura 2.12 – Sucessões do Mark I	15
Figura 2.13 – Colossus	15
Figura 2.14 – Arquitetura de von Neumann	16
Figura 2.15 – Transistor	17
Figura 2.16 – Terminal burro	18
Figura 2.17 – Altair 8800	19
Figura 2.18 – Steve Jobs e Steve Wozniak	20
Figura 2.19 – Algoritmo 1: Cálculo da área de um triângulo – Descrição Narrativa	29
Figura 2.20 – Algoritmo 1: Cálculo da área de um triângulo	31
Figura 2.21 – Exemplo de algoritmo	32
Figura 2.22 – Exemplo de português estruturado	44
Figura 2.23 – Programa em Java	45
Figura 2.24 – Programa em C	45
Figura 2.25 – Programa em Pascal	46
Figura 2.26 – Soma	46
Figura 2.27 – 12 meses	47
Figura 2.28 – Peso/ Altura ²	47
Figura 2.29 – Zero	48
Figura 2.30 – Algoritmo para elevar o número 3 ao quadrado de diferentes maneiras	56
Figura 2.31 – Algoritmo de elevar ao quadrado melhorado com função.	57
Figura 2.32 – Fluxograma: cálculo da área de um triângulo	59
Figura 2.33 – Elementos de Fluxograma	60

LISTA DE QUADROS

Quadro 2.1 – Relevância das informações	33
Quadro 2.2 – Tipos em Java	36
Quadro 2.3 – Tipos em C	37
Quadro 2.4 – Tipos em Pascal.....	38
Quadro 2.5 – Operadores	38
Quadro 2.6 – Operadores Relacionais.....	40

EXEMPLO

SUMÁRIO

2 PRINCÍPIOS DA PROGRAMAÇÃO	5
2.1 Introdução	5
2.2 Algoritmos: história, lógica de programação e conceitos introdutórios	5
2.2.1 Breve história da TI	5
2.2.1.1 3.000AC - Ábaco	5
2.2.1.2 1671 – Calculadora mecânica - Pascaline	7
2.2.1.3 1676 – Aperfeiçoamento da Pascaline - Leibniz	8
2.2.1.4 1801 – Primeira máquina mecânica programável	9
2.2.1.5 1820 – Evolução da máquina de Leibniz - Arithometer	10
2.2.2 Os pioneiros da Informática	10
2.2.3 A contribuição da Matemática	12
2.2.4 A indústria de hardware	13
2.2.5 Computadores de 1ª geração	14
2.2.5 Computadores de 2ª geração	17
2.2.6 Computadores de 3ª geração	18
2.2.7 Novas gerações de computadores	19
2.3 Linguagens de programação	20
2.3.1 Linguagens de programação de primeira geração	20
2.3.2 Linguagens de programação de segunda geração	21
2.3.3 Linguagens de programação de terceira geração	21
2.3.4 Linguagens de programação de quarta geração e posteriores	23
2.4 Introdução à lógica de programação	24
2.4.1 Algoritmos x Programas de computador	24
2.4.2 Conceituação de algoritmos	25
2.4.3 Análise – compreensão	25
2.4.4 Análise – estratégia	26
2.4.5 Projeto de programa	27
2.5 Algoritmos resolvendo problemas	28
2.5.1 O problema do cálculo da área de um triângulo	29
2.5.2 Retomando o cálculo da área de um triângulo	30
2.5.3 Variáveis e sua tipologia	33
2.5.3.1 Dados, informações e variáveis	33
2.5.3.2 Definindo variáveis	34
2.5.4 Operadores aritméticos	38
2.5.5 Operadores relacionais	39
2.5.6 O porquê das variáveis	40
2.5.7 Variáveis nas linguagens de programação	41
2.5.8 Regras na declaração de variáveis	42
2.6 Instruções de entrada e saída	43
2.6.1 Saída	43
2.6.2 Entrada	44
2.6.3 Exemplificando entradas e saídas	44
2.6.4 Problemas resolvidos	46
2.6.5 Problemas propostos	48
2.7 Instruções e Funções	56
2.8 Fluxograma	59
REFERÊNCIAS	61

2 PRINCÍPIOS DA PROGRAMAÇÃO

2.1 Introdução

Nesse capítulo teremos inicialmente uma breve história da Tecnologia da Informação, viajando a um passado remoto até chegarmos aos dias de hoje e aos primeiros aspectos introdutórios de programação de computadores.

A Parte 1 deste capítulo apresenta uma breve história da TI.

A Parte 2 traz uma breve história das linguagens de programação, uma vez que os algoritmos são concebidos exatamente com a finalidade de nos auxiliar a escrever programas nas mais diversas linguagens de programação.

A Parte 3 versa sobre lógica de programação propriamente dita, com a apresentação dos primeiros conceitos técnicos.

A Parte 4 mostra um primeiro exemplo de algoritmos aplicados à solução de problemas.

A Parte 5 trata das variáveis e seus mais diversos tipos.

Por fim, a Parte 6 apresenta um grupo muito especial de instruções algorítmicas: as instruções de entrada e saída.

2.2 Algoritmos: história, lógica de programação e conceitos introdutórios

2.2.1 Breve história da TI

2.2.1.1 3.000AC - Ábaco

Quando usamos os modernos computadores atuais, vez por outra nos perguntamos: De onde veio tudo isso? Voltando muito atrás no tempo, vamos encontrar um dispositivo que nos ajudou a fazer os primeiros cálculos, inventado ainda por volta de 3000 a.C. Estamos falando do ábaco.

Composto por inúmeras contas, engenhosamente divididas de forma a permitir a execução de cálculos, o ábaco é um poderoso instrumento nas mãos de

quem conhece como operá-lo, e de nenhuma valia para quem desconhece sua operação.

Na verdade, a operação de um ábaco só é possível para quem sabe manipular as contas de forma organizada, para que essas representem as mais diversas operações. Esse é um processo **algorítmico**, na medida em que se trata de um processo repetitivo que, ao ser concluído, permite a obtenção do resultado procurado.

O ábaco é, portanto, o primeiro dispositivo manual de cálculo conhecido. Serve, basicamente, para representar números no sistema decimal e permite realizar operações de soma, multiplicação, subtração e divisão! Como se vê, não é pouca coisa para uma “ferramenta” com 5.000 anos de idade.

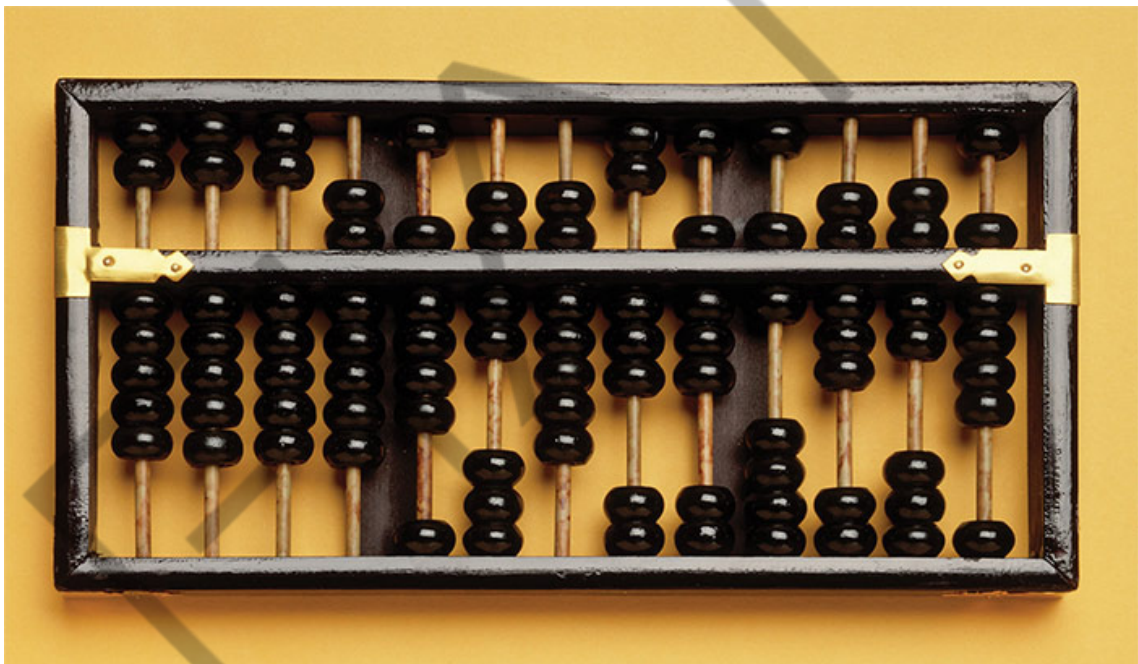


Figura 2.1 – Ábaco
Fonte: Google Images (2015)

1614 – Criação da máquina superior ao ábaco. Para se ter ideia da longevidade desse instrumento, apenas em 1614, o escocês John Napier constrói uma máquina superior que permitia o cálculo de logaritmos usando bastões que faziam multiplicações e divisões de forma automática.

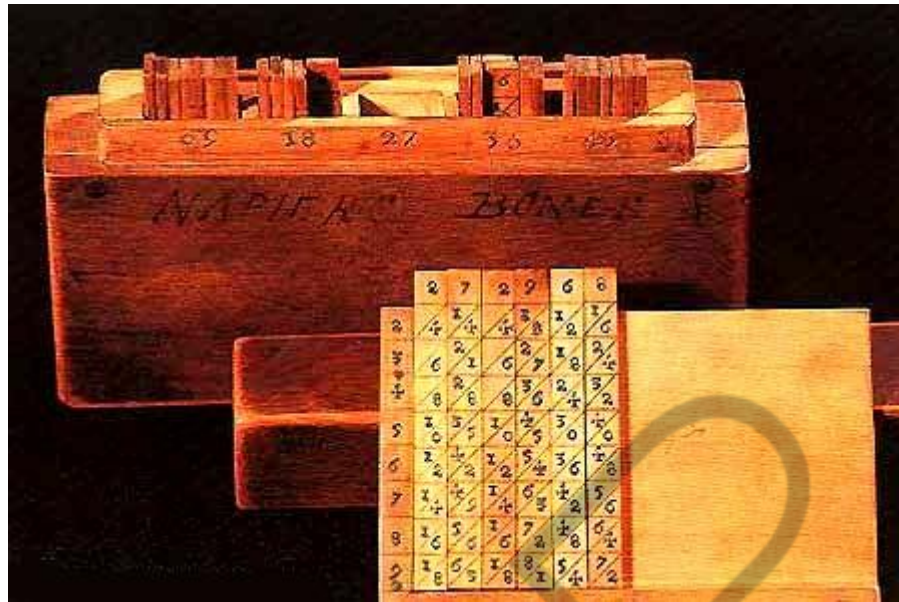


Figura 2.2 – Ábaco
Fonte: Google Images (2015)

2.2.1.2 1671 – Calculadora mecânica - Pascaline

Não muito tempo depois, em 1671, o francês Blaise Pascal a primeira calculadora mecânica capaz de fazer somas e subtrações. Chamada de Pascaline, esta funcionava a base de engrenagens e foi um grande passo na direção da automação do processo de cálculo. A linguagem de programação Pascal, base de produtos célebres na história da TI, como o MS-Pascal, o Turbo Pascal e o Delphi, tem seu nome em homenagem a esse célebre matemático e pensador francês.



Figura 2.3 – Pascaline
Fonte: Google Images (2015)

2.2.1.3 1676 – Aperfeiçoamento da Pascaline - Leibniz

Bem pouco tempo depois, ainda em 1676, foi criada pelo alemão Gottfried Wilhelm Leibniz uma máquina que era um aperfeiçoamento da Pascaline. Além das operações de soma e subtração, a máquina de Leibniz também efetuava multiplicações através de somas consecutivas. Acabou não sendo convenientemente divulgada, portanto ficou pouco conhecida.

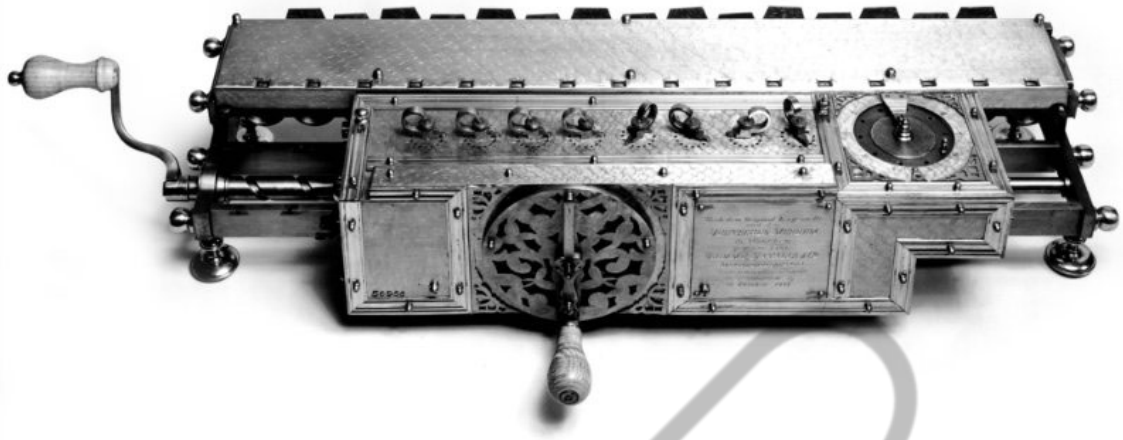


Figura 2.4 – Ábaco
Fonte: Google Images (2015)

2.2.1.4 1801 – Primeira máquina mecânica programável

Contudo, ainda demoraria bastante para ser criada uma máquina que aceitasse alguma programação. Isso foi ocorrer apenas em 1801, quando o francês Joseph-Marie Jacquard cria a primeira máquina mecânica programável, controlada por meio de cartões perfurados, que controlavam a confecção e o desenho nos tecidos.



Figura 2.5 – Ábaco
Fonte: Google Images (2015)

Todavia, mesmo na área de cálculo ainda havia muito a fazer.

2.2.1.5 1820 – Evolução da máquina de Leibniz - Arithometer

Projetada e construída em 1820, pelo francês Charles Xavier Thomas, foi criada a primeira máquina que realizava as quatro operações aritméticas básicas. Tratava-se de uma evolução da máquina de Leibniz, que foi batizada de Arithometer, provavelmente inaugurando algo que parece caminhar junto com a TI – Tecnologia da Informação, os mneumônicos, abreviações ou junções de nomes (nesse caso Ari, de Aritmética e Thometer, uma variação a partir do nome do inventor).

2.2.2 Os pioneiros da Informática

Criado em 1833 pelo inglês Charles Babbage, o “Calculador analítico” é a primeira máquina realmente programável, por isso Babbage é conhecido como “Pai da Informática”. Sua máquina dispunha de programa, memória, unidade de controle

e periféricos de saída. Talvez, o mais correto seria dar o título de pai do hardware de TI, pois a parte de *software* foi criada por outra pessoa.

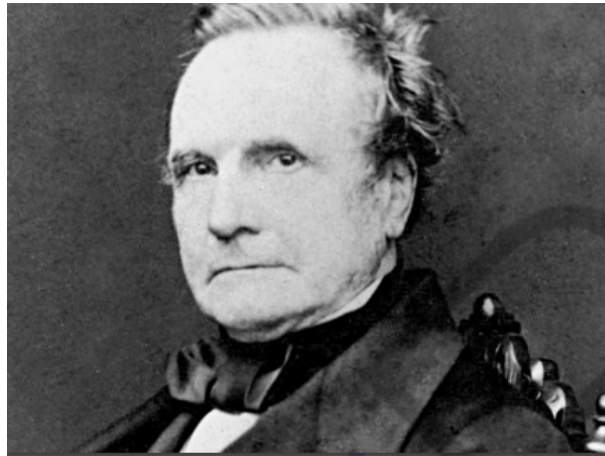


Figura 2.6 – Charles Babbage
Fonte: Google Images (2015)

De fato, os primeiros trabalhos de linguagem de programação, ou programas, foram criados por Ada Augusta Byron King (condessa de Lovelace), que eram utilizados na máquina analítica de Charles Babbage.



Figura 2.7 – Lady Lovelace
Fonte: Google Images (2015)

Lady Lovelace, como era conhecida, foi a única filha legítima do poeta britânico Lord Byron, e é reconhecida como a primeira programadora de toda a história. Durante um curto período de nove meses, entre os anos de 1842 e 1843, em que esteve envolvida com o projeto de Babbage, ela desenvolveu os algoritmos que permitiriam à máquina computar os valores de funções matemáticas (como a

sequência de Bernoulli), além de publicar uma coleção de notas sobre a máquina analítica.

Ada de Lovelace foi uma das poucas pessoas que realmente entenderam os conceitos envolvidos no projeto de Babbage. Durante o processo de tradução de uma publicação científica italiana sobre o projeto de Babbage, ela incluiu algumas notas de tradução que constituem o primeiro programa escrito na história da humanidade.

Muito justamente, Ada também foi eternizada com o nome de uma linguagem de computador que leva seu nome.

2.2.3 A contribuição da Matemática

A matemática, naturalmente, não poderia ficar à parte dessa evolução, uma vez que esta foi a força motriz da TI desde o princípio. Coube ao inglês George Boole, em 1847, a criação de novos raciocínios aplicáveis ao estudo da computação, baseados na lógica binária.

A lógica de programação atual ainda contém seus conceitos baseados nos estudos de Boole.

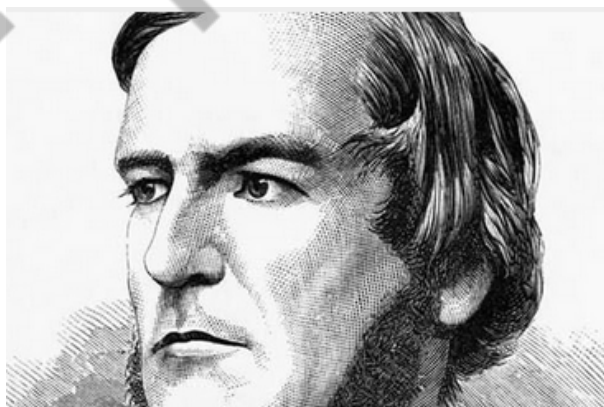


Figura 2.8 – George Boole
Fonte: Google Images (2015)

2.2.4 A indústria de hardware

Criada em 1885 pelo norte-americano Herman Hollerith, a “Tabuladora do Censo” foi utilizada no recenseamento, mas tornou-se sinônimo de recibo de pagamento dos trabalhadores. Basicamente, acumulava e classificava informações, trabalhando com cartões perfurados, sendo uma evolução da máquina de Jacquard. Foi usada em muitos outros trabalhos, como o processamento de folha de pagamento, sendo aí considerada um dos primeiros passos na direção de um computador programável.



Figura 2.9 – Tabuleiro do censo
Fonte: Google Images (2015)

Um ponto bastante interessante a observar é que a empresa de Hollerith se fundiu com outras quatro empresas, dando origem à Computing Tabulating Recording Corporation, que sob a presidência de Thomas J. Watson foi renomeada para IBM.

Muito pouco tempo depois, em 1905, foram criadas as válvulas, pelo norte-americano Lee De Forest. Essas seriam os principais componentes da primeira geração de computadores, algum tempo mais tarde.

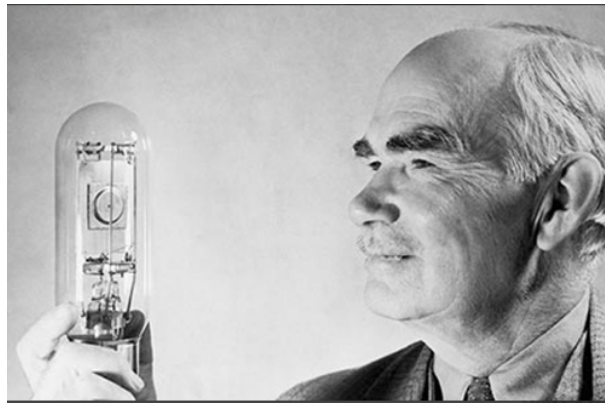


Figura 2.10 – Válvula de Lee De Forest
Fonte: Google Images (2015)

2.2.5 Computadores de 1ª geração

Os computadores da chamada primeira geração foram criados prioritariamente para uso bélico, governamental, de grandes indústrias e de centros de pesquisa. Seus “usuários” possuíam formação muito especializada e eram muito mais cientistas e pesquisadores do que qualquer outra função. O próprio termo usuário soa algo estranho, pois esses pioneiros se comunicavam em linguagem de máquina com esses equipamentos, ou seja, algo muito anterior aos assemblers, que são mneumônicos da língua inglesa que funcionam como instruções de computador. A linguagem de máquinas usada era um conjunto de “zeros e uns”, praticamente indecifrável atualmente.

Em 1937, o norte-americano Howard Aiken cria o primeiro computador **eletromecânico**, denominado Mark I.



Figura 2.11 – Mark I
Fonte: Google Images (2015)

Em 1945, os primeiros computadores e suas salas lembravam mais ambientes de filmes de terror – com suas válvulas, programação como os fios e outros apetrechos – do que a moderna computação.

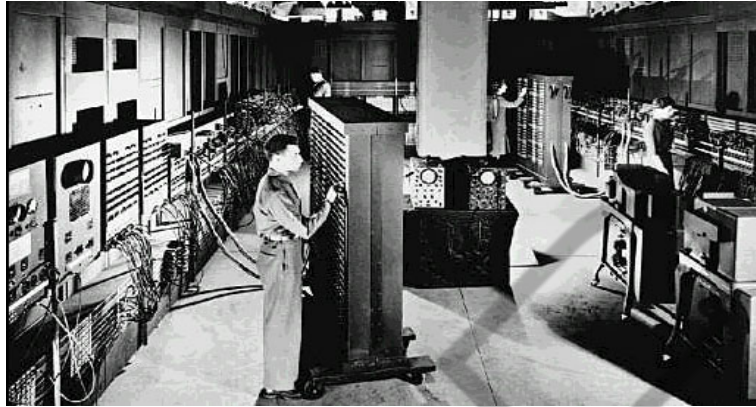


Figura 2.12 – Sucessões do Mark I
Fonte: Google Images (2015)

O Mark I e seus sucedâneos II, III e IV foram os primeiros passos na direção do mainframe, que tomou sua primeira forma com o computador Colossus, criado em 1943, a partir das ideias do inglês Alan Turing.

Trata-se do primeiro computador **eletrônico mecânico programável**. O Colossus trabalhava com símbolos perfurados em fita de papel, que eram lidos por célula fotoelétrica. Comparava a mensagem cifrada com os códigos conhecidos até encontrar uma coincidência. Ele processava 25.000 caracteres por segundo.

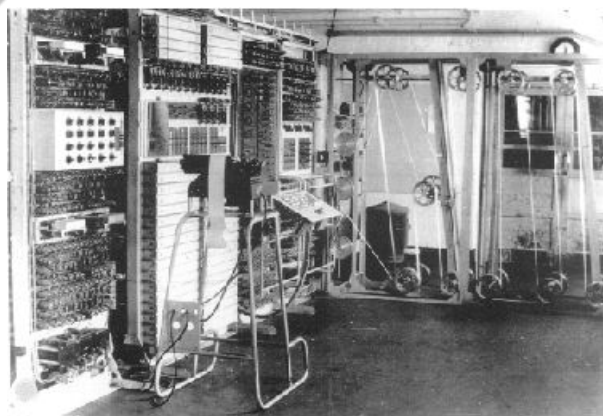


Figura 2.13 – Colossus
Fonte: Google Images (2015)

Muito pouco tempo depois, em 1946, foi criado pelos americanos John Mauchly e J. Presper Eckert o **primeiro computador eletrônico digital** de grande porte, utilizando-se de válvulas. Seu nome? **Eniac**.

Suas principais características eram:

- Operava na base dez, não binário.
- 18.000 válvulas – 175 Kw de potência – ocupava 270 m².
- 5.000 operações por segundo.
- Pesava 30 toneladas.
- Aplicação principal: cálculo balístico.
- Em 1947, pelo húngaro John von Neumann, com base em seus estudos feitos sobre o ENIAC, surge a definição do esquema básico de funcionamento dos computadores atuais. A chamada Arquitetura de von Neumann, é a seguinte:

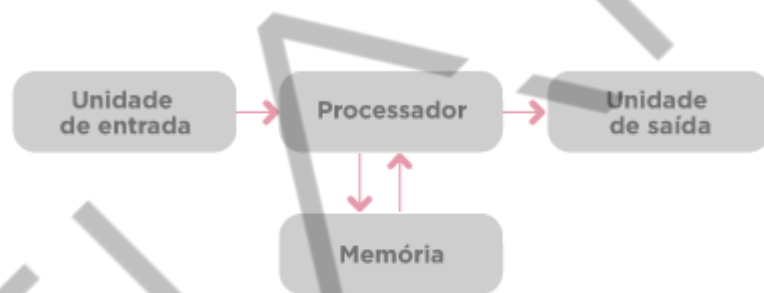


Figura 2.14 – Arquitetura de von Neumann
Fonte: FIAP (2015)

- Ainda em 1947, surge o **EDVAC – Eletronic Discrete Variable Computer**, que se utilizava da notação binária pura, o que simplificou enormemente a construção das ULA (Unidade Aritmética e Lógica).
- A memória era construída com lâmpadas de mercúrio que excitadas por impulso elétrico enviavam sinais para frente e para trás, possibilitando leituras de 0 ou 1 (ligado/desligado).
- Marca-se aqui a separação das funções de operador e programador, cabendo ao Operador a inserção dos comandos no painel e ao programador a resolução problemas e criação de novos programas. O uso do computador é otimizado a partir de então.

2.2.5 Computadores de 2ª geração

É criado em 1947, na Universidade de Stanford (EUA), o transistor, usado somente a partir da década posterior. O uso comercial em grande escala, todavia, só ocorreria no final da década de 1950. É o principal componente dos computadores de 2ª Geração. A Raytheon lança, ainda em 1948, o **Transistor** comercial.



Figura 2.15 – Transistor
Fonte: Google Images (2015)

Surge em 1952, no Bell Laboratories (EUA), o computador usando transistor (válvula em miniatura), que solucionou muitos problemas técnicos, tais como a queima de válvulas, aquecimento – foi aumentada a velocidade e diminuído o tamanho dos computadores. Nota-se que a substituição de uma simples válvula queimada era um processo trabalhoso e lento.

Nessa geração de computadores, os usuários ainda necessitavam de formação específica e o uso dos computadores era muito restrito, exclusivo para grandes empresas, organismos do governo e centros de pesquisa. É o tempo do “terminal burro”.



Figura 2.16 – Terminal burro
Fonte: Google Images (2015)

Criado em 1949, por Mauchly Computer Corporation, o UNIVAC é o primeiro computador **eletrônico disponível comercialmente**. Sua primeira aplicação: *processamento das eleições*. Trata-se também do primeiro computador adquirido no Brasil, pelo IBGE para apoio em estatísticas.

O UNIVAC - Universal Automatic Computer era uma máquina eletrônica capaz de armazenar programas de computadores através de uma fita magnética de alta velocidade, ao invés dos cartões perfurados. Usava o código BCD de 6 Bits com 1 de paridade.

Algum tempo depois, surgem as unidades de disco rígido que representaram uma revolução no armazenamento de dados, que antes eram armazenados em fitas. Eram caríssimas, portanto incentivaram o uso da compactação dos dados. O famoso “bug do milênio” em grande parte ocorreu por isso, pois era “vantajoso” armazenar “56” e não “1956”.

2.2.6 Computadores de 3ª geração

No ano de 1964, a IBM lança a série 360, permitindo o uso desta tecnologia em empresas de grande e médio portes. A base tecnológica passou a ser o Chip ou circuito integrado (compactação dos transistores em placas de silício).

2.2.7 Novas gerações de computadores

Muito antes da fundação da Intel, Gordon Moore vaticinou o que viria a ser uma das mais interessantes “leis” da TI.

Lei de Moore: “O número de transistores dos chips terá um aumento de 100%, pelo mesmo custo, a cada período de 18 meses.” (Gordon Moore, presidente da Intel, escreveu esse artigo em 1965).

Criado em 1975, baseado na CPU da Intel 8080, surge o ALTAIR 8800, que é o primeiro **computador pessoal portátil**, produzido industrialmente para venda em massa, criado pelos norte-americanos Ed Roberts, Forest M. Mims III, Stan Cagle e Robert Zaller.



Figura 2.17 – Altair 8800
Fonte: Google Images (2015)

Se não chegou a ser um grande sucesso comercial, motivou a criação de inúmeros outros microcomputadores, entre os quais se destaca o Apple II.

Lançado em 1976, por Steve Jobs e Steve Wozniak (fundadores da Apple Corp.), foi o primeiro microcomputador pessoal a ter sucesso comercial.



Figura 2.18 – Steve Jobs e Steve Wozniak
Fonte: Google Images (2015)

Em sequência, surgem os minicomputadores e derivados, até chegarmos aos microcomputadores e seus derivados, como o notebook. Trata-se de uma indústria em evolução constante, sendo impossível prever aonde chegará.

2.3 Linguagens de programação

As linguagens de programação eram separadas, para fins didáticos, em “gerações”. Atualmente, esse tipo de classificação perdeu o sentido, mas para efeito histórico relativo e para conhecermos a história das linguagens de programação, parece bastante adequado usarmos (ainda) esse antigo conceito.

2.3.1 Linguagens de programação de primeira geração

As linguagens de primeira geração (Assembly) (ou 1GL) referem-se ao código de máquina. É a única linguagem que um microprocessador pode entender nativamente. O código de máquina não pode ser escrito ou lido por um editor de texto e, portanto, é raramente usado por uma pessoa diretamente. Foi usada apenas na programação dos primeiros computadores criados. Naturalmente, era totalmente dependente do processador do computador para o qual se escrevia o programa.

2.3.2 Linguagens de programação de segunda geração

A linguagem de segunda geração (ou 2GL) é a linguagem Assembler. É considerada de segunda geração, pois embora não seja uma linguagem nativa do microprocessador, um programador que use a linguagem Assembler ainda deve compreender as características da arquitetura do microprocessador (como registradores e instruções).

Há muitas controvérsias quanto a essa classificação, pois muitos autores entendem que Assembler e Assembly fazem parte de uma mesma geração de linguagens de programação. Fazendo essa ressalva, vamos utilizar essa nomenclatura para nos referir ao Assembly (1GL) e ao Assembler (2GL).

2.3.3 Linguagens de programação de terceira geração

Também há controvérsias quanto ao que seria a primeira geração de linguagens de alto nível (3GL em nossa classificação). Alguns autores entendem que FORTRAN, COBOL e ALGOL seriam de uma geração, enquanto Pascal e C seriam de uma geração posterior, em nome de uma maior estruturação dessas linguagens. Novamente ficaremos com os autores que entendem ser de terceira geração as linguagens de alto nível escritas em derivações da língua inglesa, restritas a criação de programas na própria linguagem de programação. Assim, todas essas linguagens citadas seriam de terceira geração, embora o tempo de criação entre as mesmas tenha sido substancialmente grande, como veremos mais adiante.

Embora o FORTRAN tenha sido a primeira linguagem de terceira geração a ser usada, a primeira linguagem criada não foi esta. Criada em 1946, nomeada Plankalkül por Konrad Zuse, engenheiro alemão, esta teria sido a primeira linguagem criada de alto nível. A polêmica existente resulta da publicação do trabalho de Zuse apenas em 1972.

Zuse queria desenvolver um método sofisticado para o desenvolvimento das tarefas que a equipe deve fazer. Não existem programas criados para essa linguagem de computador. Sua máquina computadora S2 é considerada o primeiro computador controlado por processamento (e não o Mark I, como se popularmente

diz), o qual foi usado para ajudar a desenvolver os mísseis Henschel Hs 293 e Henschel Hs 294, precursores dos modernos mísseis de cruzeiro.

Feita essa ressalva, considera-se que a primeira linguagem de alto nível foi criada em 1954 pelo norte-americano John Backus, da IBM, que a denominou FORTRAN (Formula Translation)

Pouco tempo depois, precisamente em 1958, surgem o ALGOL – origem da maioria das linguagens modernas, e o LISP, criada por John MacCarthy, norte-americano, no MIT.

Em 1959, surge o COBOL, com uma série de melhorias na entrada e saída de dados. Criada por uma equipe de programadores, esta é a linguagem mais antiga em uso. Foi concebida a partir dos esforços pioneiros da norte-americana, Grace Murray Hopper, que foi analista de sistemas da Marinha dos Estados Unidos nas décadas de 1940 e 1950, chegando à patente de Almirante.

Foi ela quem criou a linguagem de programação Flow-Matic, que serviu como base para a criação do COBOL. Por isso, é considerada a “autora” do COBOL. Dentre outras realizações da Almirante, está a criação do primeiro compilador, e COBOL, sendo a primeira linguagem de programação de computadores a se aproximar da linguagem humana ao invés da linguagem de máquina. Tanto FORTRAN, como ALGOL e LISP, eram mais próximas de uma linguagem de máquina do que da linguagem humana. De fato, COBOL utiliza palavras da língua inglesa, mas busca uma maior proximidade com a construção humana das frases do que com algoritmos.

A almirante Hooper ficou conhecida popularmente como “a velha do COBOL”, todavia ela não participou efetivamente na criação da linguagem COBOL, mas sim um subcomitê proposto numa reunião no Pentágono em maio de 1959.

Este subcomitê desenvolveu as especificações da linguagem COBOL. Ele era formado por: William Selden e Gertrude Tierney da IBM; Howard Bromberg e Howard Discount da RCA; Vernon Reeves e Jean E. Sammet da Sylvania Electric Products.

A concepção de Hopper é de que havia a necessidade de se criar uma linguagem orientada para negócios comuns, desde então, deu origem ao acrônimo

COBOL (**CO**mmon **B**usiness **O**riented **L**anguage). Tudo foi feito sem definir os comandos para minimizar melindres entre os técnicos das empresas convocadas a participar da criação de um denominador comum entre todos os fabricantes existentes na época.

Grace Hopper participou contribuindo com a abertura dos comandos FLOW-MATIC.

Posteriormente, em 1964, surge o BASIC – Beginners All-purpose Symbolic Instruction Code, que foi criado originalmente para ensino, mas acabou por se tornar um dos padrões de desenvolvimento.

Bill Gates (primeiro à esquerda na foto histórica com os fundadores da Microsoft) ganhou um prêmio com a versão que criou (MBASIC, sendo o M de Microsoft, ainda sem o indefectível “S”) e que teve sua versão de maior sucesso criada em 1991.

Em 1967, surge o Simula 67, que introduz os conceitos de orientação a objetos, que muitos anos depois se tornaria o padrão a ser seguido por quase todas as linguagens de programação.

Pouco depois, em 1970, surge o Pascal, criado pelo húngaro Niklaus Wirth, que ganharia enorme popularidade na década de 1980, a partir da implementação do Turbo Pascal, pela Borland.

Em 1973, surge a linguagem C, aproveitando a linguagem “BCPL”, criada por Dennis Ritchie, da AT&T.

2.3.4 Linguagens de programação de quarta geração e posteriores

São linguagens de programação que podem criar programas em outras linguagens, que são voltadas a pesquisas, ou que por meio de interfaces permitem a criação de códigos sem a digitação efetiva de comandos.

Atualmente, faz pouco sentido buscar compreender hardware e software pelo conceito de geração, dadas as especificidades e intercorrências entre as influências sofridas e/ou causadas por determinada aplicação. Assim, a partir da 4ª geração de linguagens, não parece fazer muito sentido classificar tecnologias por geração.

São especialmente notáveis algumas situações, que passamos a enumerar:

Linguagens de interface com banco de dados: surgiram com o intuito de acessar algum banco de dados, habitualmente baseados no modelo relacional. Durante algum tempo contaram com mais usuários que as tradicionais linguagens de programação (COBOL, C, Pascal e BASIC). No entanto, foi um fenômeno passageiro, que teve início em meados da década de 1980 e tendo sido encerrado antes do final da década de 1990. O maior destaque foi da Nantucket Clipper, mas Data Ease, Data Flex, FoxPro, FORMS estão entre várias outras linguagens de acesso de forma proprietária a bancos de dados.

Linguagens orientadas a objetos: destacam-se o Java, C#, Visual BASIC e Delphi. Atualmente, Java e C# dominam amplamente o mercado de desenvolvimento de software.

2.4 Introdução à lógica de programação

Quando observamos a palavra “algoritmo”, parece soar um pouco estranho aos nossos ouvidos, que são muito mais familiarizados com palavras de origens latinas do que árabes, o que é o caso de “algoritmo”. Essa palavra deriva de Al-Khwarizmi, matemático persa do século IX, a quem se atribui a construção dos primeiros processos para realização de operações aritméticas, o que explica o porquê do nome de um matemático ter sido associado a um processo repetitivo que leva à solução de problemas complexos.

Por volta do século 3 a.C., outro matemático famoso, o grego Euclides, escreve seu famoso algoritmo para o cálculo do MDC (Máximo Divisor Comum). É, provavelmente, o primeiro algoritmo complexo e formalizado de que se tem notícia.

Algoritmo, por definição, é uma sequência de instruções ordenada com o objetivo de resolver um problema.

2.4.1 Algoritmos x Programas de computador

- Confunde-se frequentemente algoritmo e programa de computador, o que não tem razão de ser. Simplesmente, usamos de um computador para

executar um algoritmo. Isso se deve ao fato do computador ser mais rápido e preciso que um ser humano (Máquina Oligofrênica).

No entanto, primeiramente, o algoritmo deve ser transcrito para uma linguagem de programação qualquer, antes de podermos “usá-lo” num computador.

Em seguida, esse código escrito numa linguagem de programação (o “programa”) deve ser transformado num programa executável num computador, portanto esse programa deverá ser compilado (processo de validação da sintaxe utilizada) e linkeditado (processo de montagem do programa executável).

Serão apresentadas breves noções de fluxograma, observando-se sua pouca serventia em programas mais complexos, “causando mais mal do que bem” (GANE, 1983, p. 4).

2.4.2 Conceituação de algoritmos

Todo programa deveria ter um ciclo de desenvolvimento aproximado à lista:

- Análise do Problema.
- Projeto do Programa.
- Implementação.
- Testes.
- Verificação.

2.4.3 Análise – compreensão

Antes de partirmos para a solução de um problema, devemos entendê-lo o melhor possível.

Ao nos depararmos com um problema, o primeiro passo é compreendê-lo inteiramente. Lembre-se de que quem está perdido e põem-se a andar, tende a se perder ainda mais.

Assim, é uma boa estratégia perguntar:

- O que devemos descobrir, calcular ou obter? Qual é o objetivo do programa?
- Quais são os dados disponíveis? São suficientes para atendermos o que se quer?
- Quais são as condições necessárias e suficientes para resolver o problema?

É importante desenhar, rascunhar e usar de todos os artifícios para montar a “lógica” que nos leve à solução ou mesmo a um raciocínio intermediário.

A 1ª Etapa – Análise do Problema – consiste então na compreensão do problema e montagem de uma estratégia para sua resolução.

Você deve se lembrar, antes de qualquer outra coisa, que um algoritmo nada mais é que uma forma de representar a lógica que desejamos aplicar. Assim, antes de construir um algoritmo, precisamos definir alguma estratégia.

2.4.4 Análise – estratégia

Criar uma estratégia para a solução, passa pela resposta a algumas perguntas, um tanto elementares:

Já resolveu algum problema similar? Qual?

Se sim, a solução pode ser aproveitada por analogia, para referência ou por ser parte da solução do novo problema. Observe se será necessário introduzir elementos novos ou modificar os existentes.

Já se o problema for muito complexo, provavelmente pode ser fracionado em partes menores, de solução mais simples.

É possível enxergar o problema de outra forma, de modo que o entendimento se torne mais simples?

Lembre-se que descer uma escada de costas é bem mais complicado do que da maneira natural e chegamos sempre ao mesmo lugar.

Obtidas as respostas, não tente criar a solução inteira, antes de rascunhá-la!

Siga os passos:

Crie um algoritmo informal com as instruções que resolvam o problema ou que ao menos pareçam resolvê-lo.

Verifique se cada passo desse algoritmo está correto, simulando-o num papel (teste de mesa ou simulação).

Identifique os erros e os trate um por vez, sem nunca perder de vista o objetivo real do programa. Esse processo ordenado logicamente faz você aprender a desenvolver soluções.

Quando se elabora um esboço da solução do problema, para em seguida ir-se refinando essa solução, até chegar-se a uma sequência básica de operações que resolva o problema, usou-se uma das principais técnicas relacionadas à construção de algoritmos, intitulada Top-Down.

Essa técnica leva à geração de um pseudocódigo, que chamaremos de Português Estruturado (evite-se a expressão *portugol*).

O Português Estruturado, na opinião da maioria dos autores de livros de análise, programação ou algoritmos é muito mais eficaz que fluxogramas (estes são muito úteis em administração, que não é nosso caso).

2.4.5 Projeto de programa

Este nada mais é que o algoritmo gerado, que visará otimizar o “binômio tempo-espaco, isto é, visando obter um programa que apresente um tempo de execução mínimo e com o melhor aproveitamento de espaco de memória” (SALVETTI; BARBOSA, 1998).

a) Implementação

Quando codificamos um algoritmo numa linguagem de programação, o estamos implementando.

Você poderá, quando estiver cursando disciplinas de Linguagem de Programação, transformar todos os algoritmos vistos em nosso curso em **programas de computador**.

A implementação pode ser trivial, com a mera substituição de instruções algorítmicas em instruções em uma linguagem ou muito trabalhosa, dependendo da linguagem escolhida.

b) Testes

A fase de testes vem crescentemente sendo valorizada. Devemos lembrar que quem faz o programa, raramente testa seus pontos deficientes. Ora, isso ocorre porque se o analista tivesse percebido essas deficiências, provavelmente o programa não apresentaria esses problemas. Há várias técnicas para testes (caixa branca, caixa preta, entre várias outras).

Negligenciar essa fase é fatal. Numa prova, nem pensar.

c) Verificação do programa

A verificação do programa visa demonstrar que o algoritmo realmente resolve o problema proposto, qualquer que seja sua instância.

É nesta etapa que se constata se o Programa resolve todos os casos possíveis! Constatamos a sabedoria do velho provérbio inglês que diz “que todo problema complexo, aparentemente tem uma solução simples, que usualmente está errada [...]”.

Ao simularmos nossos algoritmos estamos fazendo exatamente esse processo. Todavia, quando codificado numa linguagem, mesmo que o algoritmo tenha sido simulado à exaustão, continuaremos obrigados a testar o programa que foi implementado.

2.5 Algoritmos resolvendo problemas

Admitamos um problema que possa ser solucionado através de um procedimento ordenado de instruções. Podemos comparar a questão como a confecção de um bolo a partir de uma receita. Um algoritmo nada mais que é um conjunto de instruções ordenadas logicamente de forma a resolver um problema.

Naturalmente, um algoritmo para ser executado em um computador deve ser transformado em um programa, numa linguagem qualquer. Antes de pensarmos em algoritmos, devemos montar nossa estratégia. A estratégia é muito mais importante

que o algoritmo e o programa, pois uma vez que a desenvolvermos, faremos sistemas (conjunto de programas) facilmente (ou nem tanto).

Assim temos que: uma estratégia → algoritmo → programa de computador → programa executável!

Todo programa recebe (ou parte de) dados, os transforma e gera resultados (outros dados processados), daí a expressão original **processamento de dados** que outrora definia a área de TI das organizações (os antigos CPD – Centro de Processamento de Dados).

2.5.1 O problema do cálculo da área de um triângulo

O cálculo da área de um triângulo é um processo bastante conhecido e bastante simples. Basta multiplicarmos o valor da base desse triângulo por sua altura, dividindo o resultado por dois para obter o valor da área.

Vamos a um exemplo: imaginemos que desejamos calcular a área de um triângulo. Ora, para isso precisaremos conhecer uma “fórmula” oriunda da geometria. Essa fórmula será, em nosso caso, a **estratégia**.

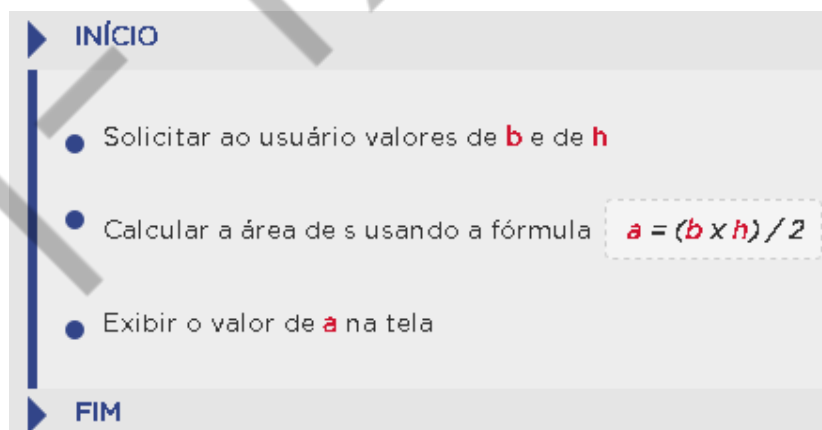


Figura 2.19 – Algoritmo 1: Cálculo da área de um triângulo – Descrição Narrativa

Fonte: FIAP (2015)

Nesse pequeno algoritmo usamos três variáveis. Variáveis são tão importantes que serão tratadas na próxima etapa desse trabalho. No momento,

basta que você admita que cada "letra", que representa um valor qualquer, seja nossa "variável".

Assim, a letra "b" é a primeira de nossas variáveis e representa a **base** do triângulo cuja área desejamos calcular. Observemos que se trata sempre de um valor numérico e como pode ser diferente de um caso para outro, por isso mesmo se trata de uma *variável*.

Analogamente, "h" faz o papel da **altura** e "a" faz o papel da **área**.

A chamada descrição narrativa descreve a solução do problema na forma de uma "receita", muito boa para seres humanos, mas péssima para computadores.

2.5.2 Retomando o cálculo da área de um triângulo

A solução anterior, embora bastante clara, exigiria que computadores processassem uma linguagem humana, ou algo próximo disso. Infelizmente com as tecnologias ainda estamos muito longe disso. Talvez não tão longe assim, quando pensamos em Siri ou Cortana.

Para se tornar uma solução viável, precisaremos de um meio termo, nem linguagem humana, nem Assembly: uma linguagem intermediária, e são estas as linguagens de computador. À esta altura, trabalhar em uma delas em específico não é importante e, por esta razão, escrevemos em um pseudocódigo ou Português Estruturado, que se assemelha a um programa escrito em uma linguagem de computador, e, portanto, pode ser transcrito posteriormente.

```
algoritmo leia;
variáveis
    a, b, h : inteiro;
fim-variáveis
início
    imprima("Digite a base do triângulo:");
    b := leia();
    imprima("Digite a altura do triângulo:");
    h := leia();
    a := (b * h) / 2;
    imprima("A área do triângulo é: ", a);
fim
```

Figura 2.20 – Algoritmo 1: Cálculo da área de um triângulo

Fonte: FIAP (2015)

Usamos as mesmas variáveis da solução narrativa, mas substituímos frases inteiras por “termos de comando” como “leia” e “imprima”, que serão doravante chamadas de **instruções**. Além disso, representamos o cálculo da área como uma instrução de atribuição, **s := (b * h) / 2**, que pode ser “traduzida” como: “a” recebe o valor da multiplicação dos valores de “b” e de “h” divididos por dois).

É importante observar que em muitas publicações encontraremos essa expressão indicada desta forma: **s ← (b * h) / 2**. Ora, como nas linguagens de programação não são aceitos símbolos gráficos como a seta, essa é substituída por um ou mais símbolos, geralmente usados em alguma linguagem de programação. Habitualmente são usados o sinal de igualdade (“=”) ou esse sinal precedido por dois pontos (“:=”).

Convencionaremos a partir daqui a última notação, visando evitar confusões entre atribuições, situações nas quais atribuímos um valor a uma variável, com comparações, em que é relacionado o valor de uma variável com outra, com um valor ou com uma expressão.

Também deve ser observada a indentação, ou seja, os espaços existentes entre linhas e instruções. Você pode imaginar uma linha invisível, criando “blocos” imaginários de instruções, como na imagem a seguir:

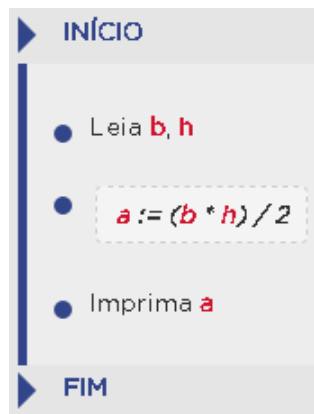


Figura 2.21 – Exemplo de algoritmo
Fonte: FIAP (2015)

Embora a indentação seja opcional na maioria das linguagens, o não endentar separa, de fato, os programadores profissionais dos amadores. Um programa feito por um profissional é rigorosamente endentado, visando os seguintes benefícios:

Facilidade na leitura do código.

Facilidade no momento de se fazer uma alteração no código, a chamada "manutenção".

Documentação do código.

Simulação:

Daqui por diante, passaremos a definir **instruções** e exemplificá-las quase sempre usando **variáveis**. De fato, programas de computador são pouco mais que um extenso conjunto de variáveis que são alteradas através de instruções. Quando tudo isso funciona harmoniosamente, temos a solução de nossos problemas. Em caso contrário, apenas mais um problema.

Então é chegada a hora de entendermos o que são variáveis de maneira mais formal.

2.5.3 Variáveis e sua tipologia

2.5.3.1 Dados, informações e variáveis

Chamamos de **dados de entrada** o conjunto de informações que o programa precisa receber para início de processamento. No exemplo anterior, a base e a altura eram nossas “entradas”, necessárias para obtenção da área do triângulo, nosso objetivo.

Chamamos de **dados de saída** o conjunto de informações que o programa devolve como resposta após o processamento.

Observemos que nossa função, basicamente, é transformar informações iniciais que usualmente chamamos de dados, em informações finais, as respostas que serão usadas por quem utiliza um programa de computador.

Toda informação para ser útil deve ser FTP (Fidedigna, Temporal e Pertinente)!

Uma informação é FTP quando é verdadeira, no momento correto e relativo à questão que queremos responder. O quadro a seguir resume essa situação:

Preciso viajar para Sorocaba agora e estou em São Paulo.

Admitir que o Google sempre diz a verdade e o Pinocchio sempre mente.

Informação	Fonte	Tempo	Relativa	FTP
A Fernão Dias está fluindo bem agora, segundo o Google.	V	v	F	F
A Castelo Branco está fluindo bem agora, segundo Pinocchio.	F	v	V	F
A Castelo Branco estava fluindo bem ontem, segundo Google.	V	F	V	F
A Castelo Branco está fluindo bem agora, segundo Google.	V	V	V	V

Quadro 2.1 – Relevância das informações
Fonte: FIAP (2015)

Entendemos, portanto, que existem vários tipos de dados, que usaremos conforme a necessidade. Por exemplo, podemos somar apenas números, quer sejam reais ou inteiros; ou somar datas com números, obtendo uma data mais adiante do que a data base dessa soma; e também concatenar expressões caracteres, algo quase que uma soma. Vamos exemplificar:

- $A \leftarrow 2$ e $B \leftarrow 3$, então $A + B$ resulta em 5.
- Já se $A \leftarrow 1,5$ e $B \leftarrow 2$ então $A + B$ resulta em 3,5.
- Ainda, se $A \leftarrow 01 \text{ jan } 1980$ e $B \leftarrow 4$, então $A + B$ resulta em 05 jan 1980.
- Finalmente se $A \leftarrow \text{"Lógica"}$ e $B \leftarrow \text{"Programação"}$ então $A + B$ resulta em LógicaProgramação (sem espaço).
- Algumas operações seriam inválidas, justamente por não fazerem qualquer sentido. Por exemplo, o que resultaria na soma de um número com uma expressão caractere? Erro, somente.
- Note que usamos uma convenção quando definimos caracteres, pois os colocamos entre aspas. Isso quer dizer, na maioria das linguagens que estamos diante de uma expressão caractere.
- Assim, admitindo que o sinal "+" some números e concatene letras teríamos também o seguinte resultado, por mais estranho que possa parecer.
- $A \leftarrow \text{"5"}$ e $B \leftarrow \text{"3"}$, então $A + B$ resulta em "53" (e não 8!).

2.5.3.2 Definindo variáveis

Variáveis são, portanto, áreas de memória que armazenarão informações pertinentes ao programa durante sua execução, e recebem este nome pois seu conteúdo pode variar ao longo deste período. Podem representar números inteiros, números reais, caracteres, tipos booleanos, palavras, datas, valores monetários, conjuntos e muito mais coisas. Em algoritmos, vamos nos ater apenas nos tipos numéricos, caracteres e booleanos.

Num programa de computador devemos especificar os tipos de dados que serão utilizados como entrada e saída. Por exemplo, no único algoritmo que vimos até agora, para cálculo da área de um triângulo, pudemos perceber que duas variáveis numéricas receberam, respectivamente, os valores da base e da altura de um triângulo, que teve sua área calculada e armazenada numa variável destinada ao cálculo da área.

Quando vertemos um programa para alguma linguagem de computador, devemos observar que nem tudo ocorre como pensamos em algoritmos, por vários motivos. Uma delas é a chamada tipologia das variáveis de qualquer linguagem de computador.

Por exemplo, enquanto a linguagem Java é tipada, ou seja, caso você tente fazer uma operação ilegal (multiplicar um caractere), o compilador dirá a você. Embora a linguagem C também seja tipada, ela o é de maneira fraca, isto é, se você tentar fazer uma operação como somar um caractere a um número, ela tentará solucionar isso para você, se isso for de interesse.

Por fim, há linguagens como Pascal, que são fortemente tipadas. Qualquer tentativa de misturar tipos, mesmo correlatos, será encarada como erro. Assim, se você tentar dividir o inteiro 2 pelo inteiro 1, em vez de uma resposta inteira 1, apenas obterá um erro. Sendo assim, caso desejarmos dividir dois números inteiros, precisaremos usar uma operação de divisão inteira (para números inteiros) e uma para divisão que tenha como resposta um número real.

Pode parecer complicado, e de fato não é algo muito simples. Todavia, programadores experientes, com o tempo, trabalham tão naturalmente nas linguagens que dominam, que situações assim passam totalmente despercebidas.

A seguir, exibimos alguns exemplos de tipos de variáveis em algumas linguagens de programação.

Tipos em Java	
Tipo	Descrição
boolean	Pode assumir o valor <i>true</i> ou o valor <i>false</i>
char	Caractere em notação Unicode de 16 bits. Serve para armazenamento de dados alfanuméricos. Também pode ser usado como um dado inteiro com valores na faixa entre 0 e 65535.
byte	Inteiro de 8 bits em notação de complemento de dois. Pode assumir valores entre $-2^7 = 128$ e $2^7 - 1 = 127$.
short	Inteiro de 16 bits em notação de complemento de dois. Os valores possíveis cobrem a faixa de $-2^{15} = -32.768$ a $2^{15} - 1 = 32.767$.
int	Inteiro de 32 bits em notação de complemento de dois. Pode assumir valores entre $-2^{31} = 2.147.483.648$ e $-2^{31} - 1 = 2.147.483.648$.
long	Inteiro de 64 bits em notação de complemento de dois. Pode assumir valores entre -2^{63} e $2^{63} - 1$.
float	Representa números em notação de ponto flutuante normalizada em precisão dupla de 32 bits em conformidade com a norma IEEE 754-1985. O menor valor positivo representável é $1.40239846e-46$ e o maior é $3.40282347e+38$.
double	Representa números em notação de ponto flutuante normalizada em precisão dupla de 64 bits em conformidade com a norma IEEE 754-1985. O menor valor positivo representável é $4.94065645841246544e-324$ e o maior é $1.7976931348623157e+308$.

Quadro 2.2 – Tipos em Java
Fonte: Elaborada pelo autor (2015)

Tipos em C		
Tipo	Tamanho em Bytes	Faixa Mínima
char	1	-127 a 127
unsigned char	1	0 a 255
signed char	1	-127 a 127
int	4	-2.147.483.648 a 2.147.483.647
unsigned int	4	0 a 4.294.967.295
signed int	4	-2.147.483.648 a 2.147.483.647
short int	2	-32.768 a 32.767
unsigned short int	2	0 a 65.535
signed short int	2	-32.768 a 32.767
long int	4	-2.147.483.648 a 2.147.483.647
signed long int	4	-2.147.483.648 a 2.147.483.647
unsigned long int	4	0 a 4.294.967.295
float	4	Seis dígitos de precisão
double	8	Dez dígitos de precisão
long double	16	Quinze dígitos de precisão

Quadro 2.3 – Tipos em C
Fonte: Autor

Tipos em Pascal		
Tipo de dado	Conjunto	Tamanho
ShortInt	-128...127	1 byte
Integer	-32768...32767	2 bytes
LongInt	-2147483648...2147483647	4 bytes
Byte	0...255	1 byte
Word	0...65535	2 bytes
Single	1.5e-45...3.4e38	4 bytes
Real	2.9e39...1.7e38	6 bytes
Double	5.0e324...1.7e308	8 bytes
Extended	3.4e-4932...1.1e4932	10 bytes
Boolean	true, false	1 byte
WordBool	true, false	2 bytes
LongBool	true, false	4 bytes
Char	1 caractere (ASCII)	1 byte
String	1 a 255 caractere (ASCII)	1.255 bytes

Quadro 2.4 – Tipos em Pascal
Fonte: Autor

2.5.4 Operadores aritméticos

Existem vários operadores que podem ser aplicados a variáveis. Vamos, inicialmente, nos concentrar num tipo específico, os números inteiros.

Tipos de dados Inteiro		
Elemento	Descrição	Símbolo
Operação	Adição	+
	Subtração	-
	Multiplicação	*
	Quociente Inteiro	Div ou /
	Resto de Divisão	Mod ou %

Quadro 2.5 – Operadores
Fonte: Elaborado pelo autor (2015)

Temos as quatro operações elementares (soma, subtração, multiplicação e divisão) conhecidas por todos, acrescidas da operação Resto da Divisão. Deve-se observar aqui o que ocorre com as variáveis, nos exemplos apresentados adiante:

Admita que a variável “a” tenha valor 2 (número dois), a variável “b” tenha valor 3 (três) e a variável “c” tenha valor 6 (seis).

Pois bem, vamos determinar os valores de “d”:

$d \leftarrow a + b$, portanto $d = 5$

$d \leftarrow c - b$, portanto $d = 3$

$d \leftarrow a * b$, portanto $d = 6$

$d \leftarrow c / b$, portanto $d = 2$ (valor de “d” é real, isto é, conta com casas decimais teóricas)

$d \leftarrow c \text{ div } b$, portanto $d = 2$ (valor de “d” é inteiro, isto é, não conta com casas decimais teóricas)

$d \leftarrow b / a$, portanto $d = 1,5$ (valor de “d” é real, isto é, conta com casas decimais teóricas)

$d \leftarrow c \text{ div } b$, portanto $d = 1$ (valor de “d” é inteiro, isto é, não conta com casas decimais teóricas)

$d \leftarrow b \% a$, portanto $d = 1$ (resto da divisão inteira de 3 por 2)

$d \leftarrow c \% a$, portanto $d = 0$ (resto da divisão inteira de 6 por 2)

2.5.5 Operadores relacionais

Além de operações aritméticas, podemos aplicar as variáveis, operações relacionais. Novamente, vamos nos concentrar inicialmente nas variáveis numéricas inteiras:

Operadores Relacionais		
Relação	Menor	<
	Maior	>
	Igual	=
	Diferente	<>
	Menor ou Igual	<=
	Maior ou Igual	>=

Quadro 2.6 – Operadores Relacionais
Fonte: Elaborado pelo autor (2015)

Exemplificando – assumindo para “a”, “b” e “c” os valores definidos anteriormente:

- a) $a > b$ (Falso, 2 é menor que 3 e não maior)
- b) $a \leq b$ (Verdadeiro, 2 é menor ou igual a 3)
- c) $a = b$ (Falso, 2 é diferente de 3)
- d) $a \neq b$ (Verdadeiro, 2 é diferente de 3)
- e) $a * b = c$ (Verdadeiro, pois 6 é igual a 6)

2.5.6 O porquê das variáveis

Variáveis funcionam como repositório de informações (dados) dentro do programa e, por esta razão, estes repositórios de armazenamento devem ser alocados na memória do computador. A maioria das linguagens de programação obriga o programador a declarar as variáveis que serão utilizadas, apresentando-as no início do programa para que os recursos necessários sejam alocados. Outras, entretanto, não obrigam a realizar esta declaração, alocando os recursos no momento em que as variáveis são mencionadas.

Se por acaso uma variável não venha a ter seu valor alterado ao longo da execução do programa, podemos declará-la como constante.

Essa situação decorre de muitas origens, por exemplo, a linguagem BCPL (origem do C) não obrigava a declaração de uma variável, antes de usá-la. No

BASIC, isso também não era obrigatório. Atualmente, entre outras, a linguagem PHP defende essa tradição.

Existem ainda situações muito específicas. Em PL/SQL, por exemplo, os contadores de uma estrutura de repetição “para” (a ser vista mais adiante), também não devem ser declarados.

Enfim, a grande regra das linguagens de programação e da lógica de programação é **que a única certeza é que nada é certo**.

Em Português Estruturado, por conseguinte, é opcional declarar ou não uma variável antes de usá-la. Talvez seja uma boa ideia não “perdermos” tempo declarando-as, antes de termos nosso algoritmo inteiramente resolvido, e só depois disso, declararmos nossas variáveis.

Lembremos, contudo, que antes de transcrevermos um algoritmo em Português Estruturado para C, Java ou Pascal, será necessário criarmos todas as diretivas necessárias nas linguagens, ou seja, deveremos declarar as variáveis.

Para resolvermos um problema qualquer, quase que certamente precisaremos armazenar as informações de entrada na memória do computador. Lembre-se, se por um lado, as variáveis são “o sangue” dos programas, por levarem as informações (nossos “nutrientes”), por outro lado, são as maiores causadoras de erros, justamente por má utilização.

Por isso, torna-se tão importante definirmos com a melhor lógica possível nossas variáveis.

2.5.7 Variáveis nas linguagens de programação

Em Java, C ou Pascal, você precisará declarar a variável antes de utilizá-la:

Exemplos em Java/C:

```
int camisapele = 10;
```

```
float salario;
```

```
String logradouro = “Av Ipiranga”;
```

Exemplos em Pascal:

```
c : char;
```

```
n : integer;
```

A sintaxe para C/Java será:

```
<tipo_de_dado> <nome_variável> [= inicialização];
```

Já para Pascal será:

```
<nome_variável> : <tipo_de_dado>
```

Como é fácil perceber, mudando-se de linguagem, muda-se a maneira como escrevemos um programa de computador, embora a lógica permaneça a mesma ou ao menos assemelhada.

2.5.8 Regras na declaração de variáveis

Depois de declaradas, variáveis podem ser usadas livremente.

A variável deve ter um nome que facilite o entendimento de sua função no programa. Um bom nome para a variável que for armazenar o salário, por exemplo, seria “salario”. Embora algumas linguagens permitam, atenha-se ao uso de apenas caracteres válidos na língua inglesa!

Toda variável é composta de um caractere alfabético ou “_”, seguido de caracteres alfabéticos, números ou “_”.

Em linguagens tipadas, apenas informações do mesmo tipo serão armazenadas nas variáveis, ou seja, não é possível armazenar um caractere em um inteiro ou vice-versa.

2.6 Instruções de entrada e saída

Para que o computador possa se comunicar com o usuário, é preciso que seja capaz de receber dados e devolvê-los. Ao conjunto de instruções que permitem escritas em tela, papel ou mídias magnéticas, damos o nome de **saídas**. Já aquelas relativas à digitação de dados ou de sua leitura em meios magnéticos, chamamos de **entradas**.

2.6.1 Saída

Entendemos “saída”, como o conjunto de instruções que permitem a comunicação com o usuário ou a escrita de dados ou de quaisquer outras informações em tela (saída principal), papel (impressora) ou em disco (gravação). Para saída, usaremos, na maioria das vezes, o comando “**imprima**”, mas “exiba” ou “escreva” são igualmente válidos.

Em Java, por exemplo, usamos a instrução `System.out.print` (“algo para imprimir”) e no C#, `Console.Write`(“imprima isso”) desempenha este papel;

Vale notar que a expressão a ser “impressa” aparece cercada por parênteses, recurso usado em inúmeras linguagens de programação. Por ser uma expressão alfanumérica e não uma variável, a expressão aparece entre aspas, outra característica encontrada em inúmeras linguagens de programação.

Além disso, notamos que a instrução tem um terminador, no caso, o ponto e vírgula, novamente característica observada em inúmeras outras linguagens de programação.

Em Pascal temos `Write` (‘algo a imprimir’); ou `WriteLn` (‘algo a imprimir’); a diferença entre um comando e outro é que no primeiro o cursor é posicionado na mesma linha do que foi impresso, enquanto que no segundo o cursor é posicionado na linha abaixo (Ln, de line).

Em Linguagem C, a instrução varia levemente, “`printf`”, mas há necessidade de envio de um caractere especial solicitando o salto de linha (“`\n`”). Em Pascal, “`write/writeln`”, como em Java.

2.6.2 Entrada

Para a entrada de dados, utilizaremos a classe Scanner em Java (conhecida como scanf em C ou Console.ReadLine(), quando falamos em C#). Já em Pascal, “read/readln”, pois com ela podemos capturar os dados informados através do teclado do computador.

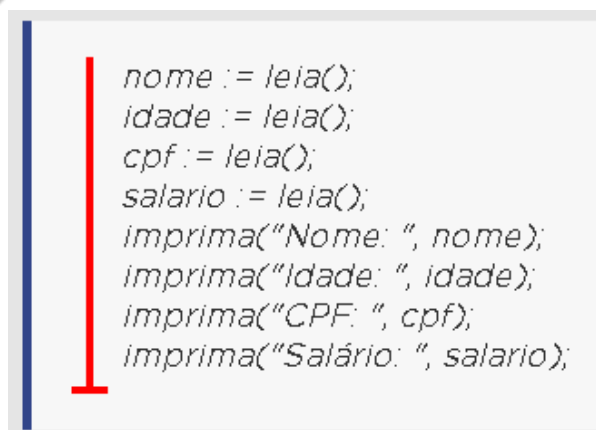
Em todas as linguagens de programação dispomos de um conjunto de instruções para leituras nos mais diversos dispositivos, que resultam em entradas de dados.

2.6.3 Exemplificando entradas e saídas

Nos exemplos apresentados adiante, veremos como escrever instruções de entrada e saída em Português Estruturado e em algumas linguagens de programação:

Em nosso primeiro exemplo, vamos ler uma série de dados pessoais, através de um teclado e mostrar esse resultado na tela do computador, em Português Estruturado.

Elabore um algoritmo que leia um nome, sua idade, seu CPF e seu salário. Em seguida, esses dados deverão ser exibidos na tela.



```
nome := leia();  
idade := leia();  
cpf := leia();  
salario := leia();  
imprima("Nome: ", nome);  
imprima("Idade: ", idade);  
imprima("CPF: ", cpf);  
imprima("Salário: ", salario);
```

Figura 2.22 – Exemplo de português estruturado
Fonte: Elaborado pelo autor (2015)

Observemos que a barra de indentação marca o início e o final do programa, ou um “parágrafo”. Também chamamos essa estrutura de “bloco de programa”, ou

seja, no final das contas um algoritmo nada mais é que um conjunto de “blocos” devidamente ordenados.

Vamos agora ver programas semelhantes, respectivamente, em Java, C e Pascal:

```
package t01;
import java.util.Scanner;

public class t01c {
    public static void main(String[] args) {
        int idade, salario;
        Scanner ent = new Scanner(System.in);
        System.out.print("Digite o Nome: ");
        String nome = ent.nextLine();
        System.out.print("Digite o CPF: ");
        String cpf = ent.nextLine();
        Scanner num = new Scanner(System.in);
        System.out.print("Digite a idade: ");
        idade = num.nextInt();
        Scanner sal = new Scanner(System.in);
        System.out.print("Digite o salario: ");
        salario = sal.nextInt();
        System.out.println("Nome: " + nome);
        System.out.println("Idade : " + idade);
        System.out.println("Salario: " + salario);
        System.out.println("CPF: " + cpf);
    }
}
```

Figura 2.23 – Programa em Java
Fonte: Elaborado pelo autor (2015)

```
salario2.c
INSERT Line: 1 Col: 1
<Exit edit: ESC or F10> <Help: ^J or F1>

main()
{
    char nome[30], cpf[20];
    int idade, salario;
    clr();
    printf("Nome : "); scanf("%s", nome);
    printf("Idade: "); scanf("%d", &idade);
    printf("CPF : "); scanf("%s", cpf);
    printf("Salario: "); scanf("%d", &salario);
    printf("\n\nNome: %s", nome);
    printf("\nIdade: %d", idade);
    printf("\nCPF: %s", cpf);
    printf("\nSalario: %d", salario);
}
```

Figura 2.24 – Programa em C
Fonte: Elaborado pelo autor (2015)

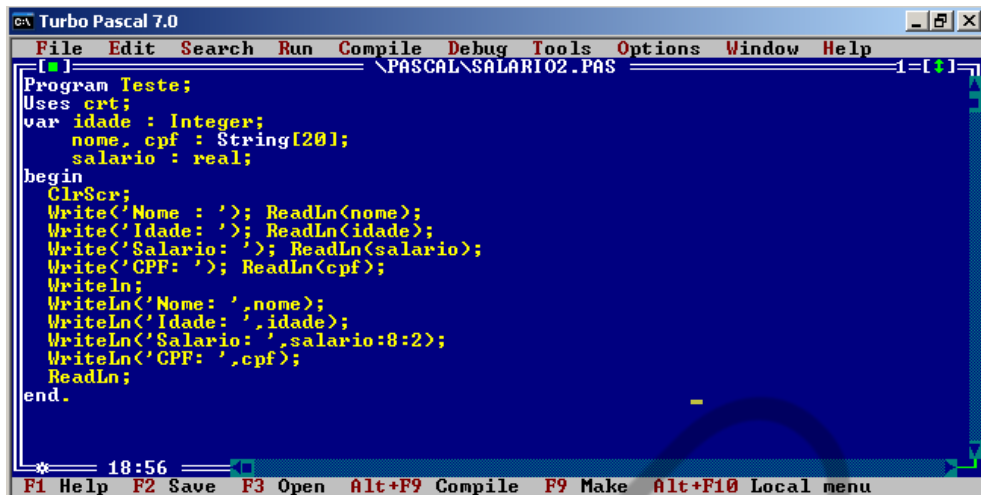


Figura 2.25 – Programa em Pascal
Fonte: Elaborado pelo autor (2015)

2.6.4 Problemas resolvidos

1. Dados dois números, imprima sua soma.

```
algoritmo soma;
variáveis
    a, b, s : inteiro;
fim-variáveis
início
    imprima("Digite o número 1:");
    a := leia();
    imprima("Digite o número 2:");
    b := leia();
    s := a + b;
    imprima("A soma dos dois números é: ", s);
fim
```

Figura 2.26 – Soma
Fonte: FIAP (2015)

2. Escreva um programa que receba a data de nascimento de uma pessoa e a data de hoje e calcule, aproximadamente, quantos dias ela viveu. Considere um mês com 30 dias e um ano com 12 meses.

```

algoritmo dias;
variáveis
    dh, mh, ah, dn, mn, an, wd1, wm1, wa, wd2, wm2, tot : inteiro;
fim-variáveis
início
    imprima("Digite o dia de hoje: ");
    dh := leia();
    imprima("Digite o mês de hoje: ");
    mh := leia();
    imprima("Digite o ano de hoje: ");
    ah := leia();
    imprima("Digite o dia de nascimento: ");
    dn := leia();
    imprima("Digite o mês de nascimento: ");
    mn := leia();
    imprima("Digite o ano de nascimento: ");
    an := leia();
    wd1 := 30 - dn;
    wm1 := (12 - mn) * 30;
    wa := (ah - an - 1) * 360;
    wd2 := dh;
    wm2 := (mh - 1) * 30;
    tot := wd1 + wm1 + wa + wd2 + wm2;
    imprima("Dias Vividos (por Totalização): ", tot);
fim

```

Figura 2.27 – 12 meses
Fonte: FIAP (2015)

3. Escreva um algoritmo que calcule o IMC de uma pessoa.

Lembrando que o IMC é calculado através da expressão “Peso / Altura²”

```

algoritmo imc;
variáveis
    peso : inteiro;
    altura, imc : real;
fim-variáveis
início
    imprima("Digite altura: \nExemplo: 1.58");
    altura := leia();
    imprima("Digite peso: ");
    peso := leia();
    imc := peso / (altura * altura);
    imprima("IMC: ", imc);
fim

```

Figura 2.28 – Peso/ Altura²
Fonte: FIAP (2015)

4. Escreva um algoritmo que resolva uma equação do 1º Grau.

Admita que a expressão " $Ax + B = 0$ " sempre seja válida, ou seja, "A" sempre será um número distinto de zero.

```
algoritmo equacao;  
variáveis  
    a,b : inteiro;  
    x : real;  
fim-variáveis  
início  
    imprima("Digite a: ");  
    a := leia();  
    imprima("Digite b: ");  
    b := leia();  
    x := -b / a;  
    imprima("Divisão: ", x);  
fim
```

Figura 2.29 – Zero
Fonte: FIAP (2015)

2.6.5 Problemas propostos

1. Descreva com os necessários detalhes os passos para ir do polo FIAP até o MASP. Se você não está em São Paulo, faça caminho análogo de sua casa até um ponto turístico de sua cidade.
- A partir da porta da unidade Lins da FIAP, olhando em direção a rua, caminhe até esta a 2 metros da guia da rua.
 - Gire o corpo para a direita em ângulo de noventa graus, ficando com o corpo paralelo a rua.
 - Caminhe até a esquina da rua. Pare a cinquenta centímetros da rua.
 - Olhe à sua direita e verifique se algum veículo automotor se aproxima. Caso exista algum veículo se aproximando, aguarde a passagem do mesmo. Repita a verificação até que não haja mais veículos automotores se aproximando. Caso não exista nenhum veículo automotor se aproximando, atravesse a rua.
 - Caminhe até a esquina da rua. Pare a cinquenta centímetros da rua.
 - Olhe à sua esquerda e verifique se algum veículo automotor se aproxima. Caso exista algum veículo se aproximando, aguarde a passagem do mesmo. Repita a verificação até que não haja mais veículos automotores se aproximando. Caso não exista nenhum veículo automotor se aproximando, atravesse a rua.

- Caminhe até a esquina da rua. Pare a cinquenta centímetros da rua.
 - Olhe à sua esquerda e à sua direita, verifique se algum veículo automotor se aproxima. Caso exista algum veículo se aproximando, aguarde a passagem do mesmo. Repita a verificação até que não haja mais veículos automotores se aproximando. Caso não exista nenhum veículo automotor se aproximando, atravesse a rua.
 - Gire o corpo para a direita em ângulo de noventa graus, ficando com o corpo paralelo a rua.
 - Caminhe até encontrar uma placa amarela onde está escrito "Ponto de Taxi".
 - Verifique se existe um taxi disponível. Caso não exista, aguarde até que um novo taxi chegue ao local. Caso exista um taxi disponível, embarque no veículo.
 - Diga ao motorista "Leve-me ao MASP".
 - Espere dentro do carro até que o motorista informe "Chegamos ao MASP".
 - Pague o motorista.
 - Desembarque do taxi.
2. Dados os números 12 e 13, encontre todos os divisores de cada um desses números. Sempre considere serem números inteiros e positivos, a menos que explicitamente indicado o contrário no enunciado. Sempre considere que os extremos (inclusive) são parte da solução a menos que explicitamente (exclusive) estiver indicado que não se devem considerar os extremos.

Divisores de 12

- Divida o número 12 por 1. Se o resto da divisão for zero, considere 1 como um dos divisores de 12.
- Divida o número 12 por 2. Se o resto da divisão for zero, considere 2 como um dos divisores de 12.
- Divida o número 12 por 3. Se o resto da divisão for zero, considere 3 como um dos divisores de 12.
- Divida o número 12 por 4. Se o resto da divisão for zero, considere 4 como um dos divisores de 12.

- Divida o número 12 por 5. Se o resto da divisão for zero, considere 5 como um dos divisores de 12.
- Divida o número 12 por 6. Se o resto da divisão for zero, considere 6 como um dos divisores de 12.
- Divida o número 12 por 7. Se o resto da divisão for zero, considere 7 como um dos divisores de 12.
- Divida o número 12 por 8. Se o resto da divisão for zero, considere 8 como um dos divisores de 12.
- Divida o número 12 por 9. Se o resto da divisão for zero, considere 9 como um dos divisores de 12.
- Divida o número 12 por 10. Se o resto da divisão for zero, considere 10 como um dos divisores de 12.
- Divida o número 12 por 11. Se o resto da divisão for zero, considere 11 como um dos divisores de 12.
- Divida o número 12 por 12. Se o resto da divisão for zero, considere 12 como um dos divisores de 12.

Divisores de 13

- Divida o número 13 por 1. Se o resto da divisão for zero, considere 1 como um dos divisores de 13.
- Divida o número 13 por 2. Se o resto da divisão for zero, considere 2 como um dos divisores de 13.
- Divida o número 13 por 3. Se o resto da divisão for zero, considere 3 como um dos divisores de 13.
- Divida o número 13 por 4. Se o resto da divisão for zero, considere 4 como um dos divisores de 13.
- Divida o número 13 por 5. Se o resto da divisão for zero, considere 5 como um dos divisores de 13.

- Divida o número 13 por 6. Se o resto da divisão for zero, considere 6 como um dos divisores de 13.
 - Divida o número 13 por 7. Se o resto da divisão for zero, considere 7 como um dos divisores de 13.
 - Divida o número 13 por 8. Se o resto da divisão for zero, considere 8 como um dos divisores de 13.
 - Divida o número 13 por 9. Se o resto da divisão for zero, considere 9 como um dos divisores de 13.
 - Divida o número 13 por 10. Se o resto da divisão for zero, considere 10 como um dos divisores de 13.
 - Divida o número 13 por 11. Se o resto da divisão for zero, considere 11 como um dos divisores de 13.
 - Divida o número 13 por 12. Se o resto da divisão for zero, considere 12 como um dos divisores de 13.
 - Divida o número 13 por 13. Se o resto da divisão for zero, considere 13 como um dos divisores de 13.
3. Escreva um algoritmo que descreva a maneira de encontrar todos os divisores de um número qualquer.
- Comece a contagem em 1.
 - Divida o número desejado pelo número que está na contagem.
 - Se o resto da divisão for zero, considere o valor que está na contagem como um dos divisores do número.
 - Incremente o valor da contagem em 1.
 - Repita as operações anteriores até que o número da contagem seja superior ao número que desejamos descobrir os divisores.
4. Calcule o salário de um programador, cujo salário é função das horas de trabalho semanais. Deverá ser multiplicado o número de horas trabalhadas por 4,5 semanas. O resultado encontrado é o total de horas-mês dedicado. Deve ser

acrescido a esse valor 10%, devido à gratificação por desempenho. Para calcular, multiplique o salário-base por esse percentual. Além disso, existe o DSR - Descanso Semanal Remunerado que corresponde a 1/6 sobre a remuneração total, portanto deve ser calculado sobre a soma anteriormente verificada.

```
Algoritmo "Calcula_Salario"

// Algoritmo desenvolvido em VisualG 3.0

Var

    salario_por_hora: real;
    horas_trabalhadas: inteiro;
    salario_calculado: real;

Inicio

    escreva("Digite o salario por hora do programador: ")

    leia(salario_por_hora)

    escreva("Digite o numero de horas trabalhadas pelo
programador: ")

    leia(horas_trabalhadas)

    salario_calculado := (salario_por_hora * 1.10 *
horas_trabalhadas)
```

```
    salario_calculado := salario_calculado + salario_calculado  
/ 6
```

```
    escreva("O salario deste mes do programado sera: ",  
salario_calculado)
```

```
Fimalgoritmo
```

5. Dados 3 números, apresente a soma e a média destes números.

```
Algoritmo "Soma_e_Media"
```

```
// Algoritmo desenvolvido em VisualG 3.0
```

```
// Observação: a instrução "escreval" salta uma linha
```

```
// após ter escrito a mensagem na tela
```

```
Var
```

```
numero_1: real;
```

```
numero_2: real;
```

```
numero_3: real;
```

```
soma: real;
```

```
media: real;
```

```
Inicio
```

```
    escreva("Digite o primeiro numero: ")
```

```
leia(numero_1)

escreva("Digite o segundo numero: ")

leia(numero_2)

escreva("Digite o terceiro numero: ")

leia(numero_3)

soma := numero_1 + numero_2 + numero_3

escreval("Soma dos tres numero : ", soma)

media := (numero_1 + numero_2 + numero_3) / 3

escreva("Media dos tres numero : ", media)

Fimalgoritmo
```

6. Dados 2 números, apresente a subtração de cada um pelo outro.

```
// Algoritmo desenvolvido em VisualG 3.0

// Observação: a instrução "escreval" salta uma linha após
```

```
// ter exibido o resultado da insturção

Var

numero_1: real;
numero_2: real;
subtracao: real;

Inicio

escreva("Digite o primeiro numero: ")

leia(numero_1)

escreva("Digite o segundo numero: ")

leia(numero_2)

subtracao := numero_1 - numero_2

escreval("Primeiro numero MENOS segundo numero : ",
subtracao)

subtracao := numero_2 - numero_1

escreva("Segundo numero MENOS primeiro numero : ",
subtracao)
```


Fimalgoritmo

2.7 Instruções e Funções

Na evolução das linguagens de programação, sempre existiram dificuldades na separação do que seria uma instrução da própria linguagem e o que seria uma função a ser escrita na própria linguagem.

Pensemos em elevar um número ao quadrado. Na prática, basta multiplicar o número por si mesmo. Assim, teríamos alternativamente e dependendo da linguagem de programação uma das três possibilidades:

```
algoritmo elevaquadrado;
variáveis
  a,b,c,d : inteiro;
fim-variáveis
início
  a := 3;
  b := a ** 2;
  c := a ^ 2;
  d := a * a;
  imprima(a," elevado ao quadrado resulta: ",b, c, d);
fim
```

Figura 2.30 – Algoritmo para elevar o número 3 ao quadrado de diferentes maneiras.,
Fonte: FIAP (2015)

Note que esse algoritmo não será processado em praticamente nenhuma linguagem de programação. Embora todas entendam a instrução “a * a” (uma multiplicação simples), algumas compreenderão “a ** 2” enquanto outras “a ^ 2”, por se tratarem de operadores de exponenciação diferentes. Na verdade, linguagens antigas como o FORTRAN usavam “**”, pois o símbolo “^” sequer estava disponível nos primeiros computadores.

Nada impede, entretanto, que o implementador entenda que elevar ao quadrado é algo que possa ser feito na própria linguagem. Na realidade, cada implementador entende da maneira que lhe aprouver o que é função e o que é instrução.

E nada é tão óbvio como possa parecer ser...

Por exemplo, Dennis Ritchie criador da Linguagem C, a construiu com o firme propósito de torná-la rápida e independente do hardware (computador) onde será executada. Para tanto, era fundamental tornar as instruções de entrada e saída funções que pudessem ser implementadas em C no próprio Sistema Operacional nativo, onde os programas compilados em C seriam executados. Assim, foi criada em C uma biblioteca chamada “stdio.h”, que significa “Standard Input Output”, ou seja, entradas e saídas padrão, em que “comandos” como *imprima*, que em C pode ser representado na função (e não comando) *printf*.

Embora as entradas e saídas padrão sejam implementadas de maneira diferente em vários sistemas operacionais, a linguagem C as padronizou em funções como scanf() e printf(), facilitando e tornando mais produtiva a codificação – do contrário, seria necessário manipular diferentes hardwares em diferentes sistemas operacionais, um verdadeiro pesadelo.

Ao repensar nosso problema de elevar um número ao quadrado, chegamos à uma solução que qualquer linguagem de programação poderá implementar:

```
algoritmo elevaquadrado;
variáveis
  a,b,c : inteiro;
fim-variáveis
início
  a := 3;
  b := a * a;
  c := quadrado(a);
  imprima(a," elevado ao quadrado resulta: ",b, c);
fim

função quadrado(n : inteiro) : inteiro
início
  retorne n * n;
fim
```

Figura 2.31 – Algoritmo de elevar ao quadrado melhorado com função.
Fonte: FIAP (2015)

Desta vez, nenhum símbolo especial foi usado, como são “***” ou “^”. Apenas usamos o “*” símbolo universal para multiplicação. Temos agora uma “função” que constrói uma solução para “elevar ao quadrado”.

Funções (e Procedimentos) são um tópico muito importante e serão tratados mais adiante num estudo bem mais aprofundado. No momento, nos basta saber que podemos construir estruturas equivalentes a alguma instrução que não exista numa linguagem.

Veremos nesse material como construir um algoritmo destinado ao cálculo da raiz quadrada, criado por ninguém menos que Isaac Newton, que pode ser implementado em qualquer linguagem de programação, pois a “função” cálculo da raiz quadrada de um número, também não é nativa em todas as linguagens de programação.

Assim, uma função é um subprograma ou uma rotina de um programa principal ou “programa chamador”; ao ser chamado no programa principal, o fluxo é desviado para a função ou subprograma, que executa uma tarefa estabelecida e, em seguida, devolve o controle ao programa chamador.

A parte interessante é que informações podem ser passadas nestes “desvios”, tornando o subprograma mais flexível e útil. No exemplo, note que a função **quadrado()** recebe um número inteiro entre seus parênteses, e a essa entrada de informações damos o nome de **parâmetros**. Com este número inteiro armazenado em “n” (qualquer número inteiro no intervalo válido!) o subprograma o multiplica por si mesmo, devolvendo-o elevado para o programa principal.

Como podemos facilmente multiplicar o número por si mesmo, parece desnecessário criarmos uma função para isso... na verdade, é mesmo! Contudo, a ideia nesse momento é apenas de apresentação de um conceito.

Conceito (as funções) que tem um criador, ou melhor, criadora. A primeira pessoa a pensar na criação de funções repetitivas que pudessem ser usadas para executar certos processos inúmeras vezes foi ninguém menos que Ada de Lovelace.

2.8 Fluxograma

Trata-se de uma técnica muito útil para documentação de processos administrativos e que é usada por alguns autores como apoio ao aprendizado de lógica de programação.

O Fluxograma, também conhecido como diagrama de blocos, é uma forma quase que universal de representação, pois utiliza figuras geométricas para ilustração dos passos a serem observados na resolução de problemas.

Retomemos nosso exemplo anterior para cálculo da área de um triângulo. Obteríamos:

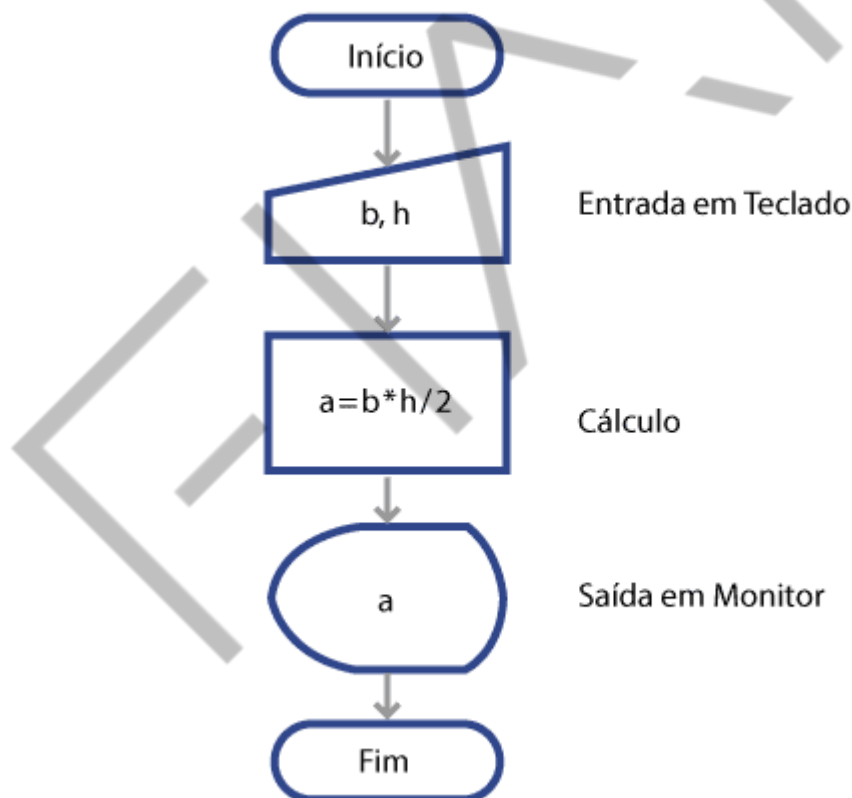


Figura 2.32 – Fluxograma: cálculo da área de um triângulo
Fonte: Autor, adaptado por FIAP (2015)

A seguir, alguns símbolos usados na representação de instruções, geralmente utilizados nas réguas de fluxograma.

Cada instrução ou ação a ser executada deve ser representada por meio de um símbolo gráfico:

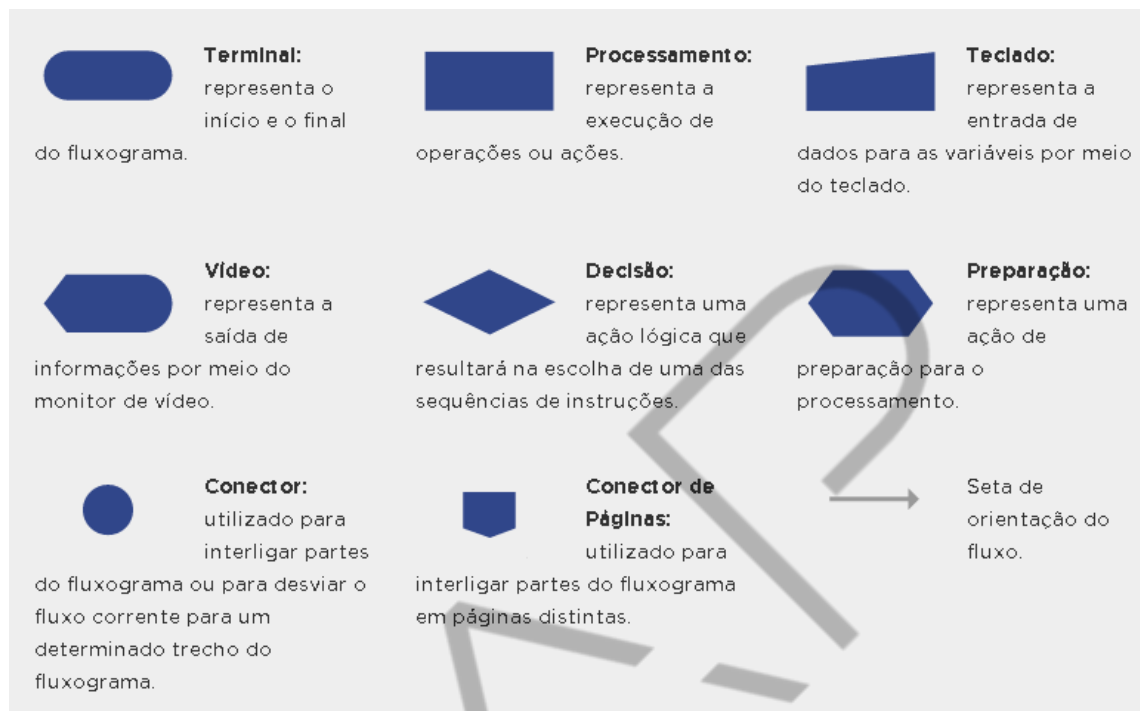


Figura 2.33 – Elementos de Fluxograma
Fonte: Autor, adaptado por FIAP (2015)

Não nos utilizaremos desse recurso nesse material, uma vez que vários autores concordam quanto a pouca eficiência dos fluxogramas, principalmente quando a complexidade dos algoritmos cresce. No entanto, para aqueles que preferirem ao menos no princípio utilizarem de uma notação mais visual, essa técnica poderá ser válida.

REFERÊNCIAS

ENCYCLOPEDIA and history of programming languages. Disponível em: <<http://www.scriptol.org/>>. Acesso em: 14 jan. 2011.

FEOFILOFF, Paulo. **Algoritmos em Linguagem C**. Rio de Janeiro: Campus, 2009.

FORBELLONE, André L.V.; EBERSPACHER, Henri F. **Construção de Algoritmos e Estruturas de Dados**. São Paulo: Pearson Prentice Hall, 2010.

FURGERI, Sérgio. **Java 2, Ensino Didático**. São Paulo: Érica, 2002.

GANE, Chris e SARSON, Trish. **Análise Estruturada de Sistemas**. São Paulo: LTC- Livros Técnicos e Científicos, 1983.

GONDO, Eduardo. **Apostila: Notas de Aula**. São Paulo, 2008.

LATORE, Robert. **Aprenda em 24 horas Estrutura de Dados e Algoritmos**. Rio de Janeiro: Campus, 1999.

MANZANO, José A.N.G. e OLIVEIRA, Jayr F. **Algoritmos: Lógica para o Desenvolvimento de Programação**. 23. ed. São Paulo: Érica, 2010.

PIVA JUNIOR, Dilermando et al. **Algoritmos e Programação de Computadores**. Rio de Janeiro: Campus, 2012.

PUGA, Sandra; RISSETTI, Gerson. **Lógica de Programação e Estrutura de Dados**. São Paulo: Pearson Prentice Hall, 2009.

ROCHA, Antonio Adrego. **Estrutura de Dados e Algoritmos em Java**. Lisboa: FCA- Editora de Informática, 2011.

RODRIGUES, Rita. **Apostila: Notas de Aula**. 2008.

SALVETTI, Dirceu Douglas e BARBOSA, Lisbete Madsen. **Algoritmos**. São Paulo: Makron Books, 1998.

SCHILDT, Herbert. **Linguagem C - Guia Prático**. São Paulo: McGraw Hill, 1989.

WOOD, Steve. **Turbo Pascal – Guia do Usuário**. São Paulo: McGraw Hill, 1987.

ZIVIANI, Nivio. **Projeto de Algoritmos com implementações em Pascal e C**. São Paulo: Pioneira, 1999.