

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/341327207>

Introduction to Programming Logic

Book · January 2015

CITATIONS

0

READS

78

1 author:



[Alvaro F. Pinheiro](#)

University of Pernambuco

29 PUBLICATIONS 5 CITATIONS

SEE PROFILE

Fundamentos da Engenharia de Software

Lógica de Programação

PINHEIRO, Álvaro Farias
Autor

Fundamentos da Engenharia de Software

Lógica de Programação

Publicação 2015

O autor acredita que todas as informações aqui apresentadas estão corretas e podem ser utilizadas para qualquer fim legal. Entretanto, não existe qualquer garantia explícita ou implícita, de que o uso de tais informações conduzirá sempre ao resultado desejado. Os nomes de sites e empresas, por ventura, mencionados, foram utilizados apenas para ilustrar os exemplos, não tendo vínculo nenhum com o livro, não garantindo a sua existência nem divulgação. Eventuais erratas estarão disponíveis para download no site de publicação.

Algumas imagens utilizadas neste livro foram obtidas a partir da Internet.

Dados da Publicação

Pinheiro, Álvaro Farias

Fundamentos da Engenharia de Software: Lógica de Programação

1. ed. – Volume VII

Recife, 2015

Selo Editorial: Publicação Independente

Bibliografia

ASIN B017GFVCVM

ISBN 9781520276984

1. Teoria da Computação
2. Autômatos
3. Paradigmas e Processos de Desenvolvimento
4. Algoritmos

Fundamentos da Engenharia de Software

Lógica de Programação

Publicação Independente
Livro em português com o título
Fundamentos da Engenharia de Software
Lógica de Programação
Volume 7
1ª Edição
Recife - Pernambuco – Brasil
Primeira revisão em Novembro de 2015
Segunda revisão em Junho de 2016

Fundamentos da Engenharia de Software

Lógica de Programação

Ao meu pai Alberto Nunes Pinheiro um homem que
me orgulho de ser filho.

Fundamentos da Engenharia de Software

Lógica de Programação

“Um raciocínio lógico leva você de A a B. A imaginação leva você
a qualquer lugar que você quiser.”
Albert Einstein

SUMÁRIO

1 Teoria da Computação	1
1.1 Definições.....	1
1.2 Modelos	1
1.3 Finalidade	2
1.4 Classes.....	2
1.5 Expressão Regular	3
1.5.1 União.....	4
1.5.2 Concatenação	4
1.5.3 Kleene	4
1.6 Teoria dos Autômatos	5
1.7 Classes dos Autômatos	5
1.7.1 Autômatos Finitos Determinísticos	5
1.7.2 Autômatos Finitos Não Determinísticos	5
1.7.3 Autômatos Finitos Não Determinísticos com Transições ϵ (Espontâneas)	5
1.8 Outros Tipos de Autômatos	5
1.8.1 Autômatos à Pilha (ou <i>pushdown</i>).....	5
1.8.2 Máquina de Turing	6
1.8.3 Autômato Linearmente Limitado ou Linear <i>Bounded</i>	6
1.9 Projetos de Linguagens de Programação.....	6
<i>Complex Instruction Set Computer</i> (CISC).....	6
<i>Reduced Instruction Set Computer</i> (RISC)	7
1.10 Linguagem Formal.....	9
1.10.1 Tipo 3	9
1.10.2 Tipo 2	11
1.10.3 Tipo 1	14
1.10.4 Tipo 0	14
1.10.5 Máquina de Turing	15
1.11 Ferramentas	17
1.12 Vídeos	17
1.13 Sugestões de Leitura.....	17
1.14 Referências Web	17

SUMÁRIO

1.15 Referências Bibliográficas	18
2 Paradigmas de Desenvolvimento	19
2.1 Paradigmas de Programação	20
2.1.1 Paradigma Funcional	20
2.1.2 Paradigma Lógico	20
2.1.3 Paradigma Declarativo	20
2.1.4 Paradigma Imperativo	21
2.1.5 Paradigma Orientado a Eventos	21
2.1.6 Paradigma Orientado a Objetos	21
2.1.7 Paradigma Orientado a Aspectos.....	22
2.2 Linguagens de Programação.....	23
Projetos de Linguagens.....	25
2.3 Compilador	29
2.4 Interpretador	30
2.5 Máquina Virtual.....	30
2.6 Análises dos Compiladores	31
2.7 Princípios dos Paradigmas de Desenvolvimento.....	31
2.8 Princípios de Linguagens de Programação	34
2.9 Backus-Naur (BNF)	40
2.10 Conceitos Básicos do Paradigma Orientado a Objetos	40
Classe	41
Objeto.....	42
Herança.....	42
Polimorfismo	43
Sobreescreta (Override)	43
Sobrecarga (Overload).....	43
Encapsulamento.....	43
2.11 Exemplos.....	43
2.12 Ferramentas	44
2.13 Vídeos	44
2.14 Sugestões de Leitura.....	44
2.15 Referências Web	44

SUMÁRIO

2.16 Referências Bibliográficas	45
3 Lógica de Programação	46
3.1 Conceitos.....	47
3.1.1 Variáveis Numéricas	49
3.1.2 Variáveis alfanuméricas ou Caractere.....	49
3.1.3 Variáveis Lógicas	49
3.1.4 Tamanho das Variáveis.....	49
3.1.5 Operações Numéricas.....	52
3.1.6 Operações Caractere	53
3.1.7 Operações Lógicas ou Relacionais	53
3.2 Atividades	57
3.3 Ferramentas	58
3.4 Sugestões de Leitura.....	58
3.5 Referências Web	58
3.6 Referências Bibliográficas	59
4 Referências Bibliográficas	60
5 Referências Web	61
6 Ferramentas	62
7 Sugestões de Leitura	63

ILUSTRAÇÕES

Figura 1.1 Classes	3
Figura 1.2 Hierarquia Chomsky	3
Figura 1.3 Camadas de Linguagens	8
Figura 1.4 Autômato	11
Figura 1.5 Autômato	12
Figura 1.6 Elementos da Transição	12
Figura 1.7 Transição	12
Figura 1.8 Gerado pela ferramenta Simulador	14
Figura 1.9 Autômato gerado pela ferramenta Simulador	14
Figura 1.10 Máquina de Turing	16
Figura 1.11 MT	16
Figura 2.1 Mapa Mental de Paradigmas de Desenvolvimento	20
Figura 2.2 Paradigmas	22
Figura 2.3 Linguagens	25
Figura 2.4 Compilador	29
Figura 2.5 Interpretador	30
Figura 2.6 Máquina Virtual	31
Figura 2.7 Conceitos Gerais de Paradigmas	36
Figura 2.8 Estrutura de Pilha	40
Figura 2.9 Estrutura de Fila	40
Figura 2.10 Símbolo de Classe	42
Figura 2.11 Símbolo de Objeto	42

SIGLAS e ACRÔNIMOS

ANSI - *American National Standards Institute*

ISO - *International Organization for Standardization*

OMG - *Object Management Group*

OOP - *Object Ooriented Programming*

SEI - *Software Engineering Institute*

SQL - *Structured Query Language*

SWEBOK - *Software Engineering Body of Knowledge*

NOTAÇÕES

Este livre adota as seguintes notações, que devem ser observadas para o perfeito entendimento do material aqui produzido.

Notações Técnicas:

<..> - Menor que / Maior que, nos comandos SQL usado para metadados.

<..> - Menor / Maior, no texto indica que é um *hyperlink* (*link* para um endereço web).

[..] - Abre e fecha parênteses, no texto indica utilização de uma sigla.

(..) - Abre e fecha colchetes, nos comandos SQL indica opcionalidade.

| - Barra, para indicar “ou lógico”.

_ - Sublinhado, nos comandos SQL indica o *default* (padrão).

_ - Sublinhado, no texto indica que é um *hyperlink* (*link* para um endereço web).

Negritos - Títulos de capítulos, seções e subseções.

Itálico - Para palavras de língua estrangeira.

SOBRE O AUTOR



Álvaro Farias Pinheiro tem mais de 30 anos de experiência no campo da computação e nestes, mais de 15 anos na área acadêmica, lecionando disciplinas com foco em engenharia de software.

É mestre em Engenharia de Software pelo Centro de Estudos e Sistemas Avançados do Recife (CESAR) com a dissertação *Uso do Balanced Scorecard na Avaliação de Fábricas de Software* utilizadas como Instrumento de Ensino de Tecnologia da Informação Baseado em Problemas Reais. Especialista em Gestão Pública pela Universidade de Pernambuco (UPE) e em

Metodologias de Desenvolvimento pela União Brasileira de Tecnologia. É Bacharel em Sistemas de Informação pela Faculdade Integrada do Recife (FIR), iniciando sua vida acadêmica no Bacharelado em Ciência da Computação da Universidade Católica de Pernambuco (UNICAP).

Iniciou as atividades profissionais em 1986 como Programador na Elógica Microsistemas desenvolvendo para baixas e altas plataformas nas linguagens dBase, Clipper, FoxPro, Cobol e Natural. Como Analista de Sistemas passou pelas instituições: Centro de Prestação de Serviços Técnicos de Pernambuco (CETEPE), Empresa de Fomento da Informática do Estado de Pernambuco (FISEPE), Instituto de Tecnologia em Informática (ITECI), Apply Solutions e está hoje na Agência Estadual de Tecnologia da Informação do Estado de Pernambuco (ATI-PE) por concurso público, trabalhando como Analista em Gestão da Tecnologia da Informação e Comunicação, em exercício na Procuradoria Geral do Estado de Pernambuco (PGE-PE), atuando como Coordenador da Área de Sistemas.

Começou a lecionar em 1992 em cursos técnicos nas empresas ITECI e IBRATEC, e desde 2006 é professor de cursos de graduação e pós-graduação em computação nas instituições de ensino superior da região, a citar, Universidade Federal Rural de Pernambuco (UFRPE), Instituto Federal de Pernambuco (IFPE), Faculdade Estácio do Recife, Faculdade Integrada do Recife (FIR), Faculdade Joaquim Nabuco, Faculdade do Recife, Unibrattec e durante esses anos realizou diversos cursos de extensão como os de Análise de Ponto de Função, Administração de Redes, Linguagens de Programação Natural, Centura, Delphi e CSharp, Banco de Dados Adabas, MySQL, Postgre e Oracle, Análise Estruturada e Orientada a Objetos, entre outros.

Possui experiência na área de Engenharia de *Software* nas disciplinas de Teoria da Computação, Sistemas de Informação e Colaborativos, Desenvolvimento Estrutural e Orientado a Objetos para aplicações Desktop, Web e Mobile, Processos de Desenvolvimento IRUP, XP e Scrum, Banco de Dados SQLServer

SOBRE O AUTOR

e Oracle, e Gestão de Projetos com PMBOK e SWEBOK. Trabalha atualmente na coordenação de projetos e com educação presencial e a distância, ministrando aulas e realizando pesquisas em ambientes virtuais de ensino e linguagens de programação.

<http://www.alvarofpinheiro.eti.br>

<http://br.linkedin.com/in/alvarofpinheiro>

<https://sites.google.com/site/alvarofpinheiro>

<http://www.facebook.com/alvarofpinheiroaulas>

PREFÁCIO

Este livro aborda basicamente os conceitos básicos de programação como autômatos, tipos de linguagens, princípios dos compiladores, paradigmas de desenvolvimento e lógica de programação. A motivação de escrita deste livro foram os meus alunos que ao longo dos anos ministrando aulas e sempre escrevendo nos quadros o conteúdo passado e também disponibilizando para os discentes materiais complementares para auxiliar o acompanhamento do assunto ministrado, sempre era abordado por um ou outro me indagando porque não disponibilizado esse material num livro. Assim este livro é uma coletânea sintetizada sobre lógica de programação.

O livro aborda os seguintes assuntos:

- **Capítulo 1** *Teoria da Computação*. Neste é abordado os conceitos básicos de linguagem de programação em relação as suas teorias de computabilidade e complexidade, entrando nos conceitos de autômatos e seguindo com os de interpretadores, compiladores e máquinas virtuais.
- **Capítulo 2** *Paradigmas de Desenvolvimento*. Aqui se introduz nas regras de criação das linguagens, seus conceitos para projetos de linguagens de programação, explicando sucintamente os principais paradigmas de programação.
- **Capítulo 3** *Lógica de Programação*. A lógica de programação é abordada aqui para introduzir nos conceitos básicos de programação como estrutura de dados, variáveis, constantes, etc. Com a finalidade de conceituar os princípios básicos da programação objetivando a introdução a linguagem de programa Java que é abordada no capítulo seguinte.

Finalizando, este livro possui várias subseções que disponibilizam *hyperlink's* para *wiki's* sobre os assuntos abordados, como também *link's* para outros *ebook's* utilizados como referencial bibliográfico para o conteúdo produzido.

Álvaro Farias Pinheiro

1 Teoria da Computação

A Engenharia de Software é um segmento da Computação que existe decorrente da necessidade de se construir *software* com qualidade, buscando planejar e organizar o desenvolvimento e dentro deste campo a disciplina Teoria da Computação tem a finalidade de conceituar o funcionamento dos compiladores, entendendo como as linguagens de programação são construídas e classificadas.

A finalidade dessa disciplina é fomentar os conceitos básicos para o conhecimento de compiladores, no qual o seu entendimento é essencial para a construção de um programa de computador que, a partir de um código fonte escrito em uma linguagem compilada, cria um programa semanticamente equivalente, porém escrito em outra linguagem, chamado de código final. E para entender um pouco mais é necessário a elucidação das gramáticas: léxica, sintática e semântica.

Análises de um compilador:

- Léxica - é o processo de analisar a entrada de linhas de caracteres e produzir uma sequência de símbolos;
- Sintática - é a análise das regras que regem a construção de frases; e
- Semântica - é a terceira fase da compilação onde se verifica os erros semânticos, isto é, de lógica.

1.1 Definições

O que é computação? Uma forma de defini-la é a forma de se estudar a solução de um problema, isto é, analisar e projetar solução a um problema.

O que é Teoria da Computação? É um subcampo da ciência da computação, que tem por objetivo descobrir quais problemas podem ser computados, fazendo o uso de um modelo.

1.2 Modelos

Foi no século XX que teve início aos estudos do que iria ser a Teoria da Computação. O objetivo era (e é ainda), descobrir quais problemas matemáticos poderiam ser resolvidos por um método simples.

Diversos modelos a partir desse momento foram propostos, como a Máquina de Turing, modelo Recursivo, Cálculo Lambda, Regras Gramaticais, Cadeias de Markov, Sistemas de Post, etc. Todos os formalismos propostos acima são

equivalentes em termos computacionais, assim o objetivo atual é verificar a possibilidade de realizar certos tipos de computação em determinados tipos de máquina.

1.3 Finalidade

A teoria da computação serve para estudar os modelos computacionais, como também os limites dessa computação. Buscando respostas para questões do tipo: Independente do *hardware*, quais problemas jamais poderão ser resolvidos por uma máquina? Problema da parada; Problema da Correspondência de Post.

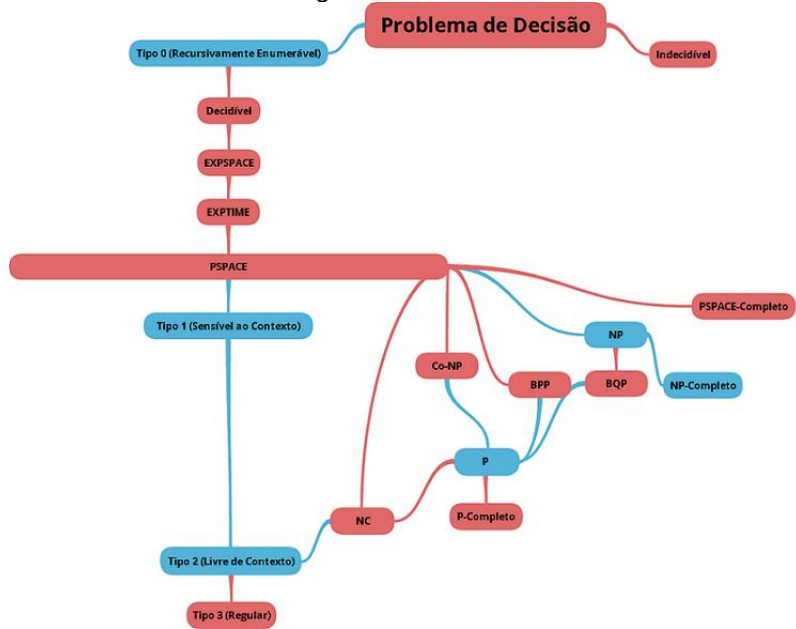
Quais problemas podem ser resolvidos, mas necessitam de um longo período que se torna impraticável? (Aritmética de Presburger) Em que situações podem ser mais difíceis a resolução de um problema do que verificar cada uma das soluções manualmente? Classes P e NP.

1.4 Classes

A teoria da computação se divide em duas classes. Observando a figura 1.1 se pode observar as classes azuis que pertencem à teoria da computabilidade e as em vermelho a teoria da complexidade. Assim, com o estudo das Expressões Regulares, podem se identificar e especificar padrões de cadeias de caracteres, que são utilizados em linguagens de programação.

Outro formalismo são os Autômatos Finitos, utilizados em desenho de circuitos e em alguns sistemas de resolução de problemas. As gramáticas livres de contexto são utilizadas para especificar a sintaxe das linguagens de programação; um formalismo equivalente são os autômatos com pilha, ou *pushdown* autômato.

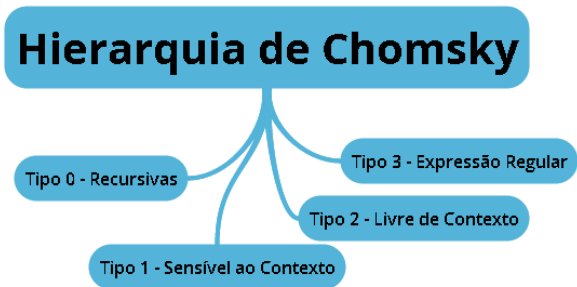
Figura 1.1 Classes



Fonte: Próprio Autor

Modelos de computação diferentes podem realizar tarefas distintas. Assim, uma forma de estudar o poder de um modelo computacional é estudar a classe das linguagens formais que o modelo pode gerar.

Figura 1.2 Hierarquia Chomsky



Fonte: Internet

1.5 Expressão Regular

Uma *regex* é uma forma flexível de identificar cadeias de caracteres. Essas expressões podem ser escritas numa linguagem formal e interpretadas por

programas que servem como analisadores sintáticos, examinando o texto e identificando partes que casam com a especificação dada.

O matemático Kleene desenvolveu as *regex* como uma notação ao que ele chamava de álgebra de conjuntos regulares, servindo como base para os primeiros algoritmos computacionais de busca e depois para ferramentas de tratamento de texto.

Atualmente as *regex* servem para procura e substituição de texto em editores de texto e linguagens de programação, validação de formatos de texto, validação de protocolos, validação de formatos digitais, realce de sintaxe e filtragem de informação.

Como uma *regex* serve para descrever um conjunto de cadeias de caracteres sem precisar listar todos os elementos do conjunto, existem alguns caracteres coringa para isso.

O formalismo padrão usa três operações básicas:

- (+) união
- (.) concatenação
- (*) kleene

Essas construções podem ser combinadas arbitrariamente para formar expressões complexas, assim como expressões aritméticas com números e operações de adição, subtração, multiplicação e divisão.

1.5.1 União

$$L(E + F) = L(E) \cup L(F)$$

$$E=\{001,110\} \quad F=\{\epsilon,11,110\}$$

$$L=\{001,110,\epsilon,11\}$$

1.5.2 Concatenação

$$L(E \cdot F) = L(E) \cdot L(F)$$

$$E=\{001,110\} \quad F=\{\epsilon,11,110\}$$

$$L=\{001,00111,001110,110,11011,110110\}$$

1.5.3 Kleene

$$L(E^*) = (L(E))^*$$

$$E=\{001,110\}$$

$$L=\{\epsilon,001,110,001001,001110,\dots\}$$

$$F=\{"ab", "c"\}^*$$

$$L=\{\epsilon, "ab", "c", "abab", "abc", "cab", "cc", "ababab", "abcbab", \dots\}$$

1.6 Teoria dos Autômatos

É um subcampo da ciência da computação que estuda as máquinas de estados finitas, considerando suas representações matemáticas.

Um autômato é um modelo matemático de uma máquina de estados que funciona como um reconhecedor de uma determinada linguagem. Baseado na Teoria dos Sistemas, os estados estão sempre em transição, por exemplo: uma TV pode estar ligada e transitar para desligada, tem então um sistema com dois estados.

Uma forma de utilização dos autômatos é no reconhecimento de padrões, a exemplo, editores de textos.

1.7 Classes dos Autômatos

A seguir serão apresentados três tipos de autômatos finitos:

1.7.1 Autômatos Finitos Determinísticos

Cada estado de um autômato desse tipo tem uma transição para cada símbolo do alfabeto;

1.7.2 Autômatos Finitos Não Determinísticos

Os estados de um autômato desse tipo podem ou não ter uma transição para cada símbolo do alfabeto, ou podem ter até mesmo múltiplas transições para cada símbolo; e

1.7.3 Autômatos Finitos Não Determinísticos com Transições ϵ (Espontâneas)

Além de ser capaz de pular para mais (ou nenhum) estados com quaisquer símbolos, esse tipo de autômato pode pular para outros estados sem que haja nenhum símbolo (*String* vazia).

1.8 Outros Tipos de Autômatos

Extensões de Autômatos Finitos. A família de linguagens aceitas pelos autômatos descritos acima é chamada de família de Linguagens regulares.

Autômatos mais poderosos podem aceitar linguagens mais complexas. Tais autômatos incluem:

1.8.1 Autômatos à Pilha (ou *pushdown*)

Tais máquinas são idênticas aos autômatos finitos determinísticos, exceto pelo fato deles adicionalmente carregarem a memória sob a forma de pilhas. A função de transição δ irá depender agora dos símbolos que estão no topo da pilha, e especificarão como a pilha deve ser modificada a cada transição. Os autômatos à pilha aceitam linguagens livres de contexto.

1.8.2 Máquina de Turing

Essas são as mais poderosas máquinas computacionais. Elas possuem uma memória infinita sob a forma de uma fita, uma cabeça que pode realizar a leitura ou alteração da fita, além de poder movimentar-se ao longo da fita. As máquinas de Turing são equivalentes a algoritmos e compõem a base teórica da computação moderna. Máquinas de Turing aceitam linguagens recursivamente enumeráveis.

1.8.3 Autômato Linearmente Limitado ou Linear *Bounded*

É uma máquina de Turing limitada, ao invés de ser uma fita infinita, a fita tem uma quantidade de espaço proporcional ao tamanho da *String* de entrada. Aceitam linguagens sensíveis ao contexto.

1.9 Projetos de Linguagens de Programação

Agora que entendemos o básico, vamos entender o que são linguagens de programação e como elas são projetadas.

Existem algumas características básicas necessárias a serem trabalhadas para projetarmos uma linguagem. Essas são: a sua arquitetura, a configuração técnica, os seus padrões e a compatibilidade com o legado.

Então vamos entender o que é arquitetura de uma linguagem. Basicamente podemos dizer que, projetar uma linguagem é levar em consideração seu *hardware*, e o modelo que o define, isto é, o modelo de von Neumann Eckert, lembrando, trata-se do modelo, entrada-processamento-saída. Então, boa parte, das linguagens, como C, é projetada para essa arquitetura de máquina, mas existem outras que não o são a exemplo o Lisp.

Detalhando um pouco mais a questão de arquitetura, temos que as linguagens são projetadas para atendê-las, e nos dias atuais, são dois os projetos básicos de arquiteturas:

Complex Instruction Set Computer (CISC), traduzindo Computador com um Conjunto Complexo de Instruções.

Explicando basicamente o que é, trata-se de um padrão de construção de processadores capazes de executar centenas de instruções complexas. Mas, o

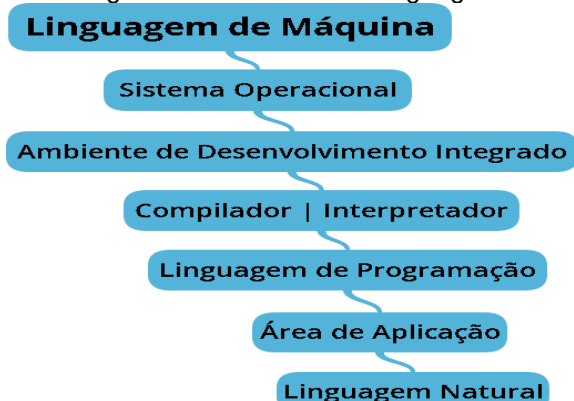
que quer dizer isso? Uma forma simples de explicar, que dizer que, no processador, são armazenadas instruções chamadas de microcódigo, que se não o existissem, teriam que ser codificadas pelos programadores nas linguagens. Esse armazenamento de instruções nos processadores minimiza e simplifica o trabalho de desenvolvimento, já que muito código comum a qualquer tipo de programa já está embutido no processador. Como consequência direta as linguagens concebidas para essa arquitetura, tendem a ter um código executável bem menor. Mas, a sua desvantagem é que como a microprogramação realizada visa prever uma grande quantidade de funcionalidades comuns aos aplicativos, torna esse tipo de processador se torna pesado em relação ao tempo de resposta. E para completar com o passar do tempo, notou-se que muitas dessas funcionalidades embutidas nos microcódigos nunca eram usadas na maioria das aplicações desenvolvidas. Resumindo temos um melhor código com pior desempenho.

Reduced Instruction Set Computer (RISC), traduzindo Computador com um Conjunto Reduzido de Instruções.

Trata-se de outro padrão de construção de processadores, que como a própria sigla já diz, oferece um conjunto simples e pequeno de instruções armazenadas no processador. Simplificando, os processadores desse padrão não possuem ou possuem muito pouco microcódigo. Consequência direta é que o esforço de desenvolvimento torna-se maior. Porém, como os processadores não possuem microcódigo desnecessário, prevendo funcionalidades comuns a diversos tipos de programas, que normalmente não são usadas, o tempo de resposta é muito superior. Grandes partes dos projetos de processadores atuais seguem esse modelo de arquitetura. Resumindo temos um melhor desempenho com pior código.

Outra preocupação no projeto de uma linguagem de programação é em relação ao nicho que ela pretende atender, para resolver os seus problemas. Esse objetivo gera uma série de restrições, de natureza técnica ou de modelo, que são normalmente observados, durante o projeto de uma linguagem de programação. Essas restrições tecnológicas estão relacionadas aos seguintes níveis de abstração, observadas na figura 1.3.

Figura 1.3 Camadas de Linguagens



Fonte: Próprio Autor

Mais uma preocupação no projeto de desenvolvimento de uma nova linguagem é a sua compatibilidade com os seus projetos anteriores, caso existam. Isso se deve pela necessidade de se poderem manter os códigos produzidos pelas versões anteriores com a nova linguagem. Uma tarefa muito complicada. Mas, a preocupação com o legado na área de TIC é uma constante.

Depois, que uma linguagem se torna bem aceita pelo mercado, mais uma preocupação se passam a ter que é a sua padronização. Essa padronização é necessária, para uniformizar o uso e a evolução da linguagem. Normalmente as entidades que formalizam a padronização são a ANSI e o ISO, veja os *links* para saber mais.

Uma categorização importantíssima nos projetos de linguagens de programação é a Hierarquia de Chomsky, que classifica as linguagens em tipos, os quais podem ser vistos abaixo.

Tipo 3: Gramática Regular > Linguagem Regular > Autômato Finito

Tipo 2: Gramática Livre de Contexto > Linguagem LC > Autômato com Pilha

Tipo 1: Gramática Sensível ao Contexto > Linguagem SC > Autômato Linear

Tipo 0: Gramática Irrestrita > Linguagem Recursiva > Máquina de Turing

Mas para se ter uma ideia dos mesmos, vamos falar sucintamente sobre eles. Começando com a explicação de Linguagens Formais e Autômatos. Basicamente é o estudo de modelos matemáticos que permitam definir e reconhecer linguagens, possibilitando a sua classificação, estruturação e caracterização. É conhecida como Teoria da Ciência da Computação. A finalidade do estudo dessas teorias é fornecer fundamentos para a Ciência da Computação em relação à decidibilidade, computabilidade e complexidade computacional. Essas teorias fundamentam várias áreas da computação, como

processamento de linguagens, reconhecimento de padrões, modelagem de sistemas.

1.10 Linguagem Formal

Sabemos que linguagem é uma forma de comunicação e que com a evolução do homem, essas ganharam especializações, do tipo léxico, sintático e semântico. Essas especializações servem tanto para linguagens naturais ou idiomas, como também para linguagens de programação e dos protocolos de comunicação. Assim, podemos dizer que as Linguagens Formais consistem de regras para representação e especificação de linguagens.

As representações são feitas por reconhecedores e geradores de linguagens, que nada mais é, que um recurso que verificam se uma sentença pertence ou não a uma determinada linguagem. Esses recursos são chamados de autômatos, que podem ser: Autômatos Finitos Determinísticos; Autômatos Finitos Não Determinísticos; Autômatos de Pilha; e Máquina de Turing.

Já os geradores são recursos que permitem a geração sistemática de todas as sentenças de uma linguagem. Esses recursos são as gramáticas, a qual se destaca a de Chomsky. Resumindo, as linguagens formais podem ser representadas de maneira finita e precisa através de teorias matemáticas, o que caracteriza a Ciência da Computação.

1.10.1 Tipo 3

Das linguagens regulares, que podem ser representados pelos autômatos finitos determinísticos e não determinísticos. Essas linguagens se baseiam nas Expressões Regulares. Essas expressões são representadas por três formas básicas: União, representada pelo símbolo (+) permitindo unir expressões; Concatenação, representada pelo símbolo (.) permitindo concatenar expressões; e Kleenee, representada pelo símbolo (*) permitindo potencializar expressões.

Seguem exemplos das *Regular Expression (Regex)*:

Para $L(E + F) = L(E) \cup L(F)$, como $E=\{001,110\}$ e $F=\{\epsilon,11,110\}$, temos $L=\{001,110,\epsilon,11\}$

Para $L(E \cdot F) = L(E) \cdot L(F)$, como $E=\{001,110\}$ e $F=\{\epsilon,11,110\}$, temos $L=\{001,00111,001110,110,11011,110110\}$

Para $L(E^*) = (L(E))^*$, como $E=\{001,110\}$, temos $L=\{\epsilon,001,110,001001,001110,\dots\}$

Essas expressões podem ser representadas por Autômatos. E o que são? São máquinas de estados finitos, isto é, um autômato finito. Trata de uma modelagem de um comportamento, composto por estados, transições e ações. Um estado armazena informações sobre o passado, isto é, ele reflete as mudanças desde a entrada num estado, no início do sistema, até o momento

presente. Uma transição indica uma mudança de estado e é descrita por uma condição que precisa ser realizada para que a transição ocorra. Uma ação é a descrição de uma atividade que deve ser realizada num determinado momento. Máquinas de estados finitos podem ser representadas por meio de um diagrama de estados ou diagrama de transição de estados.

A quintupla dos autômatos finitos é: $M = (Q, \Sigma, \delta, i, F)$ onde:

Q é um conjunto finito de estados;

Σ é o alfabeto de entrada;

δ é a função de transição de estados;

i é o estado inicial; e

F é um conjunto de estados finais ou de aceitação.

Segue um exemplo de autômato finito para definição de uma linguagem:

$L = \{p \in \{0,1\}^* \mid 111 \text{ seja uma subcadeia de } p\}$

Lê-se: Para uma linguagem, dado uma palavra que pertence ao dialeto 0 e 1 baseado na expressão de klenee, tal que, 111 seja uma subcadeia da palavra.

Temos:

Observe que os valores da quintupla para esse exemplo são:

Para Sigma temos 0 e 1;

Para Q temos q_0, q_1, q_2, q_3, q_4 e q_5 ;

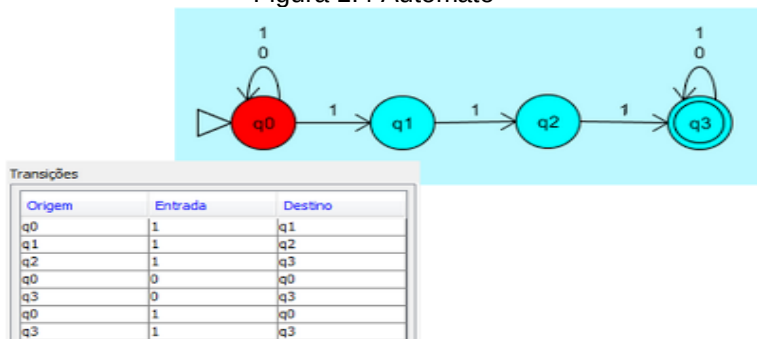
Para Delta temos a tabela de transições acima;

Para i temos q_0 ; e

Para F temos q_3 e q_4 .

Um detalhe importante para a definição dos autômatos finitos é a pequena diferença entre os determinísticos (AFD) e não determinísticos (AFND). O primeiro só permite a transição de um estado para outro, já o segundo é mais flexível, permitindo transições no mesmo estado, até mesmo transições espontâneas, isto é, independente de uma leitura. Veja o exemplo, para a mesma linguagem, sendo o primeiro atendido por um AFD e o segundo por um AFND. Para a mesma linguagem $L = \{p \in \{0,1\}^* \mid 111 \text{ seja uma subcadeia de } p\}$ temos uma representação mais limpa, observe a figura 1.4.

Figura 1.4 Autômato



Fonte: Próprio Autor

1.10.2 Tipo 2

Das linguagens livre de contexto, essas são representadas por autômatos com pilhas. Que são idênticos aos autômatos finitos determinísticos, exceto pelo fato eles adicionalmente usam o recurso de memória sob a forma de pilhas. A função de transição δ irá depender agora dos símbolos que estão no topo da pilha, e especificarão como a pilha deve ser modificada a cada transição.

Os autômatos com pilha determinísticos são definidos com a sêxtupla dos autômatos com pilha é: $M = (Q, \Sigma, G, \delta, i, I, F)$ onde:

Q é o conjunto finito de estado;

Σ é o alfabeto de entrada;

δ é a função de transição;

i é o estado inicial;

I é o símbolo inicial da pilha; e

F é o conjunto de estados finais.

Os autômatos com pilha não determinísticos são definidos com a sêxtupla dos autômatos com pilha é: $M = (Q, \Sigma, G, \delta, i, I, F)$ onde:

Q é o conjunto finito de estado;

Σ é o alfabeto de entrada;

G é o alfabeto da pilha;

δ é a função de transição;

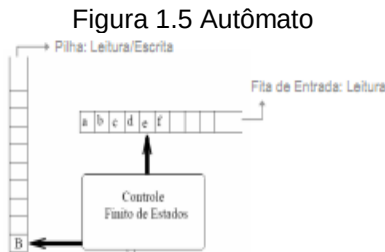
i é o estado inicial;

I é o símbolo inicial da pilha; e

F é o conjunto de estados finais.

Análogo às Linguagens Regulares (LR), as Linguagens Livres de Contexto (LCC) podem ser utilizadas como reconhecedoras. O seu funcionamento e os componentes de um Autômato com Pilha (AP) são: a) uma fita de entrada, b)

uma cabeça de leitura, c) um indicador de estado e d) uma pilha. Veja a figura 1.5.

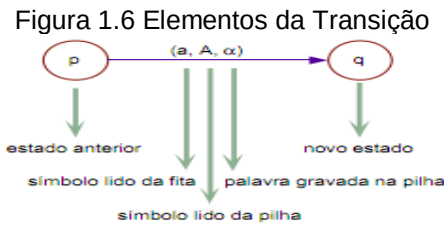


Fonte: Internet

O funcionamento de um AP consiste em posicionar a cabeça de leitura em um símbolo de entrada e em seguida posicionar a outra cabeça de leitura no topo da pilha obtendo os valores contidos na fita e na pilha e, com base nesses valores os seguintes passos devem ser realizados:

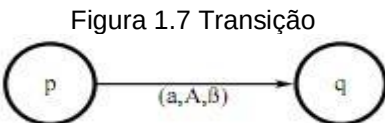
- Remover o item no topo da Pilha;
- Colocar um novo item na Pilha;
- Transitar para outro estado; e
- Avançar para o próximo símbolo na entrada.

Para entender melhor esse mecanismo, veja a figura 1.6 e observe os passos para reconhecimento da linguagem: $L = \{a^*b^* \mid * \geq 0\}$. Para cada *a* lido, deve-se inseri-lo na pilha, e para cada *b* lido deve-se remover um *a* da pilha, no término uma palavra *p* será reconhecida se no final do processo a pilha estiver vazia.



Fonte: Internet

A figura 1.7 representa a transição, onde temos (*a*) é o símbolo lido da fita, (*A*) representa o símbolo lido da pilha, e (*beta*) a palavra gravada na pilha. Resumindo temos:



Fonte: Internet

Vamos entender o AP com o exemplo:

$$L1 = \{a^*b^* \mid * \geq 0\}$$

$$M1 = (\{a,b\}, \{q1,q2,q3\}, \delta, q1, \{B\})$$

$$\delta1(q1, a, ?) = \{(q1, B)\}$$

$$\delta1(q1, b, B) = \{(q2, ?)\}$$

$$\delta1(q1, ?, ?) = \{(q3, ?)\}$$

$$\delta1(q2, b, B) = \{(q2, ?)\}$$

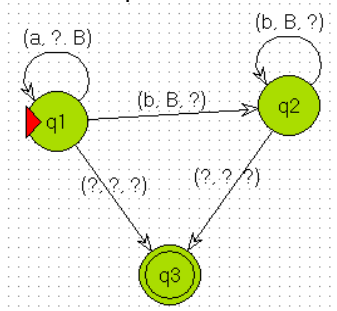
$$\delta1(q2, ?, ?) = \{(q3, ?)\}$$

A máquina M1 para representar a linguagem L1 é iniciada no estado q1 e para cada símbolo (a) lido da fita é armazenado um símbolo (B) na pilha, no estado q2 é realizada a verificação se cada símbolo (b) da fita existe um correspondente (B) na pilha. O algoritmo somente aceitará a palavra se ao terminar de ler toda ela a pilha estiver vazia.

A representação desse autômato em grafo pode ser observado na figura 1.8 e essa máquina aceita palavras como: ab, aabb, aaabbb,...

Para explicar melhor, a máquina M1 possui o estado q1 com três símbolos, os quais o primeiro da esquerda para direita é o símbolo a ser reconhecido que será lido na fita (a), o segundo é o símbolo a ser lido da pilha (?=vazio) e o terceiro é o símbolo a ser gravado na pilha (B). Para cada (a) lido na fita um (B) foi gravado na pilha, podendo ser repetido quantas vezes for desejado. A transição do estado q1 para o q2 consiste em uma leitura de três símbolos, o qual tem o símbolo que será lido da fita (b), em seguida o símbolo que será lido da pilha (B), e o terceiro o símbolo que será gravado na pilha (?) assim sendo um vazio. No estado q2 com os mesmos três símbolos, temos o símbolo que será lido da fita (b), em seguida o símbolo que será lido da pilha (B), e o terceiro o símbolo que será gravado na pilha (?) assim sendo um vazio, podendo ser repetido quantas vezes for desejado. Quando não existir mais símbolos para serem lidos na fita e da pilha, chega-se ao final.

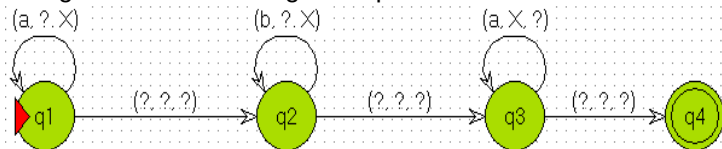
Figura 1.8 Gerado pela ferramenta Simulador



Fonte: Próprio Autor

Mais um exemplo. Fazer uma máquina para: $M2 = \{a^n b^m a^{n+m} \mid n \geq 0 \text{ e } m \geq 0\}$
 Esta máquina aceita palavras como: abaa, aabbbaaaa, aaabbbbaaaaaa,... como pode ser observado na figura 1.9.

Figura 1.9 Autômato gerado pela ferramenta Simulador



Fonte: Próprio Autor

1.10.3 Tipo 1

Das linguagens sensíveis ao contexto, essas são representadas por uma forma restrita de uma máquina de Turing não determinística que satisfaça as duas condições seguintes: Seu alfabeto de entrada inclui dois símbolos especiais, servindo como marcadores de direção da esquerda e direita; e Necessitam de um ponteiro que se move para a esquerda levando além do marcador.

1.10.4 Tipo 0

Das linguagens recursivas, essas são representadas por máquinas como as de Turing, Moore e Mealy. A Máquina de Turing é um recurso teórico conhecido também como máquina universal, que foi concebido pelo matemático britânico Alan Turing, antes do surgimento dos modernos computadores digitais. É um modelo abstrato de computador, que trata apenas dos aspectos lógicos de funcionamento da memória, estados e transições. Porém com essa abstração pode-se modelar qualquer computador digital. A Máquina de Moore é outro recurso que utiliza apenas ações de entrada e a saída depende somente do estado. A vantagem do modelo de Moore é a simplificação do comportamento.

Por exemplo, modelar um elevador com quatro estados possíveis para sua porta "Aberta", "Fechada", "Abrindo", "Fechando". A máquina de estados reconhece dois comandos: "comando_abrir" e "comando_fechar" que disparam a alteração de estado. A ação de entrada no estado "Abrindo" liga o motor que abre a porta, a ação de entrada no estado "Fechando" liga o motor na outra direção, fechando a porta. Os estados "Aberta" e "Fechada" não desempenham nenhuma ação. E a Máquina de Mealy é mais um recurso onde a saída depende da entrada e do estado. O uso de uma máquina Mealy normalmente leva a uma redução no número de estados. O exemplo anterior nessa máquina seria: inicie o motor para fechar a porta se o "comando_fechar" chegar e inicie o motor na direção oposta para abrir a porta se o "comando_abrir" chegar.

1.10.5 Máquina de Turing

A Máquina de Turing (MT) é bastante simples, baseia-se em uma fita de comprimento infinito que pode se mover para direita ou para esquerda célula-a-célula e cada célula pode conter um caractere ou estar vazio. A cabeça chamada de C é fixa e pode ler e escrever conteúdo em uma célula, deixando-o cheio ou vazio.

A MT é composta por uma fita que como já dito é infinita e pode se mover. Por um registrador de estados que armazena o estado da máquina, isto é, um dispositivo que armazena os estados, incluindo o estado inicial com o qual o registrador é inicializado. A MT também é composta por uma tabela de transição, que determina o caractere a ser escrito, como também o sentido da navegação da fita, se para esquerda e ou para direita, e qual o novo estado, isto é, caractere que foi lido na fita e o estado em que se encontra. E caso não exista entrada de dados a máquina para.

A séptupla que define a MT é $M = (Q, \Sigma, \Gamma, s, b, F, \delta)$ onde:

Q é um conjunto finito de estados;

Σ é um alfabeto finito de caracteres;

Γ é o conjunto finito de caracteres;

s é o estado inicial;

b é o símbolo branco; e

δ é o conjunto dos estados finais.

Para entender o funcionamento da MT vamos verificar a figura 1.10 com o exemplo utilizando o alfabeto $\{b, 0\}$, no qual o b (bê) representa o caractere branco e o 0 (zero) o caractere a ser lido e escrito. O comportamento dessa máquina será o seguinte: Ela espera uma série de 0's na fita, com a cabeça posicionada inicialmente no 0 mais à esquerda, e repete os 0's com um b no meio. Representando, temos "000" que se torna "000b000". Nesse exemplo o conjunto de estados é $\{s1, s2, s3, s4, s5\}$ e o estado inicial é $s1$. Para visualizar o exemplo, segue a tabela de ação

Figura 1.10 Máquina de Turing

Estado	Caractere Lido	Caractere Escrito	Caractere Sentido	Estado Novo
s1	0	b	→	s2
s2	0	0	→	s2
s2	b	b	→	s3
s3	b	0	←	s4
s3	0	0	→	s3
s4	0	0	←	s4
s4	b	b	←	s5
s5	0	0	←	s5
s5	b	0	→	s1

Fonte: Próprio Autor

A leitura dessa tabela pode ser feita da seguinte forma como mostra a figura 1.11: Estando a MT no estado s1 e o caractere lido pela cabeça for zero, então se escreve o caractere b e move-se uma posição para a direita, além de mudar o estado para s2.

Figura 1.11 MT

Passo	Estado	Fita
1	s1	00
2	s2	b0
3	s2	b0b
4	s3	b0bb
5	s4	b0b0
6	s5	b0b0
7	s5	b0b0
8	s1	00b0
9	s2	0bb0
10	s3	0bb0
11	s3	0bb0b
12	s4	0bb00
13	s4	0bb00
14	s5	0bb00
15	s1	00b00

Fonte: Próprio Autor

O comportamento dessa máquina pode ser descrito como um laço, que é iniciado em s1, substitui o primeiro zero por um b, então usa o s2 para mover para a direita, passando pelos 0's e pelo primeiro b encontrado, s3 então passa pela próxima sequência de 0's e substitui o primeiro b que encontrar por um 0, s4 move de volta para a esquerda, passando pelos 0's até encontrar um b e passando para o estado s5. O s5 então move para a esquerda, passando pelos

0's até achar o b escrito inicialmente em por s1. Substitui o b por 0, move uma posição para a direita e entra no estado s1 para realizar outra passada pelo laço, repetindo o processo até s1 encontrar um b que fique entre duas cadeias de 0's.

Finalizando, os principais objetivos de um projeto de linguagem de programação são a obtenção da:

- Abstração; Clareza;
- Confiabilidade;
- Eficiência;
- Legibilidade;
- Ortogonalidade;
- Simplicidade; e
- Suporte

1.11 Ferramentas

Auger. Em < <http://www.evoluma.com/auger/index.html> >

JFLAP. Em < <http://www.jflap.org/> >

Moodle. Em < <http://www2.ead.ufrpe.br/uab/moodle/login/index.php> >

Regex. Em < <http://alessandrosantos.com.br/lab/er/> >

Simulador. Em < <http://www.simuladordeautomatos.com/> >

Simulador Exemplos. Em < <http://goo.gl/PV1U4L> >

1.12 Vídeos

Compiladores. Em < <http://vimeo.com/112860457> >

1.13 Sugestões de Leitura

Compiladores. Em < <http://goo.gl/KRL19c> >

Engenharia de Software. Em < <http://goo.gl/p2OV92> >

Programação de Computadores. Em < <http://goo.gl/uldtPN> >

1.14 Referências Web

ANSI site oficial. Em < <http://www.ansi.org/> >

ANSI wiki. Em <

http://pt.wikipedia.org/wiki/American_National_Standards_Institute >

Autômatos Finitos Determinísticos wiki. Em < <http://goo.gl/3IJJxV> >

Autômatos Finitos Determinísticos. Em < <http://goo.gl/NZHyRP> >

Autômatos Finitos e Expressões Regulares. Em < <http://goo.gl/3Ox2WM> >

Converter Autômatos Finitos em Expressão Regular wiki. Em < <http://goo.gl/Bd2kcz> >
Expressão Regular wiki. Em < <http://goo.gl/wNRox> >
Introdução a Teoria da Computação Fascículo 1. Em < <http://goo.gl/4Q6ZHe> >
Introdução a Teoria da Computação Fascículo 2. Em < <http://goo.gl/9bhlns> >
Introdução a Teoria da Computação Fascículo 3. Em < <http://goo.gl/vQsKpg> >
ISO site oficial. Em < <http://www.iso.org/iso/home.html> >
ISO wiki. Em < <http://goo.gl/vjWG> >
Ling. Formais e Autômatos. Em < <http://www.deamo.prof.ufu.br/CursoLFA.html> >
Transição de Estados wiki. Em < <http://goo.gl/QVb3F5> >

1.15 Referências Bibliográficas

BEDREGAL, B.; BENEDITO, A. Introdução à Teoria das Ling. Formais dos Autômatos e da C. RN: Edunp, 2010.
DIVERIO, Tiarajú; *et al.* Teoria da Computação: Máquinas Universais e Comp. 3ªed. Porto Alegre: Bookman, 2011.
HOPCROFT, John; *et al.* Introdução à Teoria de Autômatos, Linguagens e Computação. 1ªed. RJ: Campus, 2002.
MENEZES, Paulo. Linguagens Formais e Autômatos. 6ªed. Porto Alegre: Bookman, 2011.
SUDKAMP, Thomas. Languages and Machines: An Introduction to the Theory of Computer Science, 1997.

2 Paradigmas de Desenvolvimento

Essa disciplina serve para entender o que é um paradigma de desenvolvimento, quais os tipos de linguagens, com suas características e arquiteturas. O objetivo é introduzir os conceitos de engenharia de *software*, mais especificadamente fazendo o uso de linguagens de programação do paradigma orientado a objetos para o desenvolvimento de *software*.

Nos dicionários, paradigma significa um sistema de regras e regulamentos, que podem ser escritos ou não, e que servem para estabelece fronteiras ou limites, determina como se comportar dentro dos limites para alcançar o sucesso. Simplificando, uma forma de pensar e agir.

Agora, um paradigma de desenvolvimento de *software* serve para fornece uma visão sobre as estruturas da programação, isto é, as formas de se desenvolver um *software*. E para isso existem vários paradigmas, por exemplo, o imperativo, o lógico, o funcional, o orientado a objetos, o orientado a aspectos, entre outros.

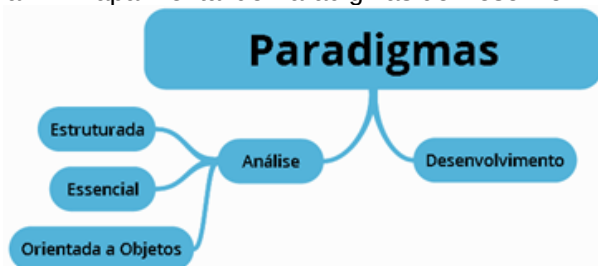
Dessa forma é possível se estabelecer um padrão de linguagem para um determinado nicho de problemas, por isso existem diversos paradigmas, além-claro, da própria evolução da ciência da computação, que de tempos em tempos, através de pesquisas, disponibilizam novas metodologias e paradigmas de programação.

Então se entende que certos grupos de linguagens foram e são desenvolvidas para determinado paradigma. Temos por exemplo Smalltalk para o Paradigma de Orientação a Objetos (POO), Scheme para o paradigma funcional. Mas, existem linguagens que suportam vários paradigmas. CSharp suporta tanto o POO como o Funcional, já o Java suporta o POO e o Aspecto, entre outras linguagens como Perl, Python, C++, etc.

A característica marcante dos paradigmas de programação são as restrições, isto é, o que pode ou não pode ser feito na linguagem. Por exemplo, a programação estruturada não aceita o comando de indireção *goto*, que é aceita no linear.

Para dar uma ideia do que veremos nessa disciplina segue na figura 2.1 um mapa mental que mostra sucintamente os vários tópicos abordados nessa disciplina.

Figura 2.1 Mapa Mental de Paradigmas de Desenvolvimento



Fonte: Próprio Autor

2.1 Paradigmas de Programação

Vamos aqui falar sobre alguns paradigmas. Entendido que as linguagens são criadas para atender os padrões definidos de um ou mais paradigmas, quais são as linguagens mais atuais e que paradigmas elas se baseiam.

2.1.1 Paradigma Funcional

É usado para criar programas cujo foco está na avaliação de funções matemáticas. Nesse paradigma existe a preocupação de se evitar estados ou dados mutáveis. Basicamente, a programação nas linguagens desse paradigma consiste em usar e criar funções, por exemplo: $f(x) = x + 1$, onde x é um parâmetro, e esse é um valor de entrada, que após ser processada a expressão matemática será fornecido o valor de saída da função. Vale salientar que as linguagens que atendem esse paradigma, são usadas na sua maioria no mundo acadêmico e de pesquisa, com pouquíssima aplicação comercial. Alguns exemplos de linguagens são: Lisp, Mathematica, Haskell e uma das mais recentes, encontrada na IDE Visual Studio Ultimate 2010 é o F#.

2.1.2 Paradigma Lógico

Também conhecido como Restritivo. De certa forma se assemelha ao funcional, pois o foco está na matemática, mais especificadamente na lógica matemática. As linguagens que fazem uso desse paradigma são orientadas a procedimentos, mas essas possuem limitações, tipo o armazenamento dos resultados dos procedimentos. Outra característica, tanto da lógica, como da funcional, é que elas não são uma sequência de afirmações e não possuem um estado global. Uma forma de entender as linguagens desses dois paradigmas, é que elas, normalmente definem o que é para ser processado, mas não como processá-lo. A linguagem mais representativa desse paradigma é o Prolog, mas como o padrão anterior, esse é muito pouco usado para aplicações comerciais, tendo um uso maior no mundo científico.

2.1.3 Paradigma Declarativo

Esse paradigma é baseado nos paradigmas funcional e lógico. Ele foi criado para determinar a fronteira entre os paradigmas já falados do imperativo, que será explicado a seguir. Mas, poderemos dizer que as linguagens declarativas, são aquelas que se descreve o que ela faz e não como seus procedimentos funcionam. Então, podemos entender que as linguagens de marcação, as que usam *tags*, são declarativas. Pois, elas descrevem o que são suas estruturas e não como elas são utilizadas. Um bom exemplo de linguagem declarativa é o HTML, outro exemplo muito utilizado é o SQL e mais recentemente o XML.

2.1.4 Paradigma Imperativo

É também denominado de modular ou procedural. Para entendê-lo, basta entender imperativo como comando, isto é, baseado em ações, que contrário aos três paradigmas anteriores, é focado na mudança de estado das variáveis. Exemplo: `if (vsexo == "F") { printf("Feminino"); }`. Observe que só haverá saída se o sexo for feminino, um comando. Ao contrário dos paradigmas anteriores, esse normalmente define como processar algo, mas não para que processá-lo, isso, dependerá da utilização do programa. As linguagens mais representativas desse paradigma são o C, Basic e o Pascal, porém uma muito utilizada no passado recente foi a Clipper e toda a linha das XBase.

2.1.5 Paradigma Orientado a Eventos

A maior característica desse paradigma é que no ato de programar não se determina o fluxo do controle, ficando a cargo do usuário do *software* desenvolvido a sua condução. Explicando melhor, nas linguagens do paradigma imperativo, tipo Clipper, Pascal,... O programador definia exatamente como a navegação iria acontecer. Para acessar tal opção, primeiro ir para o menu tal, depois submenu tal, e assim por diante. Já nas linguagens orientadas a eventos, o fluxo não era criado e sim as opções de acesso, deixando a cargo de o usuário acessar as opções, quando e como quiser, isto é, controle de fluxo de programas é definido por ocorrências externas ao *software*, e essas ocorrências são chamadas de eventos. Exemplo: um clique posiciona o cursor, um duplo clique seleciona uma palavra, um triplo clique seleciona um parágrafo, etc. Assim, o programador, define as ações, baseado nos eventos, ou melhor, quando elas devem acontecer. Mas, é o usuário que inicia o evento. Basicamente, toda a linguagem que faz uso de interface gráfica é baseada nesse paradigma. Como exemplo temos as linguagens Visual Basic, Delphi, FoxPro, SQLWindows, entre tantas outras.

2.1.6 Paradigma Orientado a Objetos

Também conhecida como POO, teve a sua motivação de criação e evolução a partir do paradigma Orientado a Eventos e se deve a necessidade da área de engenharia de *software* realizar a manutenção corretiva, adaptativa e evolutiva de forma mais rápida em relação aos paradigmas anteriores. Focado na criação

de classes e na instância de objetos dessas classes, a construção do *software* passa a ser mais criteriosa, pensando na possibilidade de reuso, isto é, reutilizar código pronto e testado. Além, de fazer as correções e implementações, de forma mais organizada e eficiente. Uma grande vantagem desse paradigma, é que ele une as três áreas básicas do desenvolvimento de aplicações, as quais são análise, programação e banco de dados. Permitindo que a mesma simbologia possa ser usada nessas três áreas. Nos dias atuais, quase todas as linguagens se baseiam nesse paradigma, umas criadas, outras adaptadas para ele. Como exemplos têm: C++, C#, Java, Object Pascal, Delphi, Delphi. NET, VB.NET, Python, Ruby e Smalltalk. Além do PHP a partir da versão quatro, do Perl a partir da versão cinco.

2.1.7 Paradigma Orientado a Aspectos

Também conhecida como POA, é um paradigma que permite a separação e organização do código de acordo com a sua importância para a aplicação, isso, passou a ser chamado de costurar o código. De certa forma, todas as aplicações desenvolvidas usando o paradigma POO, isto é, que possuem implementação de comportamento de um objeto, os chamados métodos, possuem um problema. Se não houver um cuidado na construção desse código, o mesmo pode ficar espalhado por toda a aplicação. É essa falha da POO que a POA entra, com o objetivo de resolvê-la. Permitindo que esse código seja encapsulado e modularizado. Para entender melhor, imagine um sistema escrito em Pascal, onde em várias partes do sistema é necessário fazer o registro da data, hora e usuário que acessou a rotina. Bom, em qualquer linguagem, seria criado um programa central que faria esse registro e onde fosse preciso, se criaria uma chamada para essa rotina. O problema que o uso dessa técnica, espalha a chamada da rotina, por todo o sistema. A proposta do POA é encapsular essas chamadas através de uma construção chamada de Aspect, onde esse se responsabilizaria pelo comportamento da aplicação que tem um comportamento adicional o chamado de *advice*, sobre um *joinpoint*. Atualmente a linguagem desse paradigma é o AspectJ, uma versão Java para a orientação a aspecto. Na figura 2.2 pode ser observar os nomes dos principais paradigmas de desenvolvimento de software.

Figura 2.2 Paradigmas



2.2 Linguagens de Programação

Vamos agora fornecer alguns conceitos básicos sobre as linguagens de programação, que tem como objetivo escrever uma lógica que seja reconhecida por uma máquina para executar algum conjunto de comandos produzindo um resultado.

Existem três elementos importantes que devem ser entendidos em qualquer linguagem - a sintaxe, a semântica e a simbologia (o léxico), que podem ser definidas como:

- Sintaxe que vem do grego e quer dizer disposição consiste nas regras que constituem a formação das sentenças nas linguagens de programação (o que significa algo);
- Semântica que também vem do grego e tem a significação de sinal, trata do significado, (como expressa algo); e
- Léxico determina o conjunto de símbolos de uma linguagem (que simboliza algo).

Agora, sem dúvida, um dos itens mais usados, são os identificadores ou variáveis, que objetivam armazenar na memória um valor ou expressão.

Para se armazenar em uma variável, quando temos uma linguagem tipada, temos que definir os tipos de dados, que é uma combinação de valores que podem ser armazenados em um identificador e dependem da análise léxica, sintática e semântica do compilador ou interpretador da linguagem.

Entendido isso, pode-se perguntar para que segmentos as linguagens são construídas? Seguem alguns:

- Inteligência Artificial é um dos mais recentes e tem por objetivo descobrir *software* e *hardware* que possam simular a capacidade humana de pensar;
- Educação que objetiva auxiliar no processo cognitivo;
- Ciência e Engenharia que tem como objetivo criar meios que otimizem processos de criação e construção;
- Sistemas de Informação onde seu uso é mais abrangente, pois tem o foco de auxiliar as organizações se tornarem mais eficientes e eficazes;

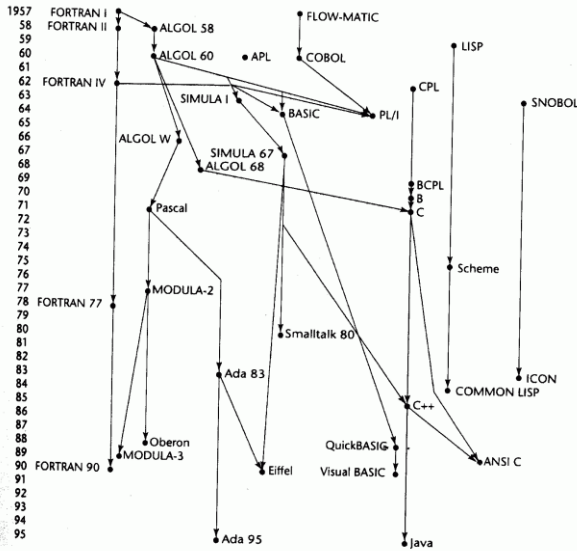
- Redes objetivando a convergência dos recursos tecnológicos; e
- Internet permitir a colaboração de informações e atividades.

Bom, até aqui vimos o "b-a-ba" das linguagens, mas como elas evoluíram? Na figura 2.3, existe um resumo dessa evolução compreendendo o período de 1957 até 1995.

Agora, clique na árvore genealógica das linguagens para ter uma ideia mais completa de quando e de onde as principais surgiram. No site <http://www.levenez.com/lang/> é possível ter um detalhamento dessa evolução.

Como vocês podem observar são muitas linguagens. E dessas quais são as mais populares? Para responder essa pergunta, estou disponibilizando o resultado de uma pesquisa entre várias já realizadas sobre o tema. Essa foi realizada pela TIOBE que é uma organização que oferece uma avaliação periódica sobre qualidade de *software*. Assim esse ranking pode ser usado para se conhecer algumas das linguagens mais populares do momento. Esse outro site ohloh possibilita fazer uma comparação entre as principais linguagens. Mas, para ser mais justo, segue outro ranking com as linguagens mais populares, realizado pelo site LangPop que divulgou mundialmente essa pesquisa na qual analisou a popularidade das principais linguagens de programação existentes no mercado, utilizando para isso o parâmetro de buscas realizadas nos principais sites de buscas da época como Google, Yahoo, Amazon, entre outros.

Figura 2.3 Linguagens



Fonte: <http://www.levenez.com/lang/>

Projetos de Linguagens

Como já vimos anteriormente neste livro os conceitos de projetos de linguagem, no capítulo anterior, seguem mais algumas informações.

Existe algumas características básicas necessárias serem trabalhadas para se projetar uma linguagem, essas são: a sua arquitetura, a configuração técnica, os seus padrões e a compatibilidade com o legado.

Então vamos entender basicamente o que é arquitetura de uma linguagem. Podemos dizer que, projetar uma linguagem é levar em consideração seu *hardware*, e o modelo que o define, o já citado modelo de von Neumann, trata-se do modelo, entrada-processamento-saída. Então, boa parte, das linguagens é projetada para essa arquitetura de máquina, mas existem outras.

Detalhando um pouco mais a questão de arquitetura, temos que as linguagens são projetadas para atendê-las, e nos dias atuais, são dois os projetos básicos de arquiteturas:

- *Complex Instruction Set Computer (CISC)*, traduzindo Computador com um Conjunto Complexo de Instruções. Explicando basicamente o que é, trata-se de um padrão de construção de processadores capazes de executar centenas de instruções complexas. Mas, o que quer dizer isso? Uma forma simples de explicar, que dizer que. No processador, são armazenadas instruções chamadas de microcódigo, que se não o existissem, teriam que

ser codificadas pelos programadores nas linguagens. Esse armazenamento de instruções nos processadores minimiza e simplifica o trabalho de desenvolvimento, já que muito código comum a qualquer tipo de programa já está embutido no processador. Como consequência direta as linguagens concebidas para essa arquitetura, tendem a ter um código executável bem menor. Mas, a sua desvantagem é que como a microprogramação realizada visa prever uma grande quantidade de funcionalidades comuns aos aplicativos, torna esse tipo de processador pesado em relação ao tempo de resposta. E para completar com o passar do tempo, notou-se que muitas dessas funcionalidades embutidas nos microcódigos nunca eram usadas na maioria das aplicações desenvolvidas. Um exemplo de padrão de processador CISC é o Intel 486. Resumindo temos: melhor código com pior desempenho; e

- *Reduced Instruction Set Computer (RISC)*, traduzindo Computador com um Conjunto Reduzido de Instruções. Trata-se de outro padrão de construção de processadores, que como a própria sigla já diz, oferece um conjunto simples e pequeno de instruções armazenadas no processador. Simplificando, os processadores desse padrão não possuem ou possuem muito pouca microcódigo. Consequência direta é que o esforço de desenvolvimento torna-se maior. Porém, como os processadores não possuem microcódigo desnecessário, prevendo funcionalidades comuns a diversos tipos de programas, que normalmente não são usadas, o tempo de resposta é muito superior. Grande parte dos projetos de processadores atuais segue esse modelo de arquitetura, a exemplo temos: DEC Alpha, o SPARC, o MIPS, o PowerPC, e os atuais x86 da Intel que usam uma arquitetura híbrida, meio CISC e meio RISC. Outro exemplo são as máquinas construídas para tipos específicos de aplicações como as de computação gráfica e de jogos. Resumindo temos: melhor desempenho com pior código.

Outra preocupação no projeto de uma linguagem de programação é em relação ao nicho que ela pretende atender, para resolver os seus problemas. Esse objetivo gera uma série de restrições, de natureza técnica ou de modelo, que são normalmente observados, durante o projeto de uma linguagem de programação. Essas restrições tecnológicas estão relacionadas aos níveis de abstração.

Também existe a preocupação no projeto de desenvolvimento de uma nova linguagem é a sua compatibilidade com os seus projetos anteriores, caso existam. Isso se deve pela necessidade de se poderem manter os códigos produzidos pelas versões anteriores com a nova linguagem. Uma tarefa muito complicada. Mas, a preocupação com o legado na área de TIC é uma constante.

Depois, que uma linguagem se torna bem aceita pelo mercado, mais uma preocupação se passa a ter que é a sua padronização. Essa padronização é

necessária, para uniformizar o uso e a evolução da linguagem. Normalmente as entidades que formalizam a padronização são a ANSI e o ISO, veja os *Links* para saber mais.

Uma categorização importantíssima nos projetos de linguagens de programação é a Hierarquia de Chomsky, que classifica as linguagens em tipos, os quais podem ser vistos abaixo: Tipo 3: Gramática Regular > Linguagem Regular > Autômato Finito; Tipo 2: Gramática Livre de Contexto > Linguagem LC > Autômato com Pilha; Tipo 1: Gramática Sensível ao Contexto > Linguagem SC > Autômato Linear; e Tipo 0: Gramática Irrestrita > Linguagem Recursiva > Máquina de Turing.

No capítulo Teoria Geral da Computação já esses tipos já foram conceituados, mas para revisar os conceitos, vamos falar sucintamente sobre eles. Começando com a explicação de Linguagens Formais e Autômatos. Basicamente é o estudo de modelos matemáticos que permitam definir e reconhecer linguagens, possibilitando a sua classificação, estruturação e caracterização. É conhecida como Teoria da Ciência da Computação. A finalidade do estudo dessas teorias é fornecer fundamentos para a Ciência da Computação em relação à decidibilidade, computabilidade e complexidade computacional. Essas teorias fundamentam várias áreas da computação, como processamento de linguagens, reconhecimento de padrões, modelagem de sistemas.

Agora o que é Linguagem Formal? Sabemos que linguagem é uma forma de comunicação e que com a evolução do homem, essas ganharam especializações, do tipo léxico, sintático e semântico. Essas especializações servem tanto para linguagens naturais ou idiomas, como também para linguagens de programação e dos protocolos de comunicação.

Assim, podemos dizer que as Linguagens Formais consistem de regras para representação e especificação de linguagens. As representações são feitas por reconhecedores e geradores de linguagens, que nada mais é, que um recurso que verificam se uma sentença pertence ou não a uma determinada linguagem. Esses recursos são chamados de autômatos, que podem ser: Autômatos Finitos Determinísticos; Autômatos Finitos Não Determinísticos; Autômatos de Pilha; e Máquina de Turing.

Já os geradores são recursos que permitem a geração sistemática de todas as sentenças de uma linguagem. Esses recursos são as gramáticas, a qual se destaca a de Chomsky. Resumindo, as linguagens formais podem ser representadas de maneira finita e precisa através de teorias matemáticas, o que caracteriza a Ciência da Computação.

Resumindo o Tipo 3, o das linguagens regulares, que podem ser representados pelos autômatos finitos determinísticos e não determinísticos. Essas

linguagens se baseiam nas Expressões Regulares. Essas expressões são representadas por três formas básicas: União, representada pelo símbolo (+) permitindo unir expressões; Concatenação, representada pelo símbolo (.) permitindo concatenar expressões; e Klenee, representada pelo símbolo (*) permitindo potencializar expressões.

Essas expressões podem ser representadas por Autômatos. E o que são? São máquinas de estados finitos, isto é, um autômato finito. Trata de uma modelagem de um comportamento, composto por estados, transições e ações. Um estado armazena informações sobre o passado, isto é, ele reflete as mudanças desde a entrada num estado, no início do sistema, até o momento presente. Uma transição indica uma mudança de estado e é descrita por uma condição que precisa ser realizada para que a transição ocorra. Uma ação é a descrição de uma atividade que deve ser realizada num determinado momento. Máquinas de estados finitos podem ser representadas por meio de um diagrama de estados ou diagrama de transição de estados.

Resumindo o Tipo 2, o das linguagens livre de contexto, essas são representadas por autômatos com pilhas. Que são idênticos aos autômatos finitos determinísticos, exceto pelo fato eles adicionalmente usam o recurso de memória sob a forma de pilhas. A função de transição δ irá depender agora dos símbolos que estão no topo da pilha, e especificarão como a pilha deve ser modificada a cada transição.

Resumindo o Tipo 1, o das linguagens sensíveis ao contexto, essas são representadas por uma forma restrita de uma máquina de Turing não determinística que satisfaça as duas condições seguintes: Seu alfabeto de entrada inclui dois símbolos especiais, servindo como marcadores de direção da esquerda e direita; e Necessitam de um ponteiro que se move para a esquerda levando além do marcador.

Resumindo o Tipo 0, o das linguagens recursivas, essas são representadas por máquinas como as de Turing, Moore e Mealy. A Máquina de Turing é um recurso teórico conhecido também como máquina universal, que foi concebido pelo matemático britânico Alan Turing, antes do surgimento dos modernos computadores digitais. É um modelo abstrato de computador, que trata apenas dos aspectos lógicos de funcionamento da memória, estados e transições. Porém com essa abstração pode-se modelar qualquer computador digital. A Máquina de Moore é outro recurso que utiliza apenas ações de entrada e a saída depende somente do estado. A vantagem do modelo de Moore é a simplificação do comportamento. Por exemplo, modelar um elevador com quatro estados possíveis para sua porta "Aberta", "Fechada", "Abrindo", "Fechando". A máquina de estados reconhece dois comandos: "comando_abrir" e "comando_fechar" que disparam a alteração de estado. A ação de entrada no estado "Abrindo" liga o motor que abre a porta, a ação de entrada no estado "Fechando" liga o motor na outra direção, fechando a porta. Os estados "Aberta"

e "Fechada" não desempenham nenhuma ação. E a Máquina de Mealy é mais um recurso onde a saída depende da entrada e do estado. O uso de uma máquina Mealy normalmente leva a uma redução no número de estados. O exemplo anterior nessa máquina seria: inicie o motor para fechar a porta se o "comando_fechar" chegar e inicie o motor na direção oposta para abrir a porta se o "comando_abrir" chegar.

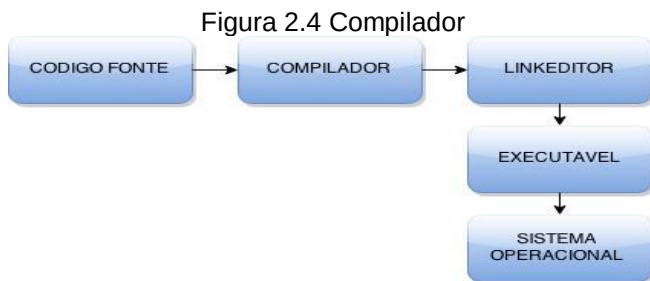
Finalizando, os principais objetivos de um projeto de linguagem de programação são a obtenção da: Simplicidade; Legibilidade; Clareza; Confiabilidade; Suporte; Abstração; Ortogonalidade, e Eficiência.

2.3 Compilador

O compilador é um programa que foi projetado para ler outro programa chamado de código fonte, escrito em uma linguagem específica para um compilador chamado de linguagem compilada.

A finalidade do compilador é gerar um terceiro programa semanticamente equivalente ao código fonte chamado de código objeto, que normalmente será o equivalente a linguagem de alto nível utilizada na sua versão em linguagem Assembly ou de máquina.

Durante a compilação são realizados vários processos chamados de análise e síntese. Na análise tem-se o objetivo de compreender o código fonte escrito, através das verificações léxica, sintática e semântica. Na síntese é gerado o código objeto, isto é, o código intermediário. Basicamente o processo é o seguinte: compila-se o código fonte e é realizada a análise léxica dessa é gerado o código simbólico (*tokens*), então se passa para o próximo passo que é a análise sintática e dessa é gerado o código analisado (*parse*), depois a análise semântica na qual é gerado o código qualificado, para daí ser gerado o código objeto, esse por fim é otimizado para gerar o código final.



Fonte: Próprio Autor

Um programa que funciona como complemento do compilador é o Linkeditor, que possui a tarefa de linkar (unir) vários programas compilados em um programa chamado de executável. Permitindo assim que o programa rode no sistema operacional para qual o compilador / Linkeditor foi feito.

[Clique aqui para visualizar uma lista de linguagens compiladas.](#)

2.4 Interpretador

O Interpretador é um programa projetado para ler um código fonte escrito em uma linguagem específica para um interpretador chamado de linguagem interpretada.

O objetivo é ler linha a linha e converte em código objeto à medida que vai executando o programa, dessa forma o interpretador executa o código fonte linha a linha.



Fonte: Próprio Autor

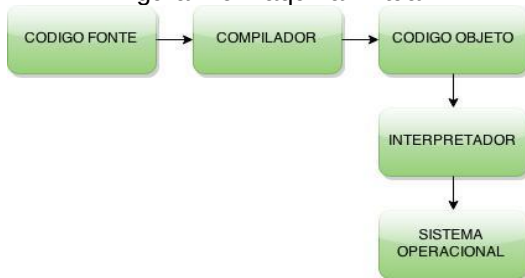
[Clique aqui para visualizar uma lista de linguagens interpretadas.](#)

2.5 Máquina Virtual

Nos dias atuais existem compiladores que geram código para o que é chamada máquina virtual, isto é, um misto de compilador e interpretador. As linguagens que se encaixam nessa categoria podem ser chamadas de linguagens híbridas. Isso quer dizer que existem linguagens que são parte compilada e parte interpretada. Assim o compilador tem a função de converter o código fonte em um código intermediário chamado de *bytecode*, que nada mais é, do que uma linguagem de baixo nível e este será interpretada por uma máquina virtual, que funciona como um interpretador.

A principal diferença entre uma linguagem compilada para uma interpretada e essas duas para uma híbrida é que na primeira todo o código fonte é lido pelo compilador para gerar de uma única vez um código final. Na segunda, a interpretada, o código fonte é lido linha a linha e o interpretador executa um comando por vez. E na última, temos um pouco de cada, já que o compilador analisa todo o código fonte, mas não gera o código final e sim um intermediário chamado de código objeto, e esse por sua vez é interpretado pela máquina virtual.

Figura 2.6 Máquina Virtual



Fonte: Próprio Autor

2.6 Análises dos Compiladores

A análise léxica consiste de um processo de verificação das entradas dos caracteres com o objetivo de gerar uma sequência simbólica chamada de símbolos léxicos ou *lexicais tokens*, para que possam ser manipulados pelo analisador sintático chamado de *parser* que tem a finalidade de verificar as saídas. Resumindo a análise léxica tem a função de verificar um determinado alfabeto. Exemplo, quando analisamos uma palavra, pode definir através da análise léxica se existe ou não algum caractere que não faz parte do nosso alfabeto.

A análise sintática, mais conhecida como *parsing*, é o processo de verificar uma sequência de símbolos para determinar se sua estrutura gramatical segue as regras gramaticais definidas pela linguagem. Resumindo a análise sintática transforma um texto na entrada em uma estrutura geralmente em forma de árvore, para que fique mais fácil o entendimento e o processamento, na verificação da hierarquia da linguagem. A grande maioria dos analisadores sintáticos utiliza alguma linguagem livre de contexto para fazer a análise e esses analisadores são classificados como *Left-Left* (LL) ou *Left-Right* (LR). O LL é um algoritmo de análise sintática que realiza a verificação no sentido cima para baixo ou *top-down*, já o LR faz a verificação de baixo para cima ou *bottom-up*.

Análise semântica tem a finalidade de verificar os erros semânticos. Exemplo, uma operação de soma entre operandos de tipos diferentes com um numérico e outro literal.

[Clique aqui para visualizar a especificação da Máquina Virtual Java. Máquina Virtual \(wiki\).](#)

[Clique aqui para visualizar a especificação do .NET Framework. DOT NET \(wiki\).](#)

2.7 Princípios dos Paradigmas de Desenvolvimento

As gramáticas que compõem as linguagens de programação são: léxica, sintática e semântica. Que respondem respectivamente por: analisar os símbolos, a escrita e o sentido das palavras de uma linguagem de programação.

As linguagens possuem o recurso básico de nomes de variáveis (identificadores) e tipos de dados. Respectivamente servem para denominar um local de memória que irá armazenar um dado e esse por sua vez deve possuir um tipo específico: número, literal, etc.

Os como já visto os paradigmas de desenvolvimento são padrões que determinam regras de construção de linguagens, tipo orientado a objetos, imperativos, etc.

Três conceitos especiais das linguagens de programação são: Eventos; Concorrência; e Correção. Que correspondem respectivamente por: Ações que ocorreram em um determinado momento; Quando ações podem ocorrer ao mesmo tempo de forma assíncrona; e Trata-se de desenvolver focado na qualidade, isto é, a manutenção corretiva e/ou evolutiva que tem o objetivo de gerar o menos impacto na construção do produto de *software*.

Existem diversos segmentos que os paradigmas de desenvolvimento são pensados para a área de desenvolvimento de *software*, mas podemos salientar os seguintes:

- Inteligência Artificial (IA) que tem como objetivo desenvolver *software* que possa aprender e assim evoluir por inferência, e uma forma de exemplificar essa lógica são com o conceito de redes neurais, que fazem uso do algoritmo MLP;
- Educação (EaD), nos dias atuais, ou melhor, na era da informação, um dos grandes problemas está no tempo, isto é, falta de tempo, nesse ponto a TIC também contribui possibilitando o uso de sistemas cooperativos ou colaborativos para disponibilizar conhecimento para quem não tem tempo para acessá-lo de forma tradicional, e o Ensino a Distância tem essa característica ;
- Ciência e Engenharia, inevitavelmente são nessa área que o emprego de TIC é mais evidente, muito devido a sua própria natureza científica;
- Sistemas de Informação (SI), é o emprego da TIC nas organizações buscando torná-las mais eficientes, eficazes e assim obter sua efetividade;
- Sistemas e Redes têm seu foco na tecnologia, possibilitando o grau de compartilhamento existente no mundo moderno; e

- Internet que é o foco do desenvolvimento atual, pois permite que o conhecimento e as necessidades sejam compartilhados e atendidos.

Para construir uma linguagem de programação algumas restrições ou necessidades devem ser observadas: Arquitetura, se o esforço de programar deve ser árduo em benefício ao poder de processamento - RISC ou se o esforço de programar deve ser atenuado em detrimento a desempenho da aplicação - CISC; Configurações Técnicas, levar em consideração a integração dos recursos da linguagem como, por exemplo, o uso de IDE's; Padrões, fazer uso de instituições ou organizações reguladoras, para padronizar a evolução da linguagem, exemplos: ISO, ANSI, OMG,...; e Sistemas Legados, que é um fator primordial na construção de novas linguagens, principalmente se for à evolução uma linguagem já existente, para que se mantenha a compatibilidade com que já foi criado.

Os objetivos que se almejam no desenvolvimento de linguagens são: Simplicidade e Legibilidade, para que se possa escrever pouco e fazer muito, sendo principalmente de fácil entendimento; Confiabilidade, para que o que se codifique gere o resultado esperado; Suporte, que a linguagem possui recursos para sanar dúvidas caso surjam; Abstração, que se possa focar no que é relevante, deixando as particularidades para as especializações da linguagem; Ortogonalidade, que o entendimento do código seja claro, caso a necessidade de leitura seja rápida, e isso só é possível se a linguagem for criada a partir de primitivas simples; e por fim a Eficiência, isto é a linguagem construir o que se propõe de forma rápida.

As linguagens de programação são ferramentas construídas para desenvolver outras ferramentas e essas precisam ser compreendidas pelos Sistemas Operacionais (SO's) que controlam a máquina (*hardware*). Para se chegar a um código-executável a partir de um código-fonte temos as seguintes formas:

- Linguagens Interpretadas: são aquelas que o código-fonte é analisado linha-a-linha pelo interpretador toda vez que o programa for executado, resultando em tempo de execução baixo;
- Linguagens Compiladas: são aquelas que todo o código-fonte é analisado gerando um código-final que toda vez que for executado não precisar ser mais analisado, resultando em um tempo de execução alto, porém com o código atrelado ao Sistema Operacional; e
- Linguagens Híbridas (compiladas-interpretadas) Máquinas Virtuais: são aquelas que todo o código-fonte é analisado e é gerado um código-objeto (*bytecode*) que para ser entendido pela máquina tem que ser interpretado por uma máquina-virtual, sendo assim mais rápido que o interpretado, mais lento que o compilado, porém portátil para qualquer Sistema Operacional.

2.8 Princípios de Linguagens de Programação

As linguagens de programação em sua maioria possuem alguns itens, ou melhor, recursos que são importantes o seu conhecimento. Um exemplo é saber distinguir o que é procedimento e função.

Um procedimento é um bloco de comando que não retorna um valor. Podemos exemplificar com o comando CLS do sistema operacional MS-DOS que faz a ação de limpar a tela, mas não retorna nada. A própria ação já é o retorno, pois se não limpar a tela não funcionou, caso contrário tudo bem. Exemplificando, podemos observar nos trechos de código abaixo, feitos em linguagens diferentes representando um procedimento.

Exemplo em Pascal

```
procedure exProc(p1: integer; p2: integer);  
  // será exibido na tela o resultado da operação, mas não será retornado  
  nenhum valor  
  // para quem chamou esse procedimento  
  var  
    v1: integer;  
  begin  
    v1 := p1 + p2;  
    write('A soma é: ' + IntToStr(v1));  
  end;
```

Exemplo em Java

```
public static void exProc(int p1, int p2) {  
  // será exibido na tela o resultado da operação, mas não será retornado  
  nenhum valor  
  // para quem chamou esse procedimento (método sem retorno)  
  int v1;  
  v1 = p1 + p2;  
  System.out.print("A soma é: " + v1.ToString());  
}
```

Uma função também é um bloco de comando, porém que retorna um valor. Podemos exemplificar com a função DATE do sistema operacional MS-DOS que faz a ação de obter a data corrente e a retorna exibindo-a na tela. Exemplificando, podemos observar nos trechos de código abaixo, feitos em linguagens diferentes representando uma função.

Exemplo em Pascal

```
function exFunc(p1: integer; p2: integer): integer;
```

```
// será retornado para quem chamou esse procedimento o resultado da
operação
var
  v1: integer;
begin
  v1 := p1 + p2;
  exFunc = v1;
end;
```

Exemplo em Java

```
public static int exFunc(int p1, int p2) {
  // será retornado para quem chamou esse procedimento o resultado da
  operação
  int v1;
  v1 = p1 + p2;
  return (v1);
}
```

Agora observem que em todos os exemplos tanto os procedimentos como as funções recebem valores em variáveis chamadas p1 e p2 do tipo inteiro, para serem processados. Essas variáveis recebem o nome de parâmetros e tem o objetivo de receber um valor passado por um programa chamador. Vejam o exemplo abaixo nas duas linguagens já utilizadas de programas que chamam as funções exFunc para exibir o resultado da operação.

Exemplo em Pascal

```
program ExPascal;
var
  vx: integer;
  vy: integer;
begin
  vx := 2;
  vy := 3;
  write('A soma é: ' + exFunc(vx, vy));
  // o resultado exibido é 5, já que foram passado 2 e 3 como parâmetros
end.
```

Exemplo em Java

```
public static void main(String Args[]) {
  int vx;
  int vy;
  vx = 3;
```

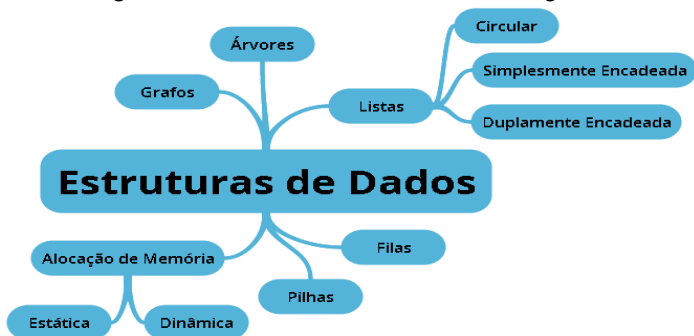
```

vy = 5;
System.out.print("A soma é: " + exFunc(vx, vy));
// o resultado exibido é 8, já que foram passado 3 e 5 como parâmetros
}

```

Porém, as passagens de parâmetros basicamente podem ser de dois tipos. Um chamado de passagem por valor, que é o mais comum, e o outro de passagem por referência, menos usado. E qual a diferença? Por valor o procedimento ou a função recebem uma cópia da variável que é fornecida quando é invocada. O principal é que todas as alterações feitas dentro do bloco de comandos da função ou procedimento não vão afetar os valores originais da variável origem, já que o que foi passado foi uma cópia. Por referência a função ou o procedimento recebem é uma referência das variáveis utilizadas, e não uma cópia, isto quer dizer, que o que é passado é o endereço de memória da variável origem e decorrente disso as alterações realizadas nas variáveis dentro do bloco de comandos da função ou procedimento irão afetar os valores originais, já que se está usando o mesmo local de memória das rotinas chamadoras. Para entender melhor, observe a figura 2.7 e veja os exemplos abaixo feitos nas linguagens C, C++, Pascal e Java.

Figura 2.7 Conceitos Gerais de Paradigmas



Fonte: Próprio Autor

Passagem por valor em C

```

#include<stdio.h>
void troca(int v1, int v2) {
    int aux;
    aux = v1;
    v1 = v2;
    v2 = aux;
}
int main() {
    int v1 = 2, v2 = 3;
    printf("Antes de chamar a função : \n v1 = %d \n v2 = %d \n", v1, v2);
}

```

```
troca(v1,v2);
printf("Depois de chamar a função:\n v1 = %d\n v2 = %d\n",a,b);
return 0;
// mesmo o procedimento troca ter feito a troca dos valores de v1 e v2 o que é
// exibido antes é o mesmo que é exibido depois, pois a troca só ocorreu
dentro
// do procedimento, visto que a passagem foi por valor e sendo assim o que é
// enviado é uma cópia das variáveis v1 e v2.
}
```

Passagem por referência em C

```
#include<stdio.h>
void troca(int *v1, int *v2) {
    int aux;
    aux = *v1;
    *v1 = *v2;
    *v2 = aux;
}
int main() {
    int v1 = 2, v2 = 3;
    printf("Antes de chamar a função :\n v1 = %d\n v2 = %d\n",v1,v2);
    troca(&v1,&v2);
    printf("Depois de chamar a função:\n v1 = %d\n v2 = %d\n",a,b);
    return 0;
    // a passagem por referência na linguagem C é representada pelo uso de
    // '&' e comercial' o que faz com que os valores sejam passados através da
    // referência de memória das variáveis, e assim o sendo, o que é exibido antes
    // é diferente do que é exibido depois já que o procedimento troca,
    // inverteu os valores das variáveis. Um detalhe é que em C para se manipular
    // variáveis passados por referência, deve-se usar o '*' = asterisco' antes delas.
}
```

Passagem por valor em C++

```
#include<stdio.h>
void troca(int v1, int v2) {
    int aux;
    aux = v1;
    v1 = v2;
    v2 = aux;
}
int main() {
    int v1 = 2, v2 = 3;
    cout<<"Antes de chamar a função :\na = " << v1<< " \nb = " << v2 << endl;
    troca(a,b);
}
```

```

cout<<"Depois de chamar a função : \na = " << v1 << " \nb =" << v2 <<endl;
return 0;
// mesmo o procedimento troca ter feito a troca dos valores de v1 e v2 o que é
// exibido antes é o mesmo que é exibido depois, pois a troca só ocorreu
dentro
// do procedimento, visto que a passagem foi por valor e sendo assim o que é
// enviado é uma cópia das variáveis v1 e v2.
}

```

Passagem por referência em C++

```

#include<stdio.h>
void troca(int &v1, int &v2) {
    int aux;
    aux = v1;
    v1 = v2;
    v2 = aux;
}
int main() {
    int v1 = 2, v2 = 3;
    cout<<"Antes de chamar a função : \na = " << v1<< " \nb = " << v2 << endl;
    troca(a,b);
    cout<<"Depois de chamar a função : \na = " << v1 << " \nb =" << v2 <<endl;
    return 0;
    // a passagem por referência na linguagem C++ é representada pelo uso de
    // '&' e comercial' o que faz com que os valores sejam passados através da
    // referência de memória das variáveis, e assim o sendo, o que é exibido antes
    // é diferente do que é exibido depois já que o procedimento troca,
    // inverteu os valores das variáveis.
}

```

O outro recurso importante nas linguagens de programação é a recursividade, porém nem todas possuem esse importante recurso. Agora o que é recursividade? Podemos dizer, que é a capacidade de uma função se auto-chamar. Isso otimiza a lógica de programação em muitos casos. O exemplo clássico é a criação da função fatorial. Veja o exemplo do uso da função matemática fatorial $5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$ escrita em pascal:

Fatorial em Pascal

```

program TestaFatorial;
var
    num, fat: integer;
function Fatorial (num:integer): integer;
begin
    if (num = 0) then
        begin

```

```
        Fatorial := 1;
    end
    else
    begin
        Fatorial := num * Fatorial(num - 1);
    end;
end;
begin
    writeln('entre com um número inteiro');
    readln(num);
    fat := fatorial(num);
    writeln('o fatorial de ',num,' é ',fat);
end.
```

Fatorial em Java

```
public class ExemploFatorial {
    public static void main (String args[]) {
        for(long i=0; i<=10; i++) {
            System.out.println(i + "! = " + fatorial(i));
        }
    }
    public static long fatorial(long num) {
        if (num <=1) {
            return (1);
        }
        else {
            return (num * fatorial(num - 1));
        }
    }
}
```

Para finalizar seguem mais dois recursos chamados pilhas e filas que serão detalhados na disciplina de Estrutura de Dados, mas para que se tenha um conhecimento breve segue suas descrições:

Pilha é uma estrutura computacional que funciona como uma pilha de anilhas, onde o Primeiro que Entra é o Último que Sai, que em inglês é chamado de *Last In First Out*, porém mais conhecido pelo acrônimo LIFO.

Figura 2.8 Estrutura de Pilha



Fila é outra estrutura computacional que funciona como uma fila de banco, onde o Primeiro que Entra é o Primeiro que Sai, que em inglês é chamado de *First In First Out*, mas mais conhecido como FIFO.

Figura 2.9 Estrutura de Fila



Esses conceitos são amplamente utilizados na computação, como, por exemplo, durante a execução de um programa, para o armazenamento de valores de variável local a um bloco e também para conter o endereço de retorno do trecho de programa que chamou a função ou procedimento atualmente em execução.

2.9 Backus-Naur (BNF)

Formalismo de Backus-Naur (BNF) é uma meta-sintaxe usada para expressar gramáticas livres de contexto, isto é, um modo formal de descrever linguagens formais.

A BNF é amplamente usada como uma notação para as gramáticas de linguagens de programação, conjuntos de instruções e protocolos de comunicação, e também como notação para representar partes de gramáticas de linguagens naturais.

[Clique aqui para ver exemplo \(BNF\) de Gramática da Linguagem Java](http://cui.unige.ch/isi/bnf/JAVA/BNFIndex.html)

Disponível em: < <http://cui.unige.ch/isi/bnf/JAVA/BNFIndex.html> >

2.10 Conceitos Básicos do Paradigma Orientado a Objetos

Classe O diagrama de classes na Programação Orientada a Objetos (POO), os problemas são pensados e expressados como objetos, ao contrário da análise tradicional os quais eram em rotinas e dados, que aqui foram substituídos por métodos (comportamento) e atributos (propriedades). Assim, quando é colocado o problema de desenvolver um sistema acadêmico na análise orientada a objetos, deve-se pensar como dividir esse problema em objetos.

Assim teríamos Alunos, Administradores, Professores, Cursos, Turmas, etc. E a melhor maneira de conceituar estes termos é considerar um objeto do mundo real e mostrar como podemos representá-lo em termos conceituais em POO. Assim, um objeto é um conceito que usamos para representar uma entidade do mundo real. Exemplificando, um Aluno possui nome, data de nascimento, identidade, etc. E além dessas características (propriedades), possuem ações (métodos) como frequentar as aulas, fazer as avaliações, etc. Em termos de POO para podermos tratar os objetos temos que criar classes. Assim, uma classe representa um conjunto de objetos que possuem comportamentos e características comuns.

Na UML em primeiro lugar denomina-se uma classe e recomenda-se nomeá-la capitalizando-a e deixando no singular. E em segundo lugar denominam-se as propriedades, informações específicas relacionadas a uma classe de objeto, isto é características dos objetos que as classes representam.

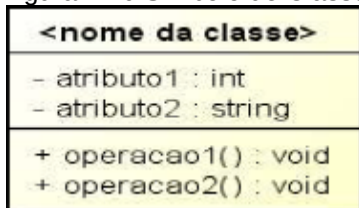
Em terceira e última parte, métodos, que são ações que os objetos de uma classe podem realizar. Assim tem-se um modelo para instanciar quantos objetos forem necessários. Dessa forma os objetos instanciados possuirão todas as características e comportamentos definidos pela classe. Então as classes especificam a estrutura (propriedades) e os comportamentos (operações) dos objetos, que são instâncias das classes.

Geralmente em um sistema de médio porte serão identificadas diversas classes que compõem o sistema. Neste contexto a UML surgiu como uma proposta de ser uma linguagem para modelagem de dados que usava diversos artefatos para representar o modelo de negócio; um destes artefatos é o diagrama de classes.

Os diagramas de classes registram atributos e operações de uma classe e as restrições de como os objetos podem ser conectados, descrevendo também os tipos de objetos no sistema e os relacionamentos entre eles e esse podem ser associações e abstrações. Para poder representar a visibilidade dos atributos e operações, a UML usa os seguintes símbolos:

- + público, visível em qualquer classe;
- - privado, visível somente dentro da classe;
- # protegido, na classe e suas subclasses.

Figura 2.10 Símbolo de Classe

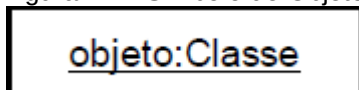


Fonte: Próprio Autor

Objeto Agora que sabemos o conceito e como codificar uma classe, vamos entender sobre objeto.

Segundo os pais da UML (Rumbaugh, Jacobson e Booch) um objeto é uma instância de uma classe, isto é, trata-se de uma cópia da classe na memória em tempo de execução (*runtime*), sendo assim, um elemento específico que possui valores (estados) nos atributos (campos).

Figura 2.11 Símbolo de Objeto



Fonte: Próprio Autor

A UML representa um objeto usando um retângulo que o representa, onde o nome da instância do objeto é seguido de dois pontos, seguido do nome da classe, com essa formação sublinhada.

Herança O relacionamento entre classes retrata as relações entre os objetos. Exemplo: um professor ministra uma disciplina para alunos numa sala. A UML reconhece três tipos mais importantes de relações: dependência, associação e generalização ou herança. Geralmente as classes não estão isoladas (coesas) e se relacionam entre si (acoplamento). O relacionamento e a comunicação entre as classes se definem em três tipos: Associações que podem ser de Agregação ou Composição; Generalização ou herança; e Dependências.

As associações são relacionamentos estruturais entre instâncias e especificam que objetos de uma classe estão ligados a objetos de outras classes, podendo ser unária, binária, etc. As associações podem existir entre classes ou entre objetos. Uma associação entre a classe Professor e a classe Disciplina (um professor ministra uma disciplina) significa que uma instância de Professor (um professor específico) vai ter uma associação com uma instância de uma Disciplina. Esta relação significa que as instâncias das classes são conectadas, seja fisicamente ou conceitualmente.

Polimorfismo É a capacidade de classes mais abstratas possuírem comportamentos diferentes das classes concretas, isto é, duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação (assinatura), mas comportamentos distintos, especializados para uma das classes derivadas. A decisão sobre qual o método que deve ser selecionado, de acordo com o tipo da classe derivada, é tomada em tempo de execução, através do mecanismo de ligação tardia.

Mas para ser considerado polimorfismo, é necessário que os métodos tenham exatamente a mesma identificação, quer dizer, o mesmo nome de método, sendo utilizado o mecanismo de redefinição de métodos, isto é, sobrescrita dos métodos.

Sobreescrita (Override) a redefinição ocorre quando um método cuja assinatura já tenha sido especificada recebe uma nova definição, ou seja, uma nova implementação em uma classe derivada. O mecanismo de redefinição, juntamente com o conceito de ligação tardia, é a chave para a utilização do polimorfismo. É importante observar que, quando polimorfismo está sendo utilizado, o comportamento que será adotado por um método só será definido durante a execução.

Sobrecarga (Overload) um método aplicado a um objeto é selecionado para execução através da sua assinatura e da verificação a qual classe o objeto pertence. Dois ou mais métodos de uma mesma classe podem ter o mesmo nome, desde que suas listas de parâmetros sejam diferentes, constituindo assim uma assinatura diferente. Tal situação não gera conflito, pois o compilador é capaz de detectar qual método deve ser escolhido a partir da análise dos tipos de argumentos do método.

Encapsulamento Consiste na separação de aspectos internos e externos de um objeto. Este mecanismo é utilizado amplamente para impedir o acesso direto ao estado de um objeto (seus atributos), disponibilizando externamente apenas os métodos que alteram estes estados.

2.11 Exemplos

BNF da Gramática do Java. Em < <http://cui.unige.ch/isi/bnf/JAVA/BNFindex.html> >

Exemplo de Compilador Fácil (em Java). Em < <http://goo.gl/PnHW7p> >

Exemplo de Compilador Simplíssimo (em CSharp). Em < <http://goo.gl/Dpf1wK> >

Exemplo de Compilador Simplíssimo (em Java). Em < <http://goo.gl/aikTAL> >

Exemplo de Compilador Muito Simples (em Java). Em < <http://goo.gl/8YQbik> >

Exemplo de Compilador Simples (em Java). Em < <http://goo.gl/8YQbik> >

Exemplo de Compilador Completo (em Java). Em < <http://goo.gl/Mglwlt> >

Exemplo de Autômatos (em Java). Em < <http://goo.gl/Vudw8L> >

Exemplo de Autômato com Pilha (em Java). Em < <http://goo.gl/l8yC8p> >

2.12 Ferramentas

Case-Tools. Em < <http://case-tools.org/uml.html> >

Astah. Em < <http://astah.change-vision.com/en/index.html> >

Simulação de Autômatos. Em < <http://goo.gl/kwf3lo> >

2.13 Vídeos

Compiladores. Em < <http://vimeo.com/112860457> >

2.14 Sugestões de Leitura

Introdução a Informática. Em < <http://goo.gl/LPT4Sc> >

Engenharia de Software. Em < <http://goo.gl/xk0CqE> >

Fundamentos de Programação. Em < <http://goo.gl/xwYOfC> >

UML Guia do Usuário. Em < <http://goo.gl/av5BTh> >

Análise e Projeto com UML. Em < <http://goo.gl/Q5Hp7q> >

Utilizando UML e padrões. Em < <http://goo.gl/1YPJ0l> >

2.15 Referências Web

A Theory of Object-Oriented Design. Em < <http://goo.gl/fq3q5> >

ANSI site oficial. Em < <http://www.ansi.org/> >

ANSI wiki. Em < <http://goo.gl/ZCrMxk> >

Aulas de UML. Em < <http://goo.gl/2TRbUF> >

Design Patterns Languages for OOP. Em < <http://www.vincehuston.org/dp/> >

Exemplo de Modelagem UML com a ferramenta ASTAH. Em <

<http://goo.gl/JvghSt> >

Exemplo de Modelagem UML. Em < <http://goo.gl/zw0YCF> >

Guia Básico sobre UML. Em < <http://goo.gl/eGhjzM> >

IBM Rational Unified Process. Em < <http://goo.gl/ugl9H7> >

ISO site oficial. Em < <http://www.iso.org/iso/home.html> >

ISO wiki. Em < <http://goo.gl/vjWG> >

Langpop. Em < <http://www.langpop.com/> >

Mais sobre relacionamentos de Inclusão e Extensão. Em < <http://goo.gl/uH8X6Z> >

Manuais de Referência OMG-UML. Em < <http://www.omg.org/spec/UML/> >

Object Management Group (OMG). Em < <http://www.omg.org/> >

Object-Oriented Programming (OOP). Em < <http://goo.gl/AGVpPX> >

Ohloh. Em < <http://www.ohloh.net/> >

Rational Unified Process (RUP). Em < <http://www-01.ibm.com/software/br/rational/> >

Tiobe. Em < <http://www.tiobe.com/index.php/content/company/Home.html> >

Unified Modeling Language (UML). Em < <http://www.uml.org/> >

Wiki paradigma de desenvolvimento. Em < <http://goo.gl/r1YFIs> >

Wiki paradigma. Em < <http://pt.wikipedia.org/wiki/Paradigma> >

2.16 Referências Bibliográficas

AHO, Alfred; *et al.* Compilers: Principles, Techniques, and Tools. 2ªed. US: Pearson, 2006.

FERNANDES, Daniel. Análise de Sistemas Orientada ao Sucesso. 1ªed. Rio de Janeiro: Ciência Moderna, 2005.

LOPES, Filomena; *et al.* Desenvolvimento de Sistemas de Informação, Métodos e Técnicas, 2005.

LOUDEN, Kenneth. Compiladores: Princípios e Práticas. 1ªed. Pioneira Thomson Learning, 2004.

MEDEIROS, Ernani. Desenvolvendo Software com UML 2.0: Definitivo. 1ªed. São Paulo: Makron Books, 2004.

MELO, Ana. Desenvolvendo Aplicações com UML 2.2: do conceitual à implementação. 3ªed. RJ: Brasport, 2010.

RUMBAUGH, James; *et al.* Modelagem e Projetos Baseados em Objetos com UML. 2ªed. RJ: Campus, 2006.

RUMBAUGH, James; JACOBSON, Ivar; BOOCH, Grady. UML: Guia do Usuário. 1ªed. RJ: Campus, 2006.

SCHACH, Stephen. Engenharia de Software: Os Paradigmas Clássico e Orientado a Objetos. McGraw-Hill, 2008.

SOMMERVILLE, Ian. Engenharia de Software. 8ªed. Pearson, 2007.

TUCKER, Allen; NOONAN, Robert. Linguagens de Programação: Princípios e Paradigmas. 2ªed. McGraw, 2009.

WAZLAWICH, Raul. Análise e Projeto de Sistemas de Informação Orientados a Objetos. RJ: Campus, 2011.

3 Lógica de Programação

Programar um computador não é uma tarefa tão difícil. Difícil é programar as nossas cabeças.

O computador, na realidade, é uma réplica bastante simplificada de nossa mente, com apenas uma diferença essencial: a auto-programação. A finalidade desse capítulo é introduzir nos conceitos básicos de programação, porém esta no paradigma estruturado que possui menos conceitos que o orientado a objetos, para então no próximo iniciar a introdução a linguagem de programação Java nos conceitos de POO.

Ao longo de nossas vidas, aprendemos coisas a partir de alguns elementos fundamentais como cores, símbolos, sons, cheiros, e sentimentos. Esses elementos vão se combinando através de um processo conhecido como aprendizado.

O processo de aprendizado nada mais é do que uma auto-programação de nossas mentes. “Alguém” diz para nosso cérebro o que ele tem que fazer ao se deparar com uma situação específica, tal como: dobrar uma esquina, parar ao perceber o sinal vermelho, escovar os dentes ao acordar, em fim - tudo o que fazemos é regido por programas que, um dia, nós mesmos desenvolvemos.

Ora, se somos capazes de desenvolver programas para um aparelho tão complexo e genial como o nosso cérebro, por que não um simples computador? Este é o nosso desafio. Desenvolver as técnicas necessárias para conseguirmos codificar programas em um computador digital.

Programar um computador, assim como a nossa mente, requer duas exigências básicas: Conhecer a linguagem na qual se vai programar; e Estruturar o raciocínio de forma, a saber, combinar os comandos desta linguagem, dentro de uma sequência lógica, ou seja, de modo a atingirmos o objetivo perseguido. A linguagem de programação utilizada pelas nossas mentes é composta de comandos (olhe, escute, memorize, decida, sinte, ame, odeie...), elementos (cores, tipos de cheiros, notas musicais, símbolos sonoros, símbolos visuais...) e é capaz de reconhecer e utilizar os recursos disponíveis no organismo (olho, ouvido, cérebro, nariz, boca...), sem os quais se torna impossível à execução de um programa.

A linguagem de programação utilizada pelos computadores é composta também por comandos (receba do teclado, escreva no vídeo, some, multiplique...) e elementos (números, palavras e símbolos), e também é capaz de reconhecer e utilizar os recursos disponíveis no computador e periféricos (impressora, monitor de vídeo, mouse, teclado, memória...).

Ao aprendermos uma receita de bolo, colocamos uma porção de comandos em nossa mente. Estes comandos são gravados de uma forma analógica e, logo em seguida, consolidamos o processo de aprendizagem através da execução daquele programa. Claro, da primeira vez que fazemos algo sempre cometemos erros, que são automaticamente ajustados pela nossa mente para que, da próxima vez, não voltem a acontecer. Da mesma maneira, sempre que um programador desenvolve a primeira versão de seu programa, há uma fase de testes, quando os erros aparecem e são ajustados à necessidade do usuário. A esta fase damos o nome de depuração.

Como vemos, a programação de computadores é bastante semelhante à programação de nossas mentes, o que torna suas dificuldades e características também similares. Sendo assim, começaremos o nosso estudo das técnicas de programação estudando o comportamento dos programas que estão gravados em nossas mentes. Logo em seguida, aprenderemos a aplicar essas técnicas aos elementos e recursos inerentes a computadores, interagindo com variáveis de memória, entrada e saída de dados (teclado e monitor), vetores, matrizes e por fim: arquivos em disco magnético.

Mesmo assim, utilizaremos uma linguagem não comercial para aprendermos a manipular o computador. Uma linguagem que lhe poupe de comandos em inglês, e que não apresente grandes sofisticções. A linguagem de programação que utilizaremos para exercitar a lógica de programação será a nossa própria língua portuguesa é o Portugal ou Visualg.

3.1 Conceitos

Damos o nome de variável ao local de memória onde serão armazenados os dados utilizados nos algoritmos. Em nossos algoritmos não nos preocuparemos com o endereço real dessas variáveis, pois as maiorias das linguagens de programação tornam estes endereços transparentes ao programador.

Para exemplificarmos a diferença entre dados e variáveis, bem como entender melhor o endereçamento de variáveis, podemos citar o exemplo de uma estante de prateleiras, onde estão guardados livros ou quaisquer outros objetos. Os livros e objetos podem ser chamados de dados, conteúdo das variáveis. Para nos referenciarmos à variável é necessário darmos um nome à mesma, pois não trabalharemos com endereço de memória propriamente dito, e sim com apelidos (identificadores).

É aconselhável que o nome da variável seja expressivo, isto é, um nome que expresse o que vai ser armazenado dentro dela. As variáveis podem ainda ser simples ou compostas. As variáveis simples são aquelas que recebem um único dado por vez, enquanto que as compostas podem armazenar vários dados de uma só vez.

Para exemplificarmos melhor, imagine que em um programa você precise armazenar certa idade em algum lugar, para posteriormente decidir se esta idade permite que alguém possa requerer a sua carteira de motorista. Para tanto, tudo o que temos a fazer é declarar, em nosso programa, esta variável. Quando declaramos uma variável, temos que associar a ela algumas características:

- Nome ou Identificador;
- Tipo; e
- Tamanho.

Como dissemos anteriormente, o endereço físico da variável não precisa ser declarado, a menos que estejamos programando em alguma linguagem de baixo nível, como o Assembly. As linguagens de alto nível procuram espaços livres na memória do computador para associarem bytes à variável declarada. Algumas linguagens não exigem sequer a declaração do tamanho da variável.

Como já dissemos o nome a ser associado à variável não é importante para o computador, pois este servirá apenas como uma mera referência. Entretanto para outros programadores que possam vir a analisar os seus programas, ou até para você mesmo após algum tempo, é necessário que esses nomes sejam expressivos, simples e objetivos. Vejamos agora alguns exemplos de identificadores:

- SALARIO Um bom nome para variável que irá armazenar um valor salarial;
- CONT Um bom nome para variável que irá registrar uma contagem;
- TOTAL Um bom nome para variáveis acumuladoras de somas; e
- DATANASC Um bom nome para variáveis usadas para armazenar uma data de nascimento.

Devemos evitar nomes do tipo: X, K, C1, ABC, etc. a menos que eles expressem algo real. Nomes de variável, na maioria das linguagens, não devem: Iniciar por números: 1C2, 9ANOS, 100, 4CANTOS, etc.; e ser descontínuos: DATA NASC, FONE COMERC, etc. A maioria das linguagens apresenta certas restrições à livre escolha dos nomes das variáveis. A mais evidente é a impossibilidade da utilização de palavras reservadas da própria linguagem. Outras vezes, a linguagem traz consigo algumas variáveis reservadas. Trata-se de algumas variáveis que são automaticamente declaradas pela linguagem. Normalmente elas registram a data e a hora do sistema operacional, ou algumas informações internas como códigos de periféricos, endereços de memória para buffers, etc. O segundo argumento presente na declaração de uma variável é o seu tipo. O tipo de uma variável define que tipos de dados poderão ser armazenados nela.

3.1.1 Variáveis Numéricas

São variáveis que aceitam apenas dados numéricos, seja ele inteiro ou real, sem ou com casas decimais respectivamente, o que será definido junto com a sua declaração no início do algoritmo. Ex.: 1200; 10,9; -12,5; 0; etc.

3.1.2 Variáveis alfanuméricas ou Caractere

São variáveis que aceitam conteúdos de qualquer natureza, portanto: números, letras e símbolos em geral, não podendo estes números serem utilizados para operações aritméticas. Os conteúdos de variáveis do tipo caractere são delimitados por aspas. Ex.: "MARIA JOSE"; "CASA AMARELA"; "123"; "NOMES".

3.1.3 Variáveis Lógicas

Este tipo de variável não permite o registro de números nem de palavras. Uma variável lógica só armazena um entre dois valores: VERDADEIRO ou FALSO. Não importa como, exatamente, o computador registra esta informação na memória. Pode ser por bytes zerados e não zerados, pelas letras V ou F, ou por outro mecanismo qualquer. O que nos interessa de fato é que este tipo de variável não registra outro tipo de informação exceto V ou F. Para efeito de representação, diremos que o conteúdo de uma variável lógica será representado por um símbolo binário delimitado por dois pontos: Exemplo: .V. ou .F.

3.1.4 Tamanho das Variáveis

Quando declaramos uma variável não precisamos delimitar o byte inicial e final que esta irá ocupar na memória do computador, entretanto é imprescindível (dependendo da linguagem e do tipo da variável) informarmos o tamanho que ela deverá ter. A escolha deste tamanho será uma função da nossa necessidade. Por exemplo, se quisermos criar uma variável que irá armazenar um número de telefone, deveremos escolher um tamanho adequado para este fim.

Muitas vezes somos surpreendidos por escolhas infelizes de tamanho. Um número telefônico, por exemplo, apresenta em geral sete dígitos. A variável TELEFONE, portanto terá que ser declarada com tamanho de sete bytes. No entanto, existem números que exigem um código de DDD, ou mesmo discagem internacional. Sempre que nos deparamos com situações desta natureza, devemos escolher um tamanho que comporte o maior dado que aquela variável possa receber. Um exemplo clássico da escolha infeliz de tamanho é o formato de datas que hoje o mundo inteiro utiliza. Essas datas ocupam apenas oito dígitos, sendo os dois primeiros destinados ao dia, as duas centrais ao mês, e os dois últimos para o ano, ou melhor, para os dois últimos algarismos do ano.

Outros tipos de variáveis também são tratados por novas linguagens, tais como: HORA, MEMORANDO, OBJETO. Entretanto aqui só trataremos os tipos mencionados anteriormente, ou seja: NUMÉRICO, CARACTERE e LÓGICO. No tipo LÓGICO, o tamanho das variáveis é predefinido pelo computador. Neste caso este tamanho é sempre 1 (um).

Algumas linguagens não apresentam comandos de declaração de variáveis. Por exemplo, declara-se automaticamente uma variável a partir do momento em que o programador a menciona pela primeira vez no programa. Como nós estamos em busca de um aprendizado básico, iremos nos disciplinar a sempre declararmos todas as variáveis utilizadas no início de cada programa. Do contrário, o nosso compilador PORTUGOL recusará este procedimento, apontando um erro de sintaxe.

É importante aprender a forma sintética de se representar um comando. Nós utilizamos algumas nomenclaturas e simbologias que você, certamente, encontrará em qualquer livro sobre quaisquer linguagens. Assim, vamos elucidar algumas delas:

- < > Delimita o espaço reservado para inserirmos um parâmetro do comando. A maioria dos comandos de todas as linguagens apresenta parâmetros. Os parâmetros direcionam o objetivo de um comando específico.
- [] Delimita o espaço reservado para inserirmos um parâmetro opcional do comando, ou seja, tudo o que vier entre colchetes nem sempre precisará ser especificado no comando.
- DEFAULT. Default é uma palavra originada do inglês, porém em *computês* quer dizer: “tudo aquilo que é assumido na falta de especificações”.

Observe ainda que os colchetes [..] e os delimitadores maior/menor < > podem acumular-se entre si, ou seja, dentro de um grupo de parâmetros opcionais [], poderemos ter outros [[..]] Esperamos que a partir de agora você entenda mais rapidamente os comandos do Portugol.

Algumas linguagens não apresentam comandos de declaração de variáveis. Já outras, não amarram a nomenclatura da variável ao seu tipo, porém definem esta característica em função do primeiro conteúdo de cada variável no programa, podendo está mudar de tipo diversas vezes, dependendo dos valores a elas atribuídos.

Como nós estamos em busca de um aprendizado básico, iremos nos disciplinar a sempre declararmos todas as variáveis utilizadas no início de cada programa.

Do contrário, o nosso compilador Portugol recusará este procedimento, apontando um erro de sintaxe.

Agora que você já aprendeu o primeiro comando de nossa linguagem o VAR, é importante aprender a forma sintética de se representar um comando. Até agora, vimos conceitos e exemplos de variáveis. Agora vamos aprender a utilizá-las em algoritmos. Representaremos o comando de atribuição pela combinação de dois símbolos do teclado, que são <- (menor que e menos). Este símbolo representará a atribuição de um dado a uma variável. <ExpC1> <- <ExpC2> onde <ExpC1> - é o nome da variável e <ExpC2> é o dado a ser armazenado em <ExpC1> Exemplos:

- NOME <- "JOÃO" Dados CARACTERE tem que vir entre aspas;
- IDADE <- 25 Dados NUMERICO não precisam de aspas;
- ALTURA <- 1,67 Vírgula decimal (linguagens comerciais adotam o ponto);
- Atribuição a uma variável o valor de outra: ALTURA <- IDADE a variável ALTURA recebe o valor de IDADE; e
- Atribuir o resultado de uma expressão numérica a uma variável: SOMA <- QUOEF * (VALOR + 10) / DIVISOR.

Até aqui nós vimos como atribuir um valor a uma variável dentro de um programa. Mas este valor é atribuído pelo próprio programa, não é? Mas como fazer com que um programa, em um dado momento, receba um valor digitado pelo usuário? Como receber este dado através do teclado? Como exibir o resultado de uma operação qualquer no monitor de vídeo ou na impressora? São para isto que existem, em todas as linguagens de programação, os comandos de entrada e saída de dados. No caso do Portugol, estes comandos são os seguintes: Leia e Escreva ou Escreval. Exemplo ao lado:

```
VAR
  NOME: CARACTERE
  IDADE: INTEIRO
INICIO
  LEIA(NOME)
  LEIA(IDADE)
  ESCREVA("O nome é ", nome, " e a idade é ", idade)
FINALGORITMO
```

A execução do comando Leia provoca uma interrupção na execução do programa, fazendo com que o computador fique em estado de espera, aguardando a digitação de algo que deva ser atribuído às variáveis especificadas na sintaxe do comando. No exemplo acima, veja que o computador aguarda que o usuário digite dois valores distintos através do teclado. Estes dois valores irão ocupar, respectivamente, as variáveis, NOME e IDADE. O comando Escreva faz com que se imprima o conteúdo na posição do cursor. Usando-se Escreval o comando irá dar um salto de linha antes de imprimir a expressão especificada.

Como dissemos anteriormente, podemos atribuir a uma variável o resultado de uma expressão numérica. Entendemos por expressão numérica um conjunto de operações aritméticas combinadas entre operandos e operadores. Mas além das expressões numéricas, podemos escrever expressões dos tipos caractere e lógica.

3.1.5 Operações Numéricas

Uma expressão numérica é constituída: do lado esquerdo pela variável receptora; no centro pelo símbolo de atribuição que em Portugal é (<-); e na parte direita, por constantes, variáveis, e/ou funções numéricas, unidas pelos operadores aritméticos. Operadores:

- +, - Operadores unários, isto é, são aplicados a um único operando. São os operadores aritméticos de maior precedência. Exemplos: -3, +x. Enquanto o operador unário - inverte o sinal do seu operando, o operador + não altera o valor em nada o seu valor;
- \ Operador de divisão inteira. Por exemplo, $5 \setminus 2 = 2$. Tem a mesma precedência do operador de divisão tradicional;
- +, -, *, / Operadores aritméticos tradicionais de adição, subtração, multiplicação e divisão; Por convenção, * e / têm precedência sobre + e -. Para modificar a ordem de avaliação das operações, é necessário usar parênteses como em qualquer expressão aritmética;
- % Operador de módulo (isto é, resto da divisão inteira). Por exemplo, $8 \% 3 = 2$. Tem a mesma precedência do operador de divisão tradicional; e
- ^ Operador de potenciação. Por exemplo, $5 ^ 2 = 25$. Tem a maior precedência entre os operadores aritméticos binários (aqueles que têm dois operandos).

Uma mesma variável pode ser receptora e fazer parte de uma expressão ao mesmo tempo. Exemplo: $v1 <- 20 + v1$ ou $v2 <- v2 / 2$. Relembrando um pouco a matemática elementar, vamos rerepresentar a ordem de execução de uma expressão aritmética segundo as prioridades de operações vista ao lado:

- Parêntesis: ();
- Potenciação: ^;
- Multiplicação e Divisão: * /; e
- Soma e Subtração: + -.

3.1.6 Operações Caractere

Uma expressão caractere é constituída: na parte esquerda pela variável receptora do resultado obtido; no centro pelo símbolo de atribuição; na parte direita, por constantes, variáveis, e/ou funções numéricas, unidas pelos operadores caracteres, + indica concatenação:

- VAR
- V1 <- "Álvaro"
- V2 <- "Pinheiro"
- V3 <- V1 + " " + V2 // "Álvaro Pinheiro"

3.1.7 Operações Lógicas ou Relacionais

Uma expressão lógica é constituída: na parte esquerda pela variável receptora do resultado obtido; no centro pelo símbolo de atribuição; na parte direita, por constantes, variáveis, e/ou funções lógicas, unidas pelos operadores lógicos: > maior que; >= maior ou igual a; < menor que <= menor ou igual a; <> diferente de; = igual a. Exemplo ao lado:

- A > B Verdadeiro se A for maior que B;
- C = D Verdadeiro se C for exatamente igual a D; e
- NOME = "JOSE" verdadeiro se NOME= "JOSE".

Os operadores relacionais incidem sobre dados e variáveis numéricas e caracteres. Assim sendo, podemos comparar não apenas números, mas também palavras. Neste sentido, a comparação respeitará a ordem alfabética das iniciais de cada uma. Exemplo:

- "MARIA">"ANA" Retorna verdadeiro; e
- "MARIA"<"MARIO" Retorna verdadeiro.

Não há porque comparar variáveis lógicas, a não ser através de igualdade, pois qual é a necessidade de sabermos se verdadeiro é maior que falso? Operadores lógicos do tipo conectores E quando todas as condições forem VERDADEIRAS ou quando uma das condições for VERDADEIRA. Usamos NAO quando queremos inverter o resultado de uma condição ou expressão lógica (negar). Exemplo ao lado:

- X=90 E Z=100; e
- A>=67 OU A<=200.

Relembrando um pouco de álgebra booleana, vamos apresentar a ordem na qual as operações lógicas são executadas segundo suas prioridades:

- () parêntesis;
- NAO negação;
- E simultaneidade; e
- OU opcionalidade.

Exemplo: $A > B$ OU $C > D$ OU $E > F$ é o mesmo que: NÃO ($A \leq B$ E $C \leq D$ E $E \leq F$).

Dados dois números digitados via teclado, faça um algoritmo capaz de exibir a soma deles no vídeo.

```
VAR
X: INTEIRO
Y: INTEIRO
SOMA: INTEIRO
INICIO
  LEIA(X)
  LEIA(Y)
  SOMA <- X+Y
  ESCREVA(SOMA)
FINALGORITMO
```

Comentário na solução acima declarou três variáveis de memória. X e Y foram criadas para armazenar os dois números que seriam digitados pelo usuário, e SOMA serviu para armazenar o resultado final da operação com essas duas variáveis. Mas sempre existe mais de uma maneira de solucionarmos um problema, não é? Com algoritmos não poderia ser diferente. Tente repensar o algoritmo acima utilizando apenas duas variáveis, em vez de três! Como você pode observar, existem no mínimo duas soluções para este problema:

```
VAR
X: INTEIRO
Y: INTEIRO
SOMA: INTEIRO
INICIO
  LEIA(X)
  LEIA(Y)
  X <- X+Y
  ESCREVA(X)
FINALGORITMO
```

```
VAR
X: INTEIRO
Y: INTEIRO
Inicio
  LEIA(X)
  LEIA(Y)
```

```
ESCREVA(X+Y)
FIMALGORITMO
```

Qual a solução mais eficiente das duas? Somente o fato de economizar uma variável torna qualquer uma dessas soluções bem mais eficiente que a primeira. Mas claro que a terceira e última foi a mais eficiente de todas, uma vez que utilizou menos linhas de código para fazer a mesma coisa. Teste você mesmo essas soluções no Visualg e confira os resultados. Comece pela primeira solução e vá evoluindo até a última.

Sabemos que as linhas de um programa são executadas de cima para baixo de forma sequencial. Mas frequentemente precisamos mudar este curso natural de acordo com decisões a serem tomadas. Por exemplo, imagine se, no exemplo dado acima se quiséssemos que o resultado da soma de X e Y não fosse exibido caso fosse menor ou igual a zero! Neste caso, a única solução seria desviar a ação natural do programa caso esta condição fosse verdadeira, ou seja, teríamos que utilizar o comando condicional SE.

```
SE (<ExpL>) ENTAO
  <COMANDO-1>
  <COMANDO-2>
  <...>
[SENAO
  <COMANDO-1>
  <COMANDO-2>
  <...>]
FIMSE
```

ExpL - Expressão lógica qualquer. <COMANDO-1> São comandos quaisquer que serão executados caso o resultado da expressão lógica <ExpL> seja verdadeira. Esses comandos devem ser escritos entre as cláusulas ENTAO e SENAO. <COMANDO-2> São comandos quaisquer que serão executados caso o resultado da expressão lógica <ExpL> seja falsa. Esses comandos devem ser escritos entre as cláusulas SENAO e FIMSE.

Em tempo de execução, se a expressão lógica <ExpL> retornar verdadeiro então os comandos <comandos 1> serão executados. Do contrário, <comandos 2> são os que serão executados. O comando SE, portanto, desvia a sequência de execução do programa para os comandos do ENTAO ou do SENAO, dependendo sempre de uma condição: <ExpL> De uma maneira ou de outra, após a execução de <comandos 1> ou de <comandos 2>, o programa seguirá sua sequência natural a partir do FIMSE. Para fixar esta ideia, tente depurar o seguinte algoritmo:

```
VAR
  X: INTEIRO
```

```
Y: INTEIRO
SOMA: INTEIRO
INICIO
  LEIA(X)
  LEIA(Y)
  SOMA <- X+Y
  SE SOMA <= 0 ENTAO
    ESCREVA("Resultado não interessa")
  SENAO
    ESCREVA (SOMA)
  FIMSE
  ESCREVA ("Fim transação")
FINALGORITMO
```

Caso a soma dos dois números digitados seja igual ou inferior à zero, então uma mensagem seria exibida: "Resultado não interessa". Do contrário, a soma seria normalmente exibida, como antes. De uma forma ou de outra, a mensagem "Fim da transação" será impressa. Quando dissemos que <comandos 1> e <comandos 2> seriam executados alternativamente de acordo com o resultado da expressão <ExpL>, não excluíamos outros comandos condicionais subordinados àquelas condições, ou seja, podemos utilizar quantos SE quisermos dentro de outros.

Para exemplificarmos melhor, imagine se quiséssemos um maior desdobramento de condições do último algoritmo, por exemplo: se a soma de X e Y for menor ou igual à zero, detectar se ela é negativa. Se for, exibir a mensagem "Resultado incompatível" em vez de "Resultado não interessa", e de qualquer maneira, nesta situação, exibir os dois números envolvidos na soma, um ao lado do outro.

```
VAR
X: INTEIRO
Y: INTEIRO
SOMA: INTEIRO
INICIO
  LEIA(X, Y)
  SOMA<-X+Y
  SE SOMA<=0 ENTAO
    SE SOMA<0 ENTAO
      ESCREVA "Resultado incompatível"
    SENAO
      ESCREVA "Resultado não interessa"
  FIMSE
  ESCREVA(X)
  ESCREVA(Y)
  SENAO
```

```
    ESCRIVA(SOMA)
    FIMSE
    ESCRIVA("Fim da transação")
FIMALGORITMO
```

Sobre esta solução, é importante salientar o seguinte: Note que a estética, apesar de não influir no resultado da execução do programa, é de extrema importância para o bom entendimento do mesmo. Perceba que todos os comandos subordinados ao primeiro SE ficam descolados na mesma margem. Apenas a cláusula SENAO fica na mesma margem do SE. O mesmo ocorre com o segundo SE e com todos os que vierem.

Veja que os comandos: ESCRIVA(X) e ESCRIVA(Y) são executados independentes da condição do segundo SE. É por isto que eles foram escritos após o FIMSE do segundo SE. Algumas vezes, sentimos necessidade de escrevermos repetidamente o mesmo comando. Em alguns casos, a quantidade de vezes que temos de escrever é indeterminada ou depende de uma condição. Por exemplo, digamos que temos de fazer o computador escrever cinco números inteiros, que serão informados pelo usuário, a partir de zero. Se escrito com os comandos aprendidos até agora, o algoritmo será longo. Se utilizarmos uma estrutura de repetição, será bem mais simples.

O comando Enquanto faça prende um trecho do programa a um laço (*looping*), ou seja, podemos determinar que certo conjunto de comandos fosse executado repetidamente enquanto uma condição seja verdadeira. A sintaxe do comando é a seguinte:

```
ENQUANTO (<condição>) FAÇA
    <COMANDO-1>
    <COMANDO-2>
    <...>
```

FIMENQUANTO

Enquanto a expressão lógica <condição> retornar verdadeiro os <comandos> situados entre o enquanto e o FIMENQUANTO será executado ininterruptamente. Antes de executar cada ciclo do bloco de comandos envolvidos na estrutura de repetição, o computador checka a veracidade da condição descrita em <condição> Quando esta for falsa, os comandos compreendidos naquela faixa não mais serão executados, passando o programa a executar o próximo comando imediatamente após o FIMENQUANTO.

3.2 Atividades

1) Elabore um algoritmo capaz de receber um número e, em seguida, listar todos os números de um até ele.

- 2) Sofisticando ainda mais o programa anterior, vamos fazer com que, além de listar todos os números de um até um determinado número digitado via teclado, ele mostre a soma de todos esses números no final da listagem.
- 3) Receba uma sequência de nomes via teclado até que seja digitado um nome em branco (= " "). Após a digitação desta lista, escrever no vídeo quantos nomes foi digitado.
- 4) Desenvolva um algoritmo capaz de receber uma frase com oitenta dígitos e exibir na tela do computador quantas letras "B" foi digitado nesta frase.
- 5) Elabore um algoritmo que receba um número e escreva a soma de todos os números ímpares até ele.
- 6) Elabore um algoritmo capaz de exibir a média aritmética de todos os números de um até certo número digitado previamente via teclado.
- 7) Desenvolva um programa que receba uma sequência de números e escreva a soma de todos eles. A sequência de números deverá se encerrar quando o número zero for digitado. Este último número não entrará no cálculo.
- 8) Desenvolva um programa que receba uma sequência de números e escreva a soma, a média e a quantidade de números digitados. A sequência de números deverá se encerrar quando o número zero for digitado. Este último número não entrará no cálculo.
- 9) Faça um algoritmo capaz de receber uma sequência de salários, até que seja digitado um salário igual à zero. No final da sequência, o computador deverá mostrar quantos salários são superiores a R\$1.500,00, quantos são inferiores a R\$150,00, e quantos se situam na faixa intermediária.
- 10) Elabore um algoritmo capaz de receber uma sequência de números e escrever, ao final, o maior número digitado. O número zero, quando digitado, deverá encerrar a sequência.
- 11) Segue lista de atividades propostas. Em < <http://goo.gl/vMo4RF> >

3.3 Ferramentas

Ferramenta de Pseudocódigo. Em <

<http://www.4shared.com/folder/10qAyQSK/UTIL.html> >

G-Portugol. Em < <http://sourceforge.net/projects/gpt.berlios/> >

Portugol Studio. Em < <http://sourceforge.net/projects/portugolstudio/> >

Visualg. Em < <http://goo.gl/HtnWkc> >

3.4 Sugestões de Leitura

Curso Essencial de Lógica de Programação. Em < <http://goo.gl/CbksbB> >

Lógica de Programação com Visualg. Em < <http://goo.gl/dMFH8V> >

Lógica de Programação. Em < <http://goo.gl/TuRR9d> >

Treinamento em Lógica de Programação. Em < <http://goo.gl/CQHgnH> >

3.5 Referências Web

Conceitos de Lógica de Programação. Em < <http://goo.gl/vUtW4U> >

Guia Rápido de Visualg. Em < <http://goo.gl/zttvBD> >

Introdução a Lógica de Programação. Em < <http://goo.gl/Y3bUfG> >

Introdução a Programação. Em < <http://goo.gl/V611q7> >

Manual Básico de Visualg. Em < <http://goo.gl/FIQofB> >

Raciocínio Lógico. Em < <http://goo.gl/Sfl72d> >

Lógica de Programação (wiki). Em < <http://goo.gl/ZofbE> >

3.6 Referências Bibliográficas

DEITEL, Harvey; DEITEL, Paul. Java: Como Programar. 6ªed. US: Pearson, 2005.

FARRER, Harry. Algoritmos Estruturados. 3ªed. Rio de Janeiro: LTC, 1988.

FARRER, Harry. Programação estruturada de computação: Pascal estruturado. 1ªed. Rio de Janeiro: LTC, 1986.

4 Referências Bibliográficas

AHO, Alfred; *et al.* Compilers: Principles, Techniques, and Tools. 2ªed. US: Pearson, 2006.

BEDREGAL, B.; BENEDITO, A. Introdução à Teoria das Ling. Formais dos Autômatos e da C. RN: Edunp, 2010.

DIVERIO, Tiarajú; *et al.* Teoria da Computação: Máquinas Universais e Comp. 3ªed. Porto Alegre: Bookman, 2011.

FARRER, Harry. Algoritmos Estruturados. 3ªed. Rio de Janeiro: LTC, 1988.

FARRER, Harry. Programação estruturada de computação: Pascal estruturado. 1ªed. Rio de Janeiro: LTC, 1986.

HOPCROFT, John; *et al.* Introdução à Teoria de Autômatos, Linguagens e Computação. 1ªed. RJ: Campus, 2002.

LOUDEN, Kenneth. Compiladores: Princípios e Práticas. 1ªed. Pioneira Thomson Learning, 2004.

MENEZES, Paulo. Linguagens Formais e Autômatos. 6ªed. Porto Alegre: Bookman, 2011.

SUDKAMP, Thomas. Languages and Machines: An Introduction to the Theory of Computer Science, 1997.

TUCKER, Allen; NOONAN, Robert. Linguagens de Programação: Princípios e Paradigmas. 2ªed. McGraw, 2009.

5 Referências Web

Autômatos Finitos Determinísticos wiki. Em < <http://goo.gl/3IJJxV> >
Autômatos Finitos Determinísticos. Em < <http://goo.gl/NZHyRP> >
Autômatos Finitos e Expressões Regulares. Em < <http://goo.gl/3Ox2WM> >
BNF for Java. Em < <http://cui.unige.ch/isi/bnf/JAVA/BNFindex.html> >
Conceitos de Lógica de Programação. Em < <http://goo.gl/vUtW4U> >
Converter Autômatos Finitos em Expressão Regular wiki. Em < <http://goo.gl/Bd2kcz> >
Expressão Regular wiki. Em < <http://goo.gl/wNRox> >
Gramática BNF para SQL ANSI. Em < <http://savage.net.au/SQL/> >
Guia de Programação. Em < <http://goo.gl/NL4sX9> >
Guia Rápido de Visualg. Em < <http://goo.gl/zttvBD> >
Introdução a Lógica de Programação. Em < <http://goo.gl/Y3bUfG> >
Introdução a Programação. Em < <http://goo.gl/V611q7> >
Introdução a Teoria da Computação Fascículo 1. Em < <http://goo.gl/4Q6ZHe> >
Introdução a Teoria da Computação Fascículo 2. Em < <http://goo.gl/9bhlns> >
Introdução a Teoria da Computação Fascículo 3. Em < <http://goo.gl/vQsKpg> >
Langpop. Em < <http://www.langpop.com/> >
Ling. Formais e Autômatos. Em < <http://www.deamo.prof.ufu.br/CursoLFA.html> >
Lógica de Programação (wiki). Em < <http://goo.gl/ZofbE> >
Ohloh. Em < <http://www.ohloh.net/> >
Paradigma (wiki). Em < <http://pt.wikipedia.org/wiki/Paradigma> >
Paradigma de Desenvolvimento (wiki). Em < <http://goo.gl/rIYFls> >
Paradigma. Em < <http://pt.wikipedia.org/wiki/Paradigma> >
Raciocínio Lógico. Em < <http://goo.gl/Sfl72d> >
Tiobe. Em < <http://www.tiobe.com/index.php/content/company/Home.html> >

6 Ferramentas

Ferramenta de Pseudocódigo. Em < <http://www.4shared.com/folder/1oqAyQSK/UTIL.html> >
G-Portugol. Em < <http://sourceforge.net/projects/gpt.berlios/> >
Portugol Studio. Em < <http://sourceforge.net/projects/portugolstudio/> >
Regex. Em < <http://alessandrosantos.com.br/lab/er/> >

7 Sugestões de Leitura

Compiladores. Em < <http://goo.gl/KRL19c> >

Lógica de Programação com Visualg. Em < <http://goo.gl/dMFH8V> >

Lógica de Programação. Em < <http://goo.gl/TuRR9d> >

Programação de Computadores. Em < <http://goo.gl/uldtPN> >

Treinamento em Lógica de Programação. Em < <http://goo.gl/CQHGnH> >

Livros da série Fundamentos da Engenharia de Software



Fundamentos da Engenharia de Software: Conceitos Básicos é uma coletânea de

disciplinas que integradas servem para fundamentar o entendimento da construção de projetos de software com qualidade, isto é, baseado em processos maduros e reconhecidos pela comunidade tecnológica. O objetivo deste livro é fornecer ao leitor as bases necessárias para o desenvolvimento de aplicações sejam Desktop, Web ou Mobile. Iniciando a leitura na Teoria da Computação, passando por Processos, Linguagens, Bancos de Dados e finalizando com Sistemas de Informação e Colaboração. Este livro pode ser lido capítulo a capítulo ou somente a disciplina desejada, pois sua elaboração consiste na compilação das disciplinas fundamentais da Engenharia de Software que são independentes, mas ao mesmo tempo se integram objetivando o desenvolvimento de aplicações.



Introdução à Banco de Dados. Neste são abordados os conceitos básicos de bancos de dados e seus

sistemas gerenciadores, mas com o foco na arquitetura relacional, porque ainda hoje o mercado faz uso em larga escala desses bancos de dados, mesmo que o paradigma predominante seja o orientado a objetos e que, já existam a um bom tempo bancos orientados a objeto, até mesmo os bancos objetos-relacionais que são um híbrido entre essas duas arquiteturas, o que predomina ainda é o relacional, assim, este material é focado na linguagem de consulta estruturada para os SGBD-Rs do mercado, com foco na comparação de cinco dos mais utilizados bancos relacionais, os quais são: Oracle, SQLServer, MySQL, SQLBase e Interbase.



Este livro é sobre processos de desenvolvimento de software, evidenciando a necessidade

de qualidade na construção de sistemas, conceituando a diferença entre desenvolvimento Adhoc e com processo. Para isso é realizado a introdução à engenharia de requisitos abordando as técnicas para a elicitación de requisitos que forneçam subsídios necessários para uma construção de software com maior qualidade, enfatizando a necessidade de se aplicar na construção de qualquer sistema as técnicas de análise e modelagem, evidenciando o uso da linguagem da Linguagem de Modelagem Unificada (UML) para diagramar um projeto de software, explicando a necessidade do uso de modelos na construção, entrando com detalhes na análise orientada a objetos, com o objetivo de explorar os seus conceitos de requisitos e modelagem integrados. Este material é finalizado com a introdução à medidas de esforço de desenvolvimento, técnica necessária para responder as perguntas básicas de qualquer desenvolvimento: Qual o prazo e custo? E para responder a essas questões é abordado o uso da métrica análise de ponto de função.



Este livro aborda os sistemas que são classificados como informação, a exemplo, sistemas de apoio a

decisão, sistemas estratégicos, sistemas gerenciais e sistemas transacionais. A produção deste material que compõe o volume 4 da coleção Fundamentos da Engenharia de Software é resultado da compilação das aulas produzidas nas disciplinas que compõem os capítulos deste livro.



A motivação deste livro é exemplificar os conceitos de Padrões de Projetos utilizando a linguagem de programação

o Java, sendo a construção uma compilação das aulas produzidas com o intuito de facilitar o entendimento do assunto abordando os seguintes temas: Paradigma Orientado a Objetos que introduz o leitor nos conceitos do POO; Linguagem de Modelagem Unificada para apresentar a simbologia UML dos conceitos de POO; Linguagem de Programação Java apresentando essa poderosa linguagem de programação orientada a objetos para exemplificar os padrões de projeto; e Padrões de Projetos que neste livro aborda os mais referenciados nas academias, sendo eles o GRASP e GoF.



Este livro é o resultado do uso da ferramenta MS Project da Microsoft utilizada na aplicação dos conceitos de

gestão de projetos do PMBOK com as premissas da engenharia de testes para aquisição de qualidade nos produtos de software.



Este livro aborda basicamente e os conceitos básicos de programação o como autômatos, tipos de

linguagens, princípios dos compiladores, paradigmas de desenvolvimento e lógica de programação.



Este livro introduz nas tecnologias Web abordando os conceitos básicos para desenvolviment

to para Internet com a apresentação da plataforma Dot Net e exibindo dicas de codificação para a linguagem de marcação ASPX, para a linguagem de script mais utilizada pelos navegadores o JavaScript com exemplos de CSS e principalmente dicas de código para a linguagem de programação CSharp e de banco de dados SQL com foco

no SQLServer.