FACULDADE DE TECNOLOGIA DE SÃO PAULO

LEANDRO FERNANDES VIEIRA

PARADIGMAS DE PROGRAMAÇÃO: UMA ABORDAGEM COMPARATIVA

SÃO PAULO

2015

FACULDADE DE TECNOLOGIA DE SÃO PAULO

LEANDRO FERNANDES VIEIRA

PARADIGMAS DE PROGRAMAÇÃO: UMA ABORDAGEM COMPARATIVA

Trabalho submetido como exigência parcial

para obtenção do Grau de Tecnólogo em

Análise e Desenvolvimento de Sistemas

Orientador: Prof. Dr. Silvio do Lago Pereira

SÃO PAULO

2015

FACULDADE DE TECNOLOGIA DE SÃO PAULO

LEANDRO FERNANDES VIEIRA

PARADIGMAS DE PROGRAMAÇÃO: UMA ABORDAGEM COMPARATIVA

Trabalho submetido como exigência parcial para a obtenção do Grau de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Parecer do Professor Orientador:_____

_____

_____

Conceito/Nota Final: _____

Orientador: Prof. Dr. Silvio do Lago Pereira

SÃO PAULO, ____ de DEZEMBRO de 2015.

A meus pais, seres iluminados que sempre colocaram todo o seu amor, dedicação e esforços na minha educação e formação como pessoa.

AGRADECIMENTOS

Agradeço a minha família por todo o apoio e compreensão ao longo da minha vida, sempre me transmitindo sabedoria e bons valores, o que com certeza me tornou uma pessoa melhor.

À minha namorada Natalia por todo amor e os ótimos momentos que vivemos juntos e por todos os sonhos que ainda iremos realizar.

Ao meu orientador Prof. Dr. Silvio do Lago Pereira pela confiança no meu trabalho, e por toda a ajuda na elaboração do mesmo.

Por fim, a todas as pessoas que contribuíram de forma direta ou indireta para que eu me tornasse uma pessoa melhor e para a realização deste trabalho.

RESUMO

Programação é um amplo campo dentro da ciência da computação, por meio desta podemos desenvolver sistemas em diversas áreas, tais como mídias sociais, robótica, espacial, cientifica e educacional. Um paradigma de programação defini um estilo de programação, determinando a visão que o programador possui sobre a estruturação e execução do programa, permitindo ou proibindo a utilização de algumas técnicas de programação. Este trabalho tem como objetivo apresentar os conceitos e técnicas dos principais paradigmas de programação: procedimental, orientado a objeto, funcional e lógico, bem como estabelecer uma análise comparativa entre os mesmos.

Palavras-chave: Programação, Paradigmas de Programação, Programação Procedimental, Orientação a Objeto, Programação Funcional, Programação Lógica.

ABSTRACT

Programming is a vast field within computer science, through this we can develop systems in several areas, such as social media, robotics, space, science and education. A programming paradigm defines a programming style, determining the view that the programmer has on the structuring and implementation of the program, allowing or prohibiting the use of certain programming techniques. This paper aims to present the concepts and techniques of the main programming paradigms: procedural, object-oriented, functional, logical, and establish a comparative analysis between them.

Keywords: Programming, Programming Paradigms, Procedural Programming, Object-oriented Programming, Functional Programming, Logic Programming.

SUMÁRIO

1. Introdução

1.1 Motivação

Atualmente, programação é uma atividade ubíqua na sociedade. Diariamente, em diversas áreas como, por exemplo, negócios, educação e saúde, sistemas informatizados, com necessidades e dificuldades distintas, são desenvolvidos.

Também é fato que o desenvolvimento de diferentes tipos de sistemas exige diferentes abordagens de programação, isto é, paradigmas de programação; pois não existe uma única abordagem que seja adequada para todo tipo de sistema. De fato, um paradigma de programação define um estilo de programação, ou seja, um conjunto de princípios, conceitos e técnicas, que visa tornar mais natural e eficiente o desenvolvimento de determinados tipos de sistemas.

Assim, conhecendo uma variedade de paradigmas de programação, o profissional da área de desenvolvimento de sistemas estará mais preparado para decidir qual abordagem é mais apropriada para o desenvolvimento de um projeto particular, reduzindo desta forma a quantidade de esforço necessário durante a programação.

Outro benefício de se conhecer diversos paradigmas de programação é que um programador que tem esse conhecimento, em geral, aprende a trabalhar com novas linguagens de programação mais fácil e rapidamente. O que é extremamente desejável na área de desenvolvimento de sistemas, uma vez que as linguagens de programação estão em constante evolução.

1.2 Paradigmas de Programação

A palavra paradigma tem origem na palavra grega παράδειγμα, que significa “padrão” ou “exemplo”, e que, por sua vez, vem de παραδεικνύναι, que significa “demonstrar”. A partir dessa observação, Thomas S. Kuhn definiu paradigma como "toda a constelação de crenças, valores, técnicas etc., partilhadas pelos membros de uma comunidade determinada" (KUHN, 1991). Assim, pode-se dizer que um paradigma expressa um padrão de pensamento de uma comunidade.

Uma vez que existem diferentes comunidades de programadores, bem como diferentes domínios de aplicação, é natural que haja diferentes paradigmas também

dentro da área de programação. Esses paradigmas de programação proporcionam aos programadores formas mais naturais de expressar suas idéias e pensamentos ao computador, utilizando estruturas de dados, técnicas e abstrações que fazem mais sentido dentro de sua comunidade, tipo de negócio ou contexto.

O uso de um paradigma de programação adequado ao domínio de aplicação permite que o programador desenvolva sistemas com maior produtividade, pois os códigos podem ser escritos de uma forma mais natural, mais legível e mais fácil de se manter ao longo da vida útil do sistema.

De forma geral, um paradigma de programação fornece e determina a visão que o programador possui sobre a estruturação e execução do programa, permitindo ou proibindo a utilização de algumas técnicas de programação.

Por exemplo, alguns dos paradigmas mais conhecidos (e algumas linguagens com suporte aos mesmos) são apresentados na Tabela 1:

Paradigma	Linguagem	Ano de Criação
Procedimental	COBOL	1959
	Pascal	1970
	C	1972
Orientado a Objetos	Simula	1967
	Smalltalk	1972
	C++	1979
	Java	1995
	C#	2001
Funcional	LISP	1958
	Erlang	1986
	Haskell	1990
	F#	2005
Lógico	Planner	1969
	Prolog	1972
	Oz	1995
Declarativo	SQL	1974
	Prolog	1972
	Mercury	1995

Tabela 1 – Principais paradigmas de programação e algumas linguagens com suporte aos mesmos

Há ainda linguagens híbridas (ou, multiparadigma), ou seja, linguagens que dão suporte total ou parcial a mais de um paradigma de programação.

A ideia de uma linguagem híbrida é motivada pelo fato de que nenhum paradigma isoladamente pode resolver todos os problemas da maneira mais elegante ou eficiente possível. Sendo assim, é necessário dar ao programador a liberdade de misturar princípios, conceitos e técnicas de diferentes paradigmas, ao resolver um problema particular.

Dessa forma, um sistema pode ser desenvolvido utilizando vários paradigmas ao mesmo tempo, com cada trecho de código obtendo apenas as vantagens do paradigma no qual foi escrito, levando assim o sistema a ter uma melhor desempenho e confiabilidade.

Alguns exemplos de linguagens híbridas são apresentados na Tabela 2.

Paradigma	Linguagem	Ano de Criação
Orientado a Objetos, Funcional	CLOS	1958
Declarativo, Lógico	Prolog	1972
Orientado a Objetos, Funcional, Procedimental	Python	1991
Orientado a Objetos (baseado em protótipos), Funcional, Procedimental	JavaScript	1995
Lógico, Orientado a Objetos, Funcional, Procedimental	Oz	1995
Orientado a Objetos, Funcional, Procedimental	Ruby	1995

Tabela 2 – Principais Linguagens híbridas e os paradigmas que suportam

1.3 Objetivos

O principal objetivo deste trabalho é introduzir os principais paradigmas de programação, descrevendo seus princípios, conceitos e técnicas, bem como apresentar uma análise comparativa entre eles. Com essa comparação, espera-se que o um programador possa perceber as vantagens e desvantagens de cada paradigma, em função do tipo de problema que se pretende resolver com as diversas linguagens de programação.

1.4 Organização

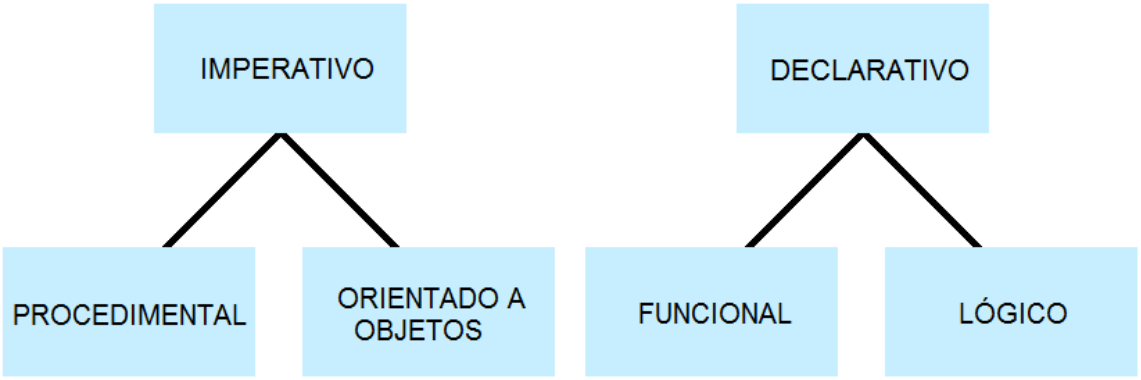
O restante deste trabalho está organizado da seguinte maneira:

- Capítulo 2 – Paradigmas Imperativo e Declarativo: apresenta os princípios nos quais estão baseados os paradigmas imperativo e declarativo;
- Capítulo 3 – Paradigma Estrutural e Procedimental: introduz os fundamentos, estruturas e técnicas utilizadas na programação estruturada e procedimental;
- Capítulo 4 – Paradigma Orientado a Objetos: introduz os conceitos fundamentais da programação orientada a objetos, apresentando exemplos de código detalhados para cada conceito;
- Capítulo 5 – Paradigma Funcional: introduz os conceitos fundamentais da programação funcional, apresentando exemplos de código detalhados para cada conceito;
- Capítulo 6 – Paradigma Lógico: introduz os conceitos fundamentais da programação lógica, apresentando exemplos de código detalhados para cada conceito;
- Capítulo 7 – Análise Comparativa: apresenta comparações entre os diversos paradigmas (visando ressaltar a expressividade, confiabilidade, eficiência, reusabilidade e manutenabilidade de cada um deles) e indica domínios em que cada paradigma seja possivelmente a melhor abordagem de programação a ser escolhida;
- Capítulo 8 – Conclusão: apresenta as conclusões finais desse trabalho.

2 Paradigmas Imperativo e Declarativo

Há dois paradigmas de programação de computadores de quais todos os outros paradigmas derivam, como mostra a Figura I: o paradigma imperativo e o paradigma declarativo.

Figura 1 – Relacionamento entre os principais paradigmas de programação.



Fonte: Elaborada pelo autor.

Dentre todos os paradigmas de programação, o paradigma imperativo é mais antigo, inicialmente implementado diretamente em linguagem de máquina e Assembly. Apesar disso, ele ainda é o paradigma mais usado na atualidade, uma vez que os paradigmas procedimental e orientado a objetos, entre outros, são derivados dele.

2.1 Paradigma Imperativo

O fundamento para o paradigma imperativo é o conceito da Máquina de Turing (TUCKER e NOONAN, 2007), que nada mais é que uma abstração matemática que corresponde ao conjunto de funções computáveis. A máquina de Turing foi aproximada por John von Neumann a uma arquitetura de computadores que fundamenta os computadores construídos até hoje (GARANHANI, 2008).

A essência da programação imperativa se resume a três conceitos (TUCKER e NOONAN, 2007):

- estados de uma máquina abstrata descritos por valores de um conjunto de variáveis;
- reconhecedores de estados - expressões compostas por relações entre esses valores ou os resultados de operações utilizando valores;

- comandos de atribuição e controle que manipulam o estado.

Em outras palavras, o programa é representado por um conjunto de dados que representam o estado do mesmo, e através da manipulação desses dados (leitura, criação de expressões e geração de novos valores) o estado do programa é alterado até que a solução do problema seja obtida. Ou seja, programas imperativos são sequências de comandos para o computador executar. Por esse motivo, é dito que o paradigma imperativo foca em “como” resolver um problema.

Por exemplo, o programa a seguir, em linguagem C (SCOTT, 2009), que é uma linguagem imperativa, resolve o problema de obter todos os números binários de três dígitos. Note que esse programa especifica “como” computar esse resultado, passo a passo. Para cada estado (representado pelos valores das variáveis `A`, `B` e `C`), o comando `printf()` mostra um número binário distinto.

Exemplo 1:

```
#include <stdio.h>

int main()
{
   for(int A=0; A<2; A++)
      for(int B=0; B<2; B++)
         for(int C=0; C<2; C++)
            printf("%d%d%d\n",A,B,C);
   return 0;
}
```

2.2 Paradigma Declarativo

A ideia do paradigma declarativo (GABBRIELLI e MARTINI, 2006), por sua vez, é que um programa pode ser visto como um teorema em uma lógica apropriada. Por essa linha de raciocínio, a programação é levada a um nível mais alto de abstração, onde o programador pode concentrar-se em descrever o resultado o que deve ser computado, e não necessariamente como esse resultado deve ser computado. Dessa forma, a programação declarativa define "o que" é para ser computado, mais do que "como" computar, entrando assim em contraste a programação imperativa.

Por exemplo, o programa a seguir, em linguagem Prolog (SCOTT, 2009), que é uma linguagem declarativa, resolve o problema de obter todos os números binários de três dígitos. Note que esse programa especifica “o que” deve ser computado como resultado. O programa declara os dígitos que podem ser usados na composição de um número binário, e define um binário de três dígitos como uma tupla `(A,B,C)`, em que cada componente representa um dígito binário. Para ver o resultado computado pelo programa, o usuário deve fazer a consulta: `?- binário(N).`

Exemplo 2:

```
dígito(0).
dígito(1).
binário(N) :- N=(A,B,C), dígito(A), dígito(B), dígito(C).
```

Na programação declarativa, não há a ideia de estado do programa, como ocorre na programação imperativa. Nesta primeira há apenas declarações de verdades que, após serem declaradas, são imutáveis. Por consequência, a programação se torna sem efeitos colaterais, isto é, dadas algumas declarações e realizada alguma interação entre elas, o resultado será sempre o mesmo para aquelas declarações, enquanto que, na programação imperativa, um mesmo trecho de código nem sempre retornará o mesmo resultado (pois o resultado depende do estado em que ele é computado).

Como a programação declarativa é baseada em dados imutáveis, ela expressa a lógica de uma computação sem descrever exatamente seu fluxo de controle. Dentro do paradigma declarativo, podemos identificar outros paradigmas, como o funcional (SCOTT, 2009), o lógico (SCOTT, 2009) e a programação restritiva (SCOTT, 2009).

3 Paradigma Procedimental

Como vimos o paradigma imperativo foca em definir “como” o programa deve ser computado, e fazem parte deste os paradigmas procedimental e orientado a objetos. Contudo, parte da forma em como explicitamos o fluxo de controle do programa nestes dois paradigmas foi fortemente influenciada pelo paradigma estrutural, sobre o qual discutiremos na Seção 3.1.

3.1 Paradigma Estrutural

Nos primórdios da computação, quando o paradigma imperativo era o único conhecido e se programava apenas em linguagens puramente imperativas, tais como Assembly (GABBRIELLI e MARTINI, 2006), FORTRAN (GABBRIELLI e MARTINI, 2006) e COBOL (SEBESTA, 2006), havia um recurso implementado pelo comando `goto`, que permitia saltos transferir a execução para qualquer linha específica dos programas. Fazendo uso deste recurso, era possível quebrar a linearidade do código, ou seja, a execução das linhas de código na ordem que foram escritas, o que tornava o software incrivelmente difícil de manter e compreender. Este estilo de programação era conhecido como código espaguete, uma vez que toda estrutura estava toda entrelaçada.

Em 1968 e 1987, Edsger W. Dijkstra (DIJKSTRA, 1968) e Niklaus Wirth (WIRTH, 1987) respectivamente, publicaram artigos nos quais defendiam a abolição do `goto` nas linguagens de programação de alto nível, alegando que o comando induzia a vários erros de programação e promovia práticas ruins, além de tornar extremamente difícil a leitura de programas.

Este movimento deu origem à programação estruturada, que se fundamenta em três estruturas de controle de fluxo de código (MILLS, 1972):

- Estrutura de sequência: na qual um comando é executado após o outro, de forma linear, e, portanto sem o uso de `goto`;
- Estrutura de decisão: na qual trechos do programa podem ser executados dependendo do resultado de um teste lógico;
- Estrutura de iteração: na qual determinado trecho de código é repetido por um número finito de vezes, enquanto um teste lógico for verdadeiro.

Os exemplos a seguir evidenciam a melhora significativa no fluxo de controle proporcionada pelo paradigma estrutural.

O Exemplo 3 mostra um programa (em pseudocódigo) que lê dois números e imprime o maior entre eles (execução condicional), usando `goto`. O Exemplo 4 mostra um programa estruturado correspondente, sem o uso de `goto`.

Exemplo 3:

```
begin:
    read a
    read b
    if b > a goto impb
    print a
    goto end
impb:
    print b
end:
```

Exemplo 4:

```
read a
read b
if b > a
    print b
else
    print a
```

O Exemplo 5 mostra um programa (em pseudocódigo) que exibe os números de 1 a 10 (execução iterativa), usando `goto`. O Exemplo 6 mostra um programa estruturado correspondente, sem o uso de `goto`.

Exemplo 5:

```
begin:
   x = 1
while:
    if x > 10 goto end
    print x
    x = x + 1
    goto while
end:
```

Exemplo 6:

```
x = 1
while x <= 10
    print x
    x = x + 1
end
```

3.2 Paradigma Procedimental

Apesar dos benefícios relativos ao aumento de legibilidade e manutenção de código proporcionada pela programação estruturada, esta ainda apresenta desvantagens em outros aspectos. Para programas extensos, que se dividem em várias tarefas complexas, por exemplo, é difícil saber quando uma tarefa acaba e outra começa, uma vez que todo o programa está em um código único. Outro ponto fraco é que o paradigma não proporciona reutilização de código. Quando uma tarefa precisa ser executada várias vezes ao longo de um mesmo programa, geralmente, a solução é reescrever a tarefa nos pontos necessários, o que, além de aumentar consideravelmente o tamanho do código, dificulta o entendimento e a manutenção do mesmo.

Como forma de solucionar esses problemas, propôs-se o paradigma procedimental, que é uma derivação do paradigma estruturado. Este novo paradigma agrega o conceito de procedimento (em inglês *procedure*, termo que dá nome ao paradigma).

Procedimento, também conhecido como rotina, subrotina, ou função, é uma unidade do programa que armazena uma sequência de instruções modularizadas a fim de executar uma tarefa específica. Esta unidade pode então ser chamada em qualquer parte do programa (e.g., ser chamada por outros procedimentos e por si própria) onde aquela determinada tarefa precise ser realizada. Desta forma, o uso de procedimentos possibilita abstrair tarefas específicas, permitindo aos programadores se preocupar em o que será feito, em vez de como.

Considerando que um programa de computador pode ser visto como um conjunto de tarefas, qualquer tarefa complexa pode ser dividida em um conjunto de tarefas menores, até que as tarefas sejam suficientemente pequenas e simples, para que sejam facilmente compreendidas. Sendo assim, o paradigma procedimental torna o código mais compreensível e facilita sua manutenção, uma vez que esse

pode ser lido como um conjunto de tarefas simples e independentes. Outro ponto importante é que o paradigma procedimental proporcionou reusabilidade de código, já que um mesmo procedimento pode ser usado em vários programas.

Para ilustrar o uso de procedimentos, o Exemplo 7 mostra a definição de um procedimento em linguagem C que ordena um vetor de números inteiros.

Exemplo 7:

```
void bubble_sort(long[] list, long n)
{
    long c, d, t;

    for (c = 0 ; c < ( n - 1 ); c++)
    {
        for (d = 0 ; d < (n - 1 - c); d++)
        {
            if (list[d] > list[d+1])
            {
                t = list[d];
                list[d] = list[d+1];
                list[d+1] = t;
            }
        }
    }
}
```

O Exemplo 8 mostra como esse procedimento pode ser usado em no corpo de programa também escrito em linguagem C. Note que o código do programa ficou mais simples e fácil de ler do que se o código de ordenação estivesse diretamente no programa principal. Essa modularização de tarefas em procedimentos permite um desenvolvimento mais produtivo (pois procedimentos já feitos e testados podem ser reusados) e de maior manutenabilidade.

Exemplo 8:

```
int main()
{
    int n= 10;
    int[n] array = {0,9,7,5,3,1,2,4,6,8};

    bubble_sort(array, n);
```

```
    printf("Numeros ordenados em ordem ascendente:\n");

    for ( int c = 0 ; c < n ; c++ )
        printf("%ld\n", array[c]);

    return 0;
}
```

4 Paradigma Orientado a Objetos

O paradigma orientado a objetos (SCOTT, 2009) tem suas raízes nas linguagens de programação Simula e Smalltalk nas décadas de 1960 e 1970, respectivamente. Contudo, este paradigma só veio a ser aceito realmente para uso comercial por volta dos anos 1990, quando surgiu a linguagem de programação Java (SEBESTA, 2006), que ficou popularizada pelo fato de se poder programar para todas as plataformas da mesma maneira, o que até então não era possível.

A orientação a objetos também se apresentou com a esperança de suprir algumas das preocupações da indústria do software: a necessidade de criar software comercial mais rapidamente, mais confiável e a um custo baixo. Os meios para conseguir essa proeza encontram-se nas características apresentadas pelo paradigma (apresentadas nas Seções 4.3, 4.4 e 4.5), o que leva a um grande salto quantitativo e qualitativo na produção do software.

Segundo a empresa de software TIOBE Programming Community, que mede a popularidade das linguagens de programação, 9 das 10 linguagens mais populares do mundo em 2015 possuem suporte a esse paradigma. São exemplos de linguagens orientadas a objetos: Java (SEBESTA, 2006), C++ (PEREIRA, 1999), C# (LIMA e REIS, 2002), Python (LUTZ e ASCHER, 1999), etc.

A idéia central deste paradigma é abstrair objetos do mundo real e representá-los com as mesmas características na programação (atributos, ações, modo de fabricação), sendo o programador o responsável por moldar estes objetos, e explicar para estes objetos como eles devem interagir entre si.

As características dos objetos são definidas por meio de classes (LIMA e REIS, 2002), nas quais os comportamentos dos objetos são descritos por métodos (também conhecidos como funções ou subrotinas no paradigma procedimental) e seus estados são descritos por atributos. As classes também descrevem relacionamentos entre objetos (da mesma classe, ou de classes distintas). Ou seja, o software é representado por um conjunto de estruturas de dados que interagem entre si, cada uma delas contendo uma coleção de dados específicos e um conjunto de rotinas para manipular esses dados (ações sobre dados).

Para uma linguagem de programação fazer parte inteiramente do paradigma orientado a objetos, deve implementar seus três mecanismos básicos, ou pilares da orientação a objeto, que são: herança, polimorfismo e encapsulamento (LIMA e REIS, 2002). Esses conceitos serão abordados mais adiante neste capítulo. Todos os exemplos deste paradigma serão escritos utilizando a linguagem de programação C# (exceto quando especificado), que dá suporte ao paradigma orientado a objetos (LIMA e REIS, 2002).

4.1 Classe

Uma classe é uma estrutura que abstrai um conjunto de objetos do mesmo tipo (DEITEI e DEITEI, 2001). É nela que são definidas as características (estrutura) dos objetos daquele tipo, ou seja, que dados (atributos) são armazenados pelo objeto e que operações (métodos) podem ser efetuadas com estes dados.

É importante que cada classe tenha apenas uma ou um pequeno número de responsabilidades. Se uma classe possui excesso de responsabilidades, sua implementação se tornará muito confusa e difícil de manter e entender. Pois, nesse caso, ao alterar uma responsabilidade, você correrá o risco de alterar outro comportamento inadvertidamente. Ela também centralizará muito conhecimento, que seria melhor gerenciado se fosse espalhado. Quando uma classe fica grande demais, ela quase se torna um programa completo e pode cair nos problemas da programação procedimental.

A definição básica de uma classe deverá ter mais ou menos o seguinte aspecto:

```
escopo class NomeClasse
{
    // Construtores
    escopo NomeClasse([parâmetros])
    {
        // Especificações
    }

    //Atributos
    escopo tipo nomeAtributo;
    escopo tipo nomeAtributo;
```

```
    // Métodos
    escopotipoRetornoNomeMétodo([parâmetros])
    {
        // Especificações
    }
}
```

4.1.1 Atributo

Os dados contidos em uma classe são conhecidos como atributos daquela classe e representam características presentes nos objetos do tipo da classe. Cada atributo deve ter um escopo de acessibilidade, um nome e ser de um tipo, que será ou um tipo de dado nativo da linguagem (geralmente tipos inteiro, real, caractere, string, booleano), ou outra classe já existente na linguagem ou definida pelo programador.

Por exemplo, para a classe `Pessoa`, que representa uma abstração de uma pessoa, seus atributos poderiam ser os seguintes: `nome` (cadeia), `idade` (inteiro), `peso` (real) e `rg` (cadeia). Consequentemente cada objeto do tipo `Pessoa` teria estes atributos, cada um deles com um valor específico.

Exemplo 9:

```
public class Pessoa
{
    public string nome;
    public int idade;
    public double peso;
    public string rg;
}
```

Vale ressaltar que um atributo de uma classe também pode ser declarado como sendo um objeto de outra classe. Por exemplo, na classe `Pessoa`, o atributo `rg` poderia ser da classe `Rg` e esta, por sua vez, teria os atributos nome, nome do pai, nome da mãe, data de nascimento, naturalidade, número do RG, etc.

Exemplo 10:

```
public class Rg
{
```

```
    public string nome;
    public string nomePai;
    public string nomeMae;
    public string dataNascimento;
    public string naturalidade;
    public string registroGeral;
}

public class Pessoa
{
    public string nome;
    public int idade;
    public double peso;
    public Rg rg;
}
```

4.1.2 Método

Métodos definem os comportamentos dos objetos da classe, sendo equivalentes às funções e/ou procedimentos do paradigma procedimental, mas diretamente ligados a um objeto ou uma classe. Ou seja, são trechos de código modularizados que executam uma ação com os atributos acessíveis ao seu contexto.

Assim como no paradigma procedimental, além dos métodos poderem ter suas próprias variáveis locais, estes também podem acessar as variáveis globais, que no caso da orientação a objetos são os atributos da classe ou da sua instância em questão. Estes também podem receber ou não parâmetros, e podem retornar um valor a quem o chamou (similar a uma função) ou apenas executar alguma ação com os atributos acessíveis a ele (similar a um procedimento).

Vale ressaltar que quando uma função ou procedimento do paradigma procedimental acessa uma variável global aquela variável é uma variável do programa e é visível a todo o programa, enquanto que quando um método acessa uma “variável global” (atributo definido na classe), ele acessa apenas os atributos do objeto em questão ou atributos próprios da classe, chamados estes de atributos estáticos (i.e., são atributos que não pertencem aos objetos, sendo acessáveis somente pela classe). Ou seja, quando invocamos algum método de um objeto da

classe `Pessoa`, esse método só terá acesso aos atributos do seu objeto, não "enxergando" quaisquer atributos de outros objetos do tipo `Pessoa`.

Como vimos, métodos são extremamente necessários na manipulação dos atributos dos objetos e são utilizados para definir o comportamento destes. Consequentemente são o principal meio de comunicação pelo qual o ambiente externo interage com o ambiente interno do objeto. Na classe `Pessoa`, por exemplo, poderíamos definir os métodos: `falar()`, `comer()` e `obterIMC()`.

Exemplo 11:

```
public class Pessoa
{
    public string nome;
    public double peso;
    public double altura;

    public void falar()
    {
        Console.Write("Olá, meu nome é " + nome + "!");
    }

    public void comer(double kilos)
    {
        peso = peso + kilos;
    }

    Public double obterIMC()
    {
        return peso / (altura * altura);
    }
}
```

4.1.3 Construtores e Destrutores

Construtores de instâncias, ou simplesmente construtores, são métodos chamados automaticamente quando ocorre uma instanciação de uma classe, ou seja, quando um objeto é criado. Estes realizam as ações necessárias para a inicialização correta do objeto, como por exemplo, atribuir valores aos atributos de controle interno do objeto, para que então este possa ser usado corretamente (LIMA e REIS, 2002).

O método construtor deve ter o mesmo nome da classe e não deve ter tipo de retorno, nem mesmo `void`. Um construtor que não tem parâmetros é chamado de construtor default. O exemplo a seguir mostra um possível construtor default para a classe `Pessoa`, onde atribuímos valores default para idade e peso dos objetos da classe.

Exemplo 12:

```
public class Pessoa
{
    public Pessoa()
    {
        idade = 0;
        peso = 3.0;
    }
    ...
}
```

Além da finalidade de preparar o ambiente interno para o uso do objeto, podemos também criar um método construtor que receba parâmetros com os quais podemos atribuir valores aos atributos do objeto durante sua criação, tornando o processo mais rápido.

Exemplo 13:

```
public class Pessoa
{
    public Pessoa(string Nome, int Idade, double Peso)
    {
        nome = Nome;
        idade = Idade;
        peso = Peso;
    }
    ...
}
```

Considerando o construtor parametrizado definido no Exemplo 13, os trechos de códigos apresentados nos Exemplos 14 e 15 são equivalentes (o que mostra um ganho em produtividade ao se utilizar construtores personalizados durante a inicialização do objeto).

Exemplo 14:

```
Pessoa fulano = new Pessoa("Fulano de Tal", 20, 63.3);
```

Exemplo 15:

```
Pessoa fulano = new Pessoa();
fulano.nome = "Fulano de Tal";
fulano.idade = 20;
fulano.peso = 63.3;
```

Em contraste com os métodos construtores, temos os métodos destrutores, que são executados automaticamente quando o escopo em que os objetos estão definidos é finalizado (i.e., quando uma instância de uma classe não pode ser mais referenciada, ela é removida da memória pelo mecanismo de coleta automática de lixo, também denominado *Garbage Collector*. Contudo, quando seu objeto utiliza recursos não gerenciáveis automaticamente, como arquivos, conexões de rede, etc. é necessário usar o destrutor para liberar esses recursos). Seu principal objetivo é a liberação de recursos alocados pelo objeto que será destruído (liberar arquivos, sockets, conexões com banco de dados, etc).

Na linguagem C# (LIMA e REIS, 2002) o destrutor é indicado por um método com o nome da classe, sem tipo de retorno (nem mesmo `void`) e sem parâmetros, com um ~ antes do nome do método.

Exemplo 16:

```
public class Pessoa
{
    ~ Pessoa()
    {
        //código a ser executado antes da
        //destruição do objeto Pessoa
        //pelo Garbage Collector
    }
    ...
}
```

4.2 Objeto

Um objeto ou instância é uma materialização da classe, e pode ser usado para armazenar dados e realizar as operações definidas na classe. Na grande maioria das linguagens orientadas a objetos mais recentes, como Java (DEITEL e DEITEL, 2010) e C# (LIMA e REIS, 2002), a instanciação de um objeto é feita quando o operador `new` é seguido da chamada de algum método construtor. Para que os objetos ou instâncias possam ser manipulados, é necessária a criar uma instância da classe, que é basicamente criar variáveis do tipo da classe. Após alterar o valor de algum dos atributos do objeto é dito que o estado do objeto foi alterado.
Exemplo 17:

```
Pessoa fulano = new Pessoa();
fulano.nome = "Fulano de Tal";
fulano.idade = 20;
fulano.peso = 63.3;
```

4.3 Encapsulamento

4.3.1 Encapsulamento

Encapsulamento permite que detalhes complicados da implementação da classe (regras de negócio, atributos para controle interno, métodos, etc.) sejam escondidos de quem os usa, sendo que estes apenas interagem com o que precisam através de interfaces, permitindo assim ocultar do usuário toda a parte que é irrelevante para este (SCOTT, 2009).

Dessa maneira, podemos pensar em uma classe como uma caixa preta, pois sabemos o que ela faz, conhecemos e interagimos com sua interface externa, mas não nos preocupamos como o processo é feito lá dentro.

A ideia do encapsulamento é tornar o software mais flexível, fácil de modificar e de criar novas implementações, umas vez que cada parte possui sua implementação e realiza seu trabalho independentemente das outras partes (i.e., pacotes, bibliotecas, componentes, classes, métodos, etc.).

4.3.2 Modificadores de Acesso

Nas linguagens de programação orientada a objetos (e.g., C++, Java, C#), em geral, os níveis de encapsulamento são indicados através das seguintes palavras-chaves:

- `public`: é o modificador menos restritivo; ele permite que o membro da classe pode ser acessado externamente a ela, assim como das classes derivadas desta;
- `protected`: esse modificador permite que o membro seja acessado apenas pela própria classe ou pelas classes que derivam desta;
- `private`: é o modificador mais restritivo; ele permite que o membro da classe seja acessado somente de dentro desta, não podendo ser acessado nem mesmo por classes derivadas.

Essas palavras chaves são chamadas de “modificadores de acesso” e são utilizadas para definir quem terá acesso à classe e a seus membros (atributos e métodos), sendo assim, essenciais para o encapsulamento.

Geralmente atributos encapsulados são atributos que não podem ser alterados livremente, por estarem intimamente relacionado a outros atributos e a integridade do objeto, de tal forma que se forem alterados de forma indevida, podem tornar o estado do objeto inconsistente.

Para lidar com atributos encapsulados com o uso dos modificadores `protected` ou `private`, é preciso implementar métodos públicos que acessam e alteram esses atributos, fazendo todo tratamento para que a integridade do objeto seja mantida.

No Exemplo 17 é possível alterar diretamente o atributo `idade` do objeto da classe `Pessoa` de fora da mesma, sem qualquer validação, o que pode quebrar a consistência do objeto, uma vez que nos é permitido atribuir o valor `-1` ao atributo `idade`, o que não deve acontecer, uma vez que torna o objeto inconsistente.

Utilizando a técnica de encapsulamento podemos remover esse atributo da interface do objeto (e consequentemente impedir a atribuição direta de valores a ele)

e implementar um método disponível na interface do objeto que faz a consistência do valor antes de atribuí-lo ao atributo `idade` do objeto.

Exemplo 18:

```
public class Pessoa
{
    protect int idade;
    ...
    public int setIdade(int Idade)
    {
        if (Idade > 0)
        {
            idade = Idade
        }
    }
    ...
}
```

Após isso, o atributo `idade` não é mais visível na interface do objeto, e a única forma de atribuir um valor a ele, é utilizando o método `setIdade()`, que faz a consistência do valor informado.

Como foi tirado o acesso direto ao atributo idade, também precisamos criar e disponibilizar na interface um método que apenas retorna o valor do atributo `idade`. Ficando o exemplo completo como o trecho abaixo:

Exemplo 19:

```
public class Pessoa
{
    protect int idade;
    ...
    public int setIdade(int Idade)
    {
        if (Idade > 0)
        {
            idade = Idade
        }
    }

    public int getIdade()
    {
        return idade;
```

```
        }
        . . .
}
```

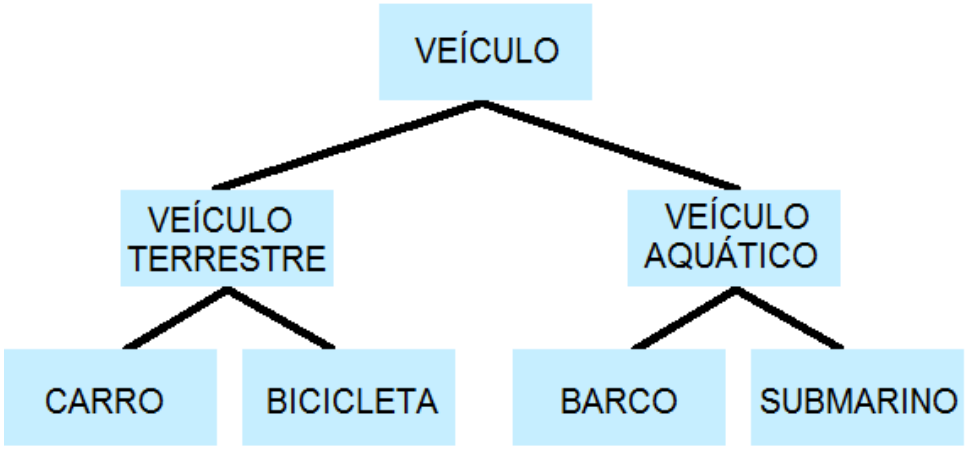
4.4 Herança

4.4.1 Herança Simples

O conceito de herança (WATT, 2004) é construído sobre a ideia de que algumas classes estão intimamente ligadas a outras classes, e que por conta disso, compartilham características em comum (i.e., atributos, métodos, comportamentos, etc.).

Partindo dessa ideia, podemos estabelecer um relacionamento hierárquico entre classes, em que a classe no topo da árvore é a classe mais genérica (geralmente chamada de classe base, ou classe mãe) e à medida que descemos de nível na hierarquia, temos uma nova classe (classe derivada, ou classe filha), que é uma especialização da classe acima. Por conta disso, geralmente nos referimos à relação que vincula uma classe derivada a sua classe base pela frase "Y é um tipo de X" ou "Y é uma especialização da classe X", sendo X a classe base e Y a classe derivada (PEREIRA, 1999).

Figura 2 – Árvore hierárquica representando o conceito de herança aplicado a classe veículo.



Fonte: Elaborada pelo autor.

O mecanismo de herança permite que uma classe seja definida a partir de uma outra classe já existente, com isso a definição da classe derivada herda automaticamente atributos, métodos e comportamentos da classe base. Por conta

disso, herança é considerada uma técnica poderosa, uma vez que encoraja a reusabilidade durante o desenvolvimento e provê uma grande economia de tempo, sem perda de qualidade no desenvolvimento (pois você não precisa reescrever, testar, depurar e manter o código herdado) (DEITEI e DEITEI, 2001).

Para exemplificar, imaginemos que precisamos definir uma classe `Estudante`, a qual possui atributos como número de matrícula, curso, período, etc, ou seja, coisas que remetem à vida escolar de um estudante. Contudo, é relevante também para quem manipulará os objetos do tipo `Estudante`, saber qual o nome do estudante, sua data de nascimento, seu RG, etc., ou seja, coisas que remetem à pessoa propriamente dita.

Podemos então utilizar o mecanismo de herança e definir a classe `Estudante` herdando a classe `Pessoa` que já definimos em exemplos anteriores, uma vez que um estudante é uma especialização de pessoa, ou seja, um tipo especifico de pessoa e que, portanto, além de possuir suas características especificas (matricula, curso, etc.) também deve possuir características de uma pessoa (nome, data de nascimento, etc.).

Utilizando o mecanismo de herança, a definição da classe `Estudante` fica bem mais simples do que escrever a classe desde o início, como ilustrado no Exemplo 20:

Exemplo 20:

```
public class Estudante : Pessoa
{
    public int matricula;
    public string instituicao;
    public string curso;
    public string periodo;
    public int turma;
    public DateTime ingresso;
    public DateTime jubilamento;

    public void apresentarSe()
    {
        Console.Write("Olá, meu nome é " + nome);
        Console.Write(", tenho " + idade + "anos ");
        Console.Write("e atualmente estudo " + curso);
```

```
            Console.Write(" na " + instituicao);
        }
}
```

Note que entre `Estudante` e `Pessoa` há um sinal de dois pontos, o que na definição de uma classe em C# indica que a classe `Estudante` herda a classe `Pessoa`.

Feito isto os objetos do tipo Estudante compartilham as características de uma `Pessoa`, como ilustrado no Exemplo 21:

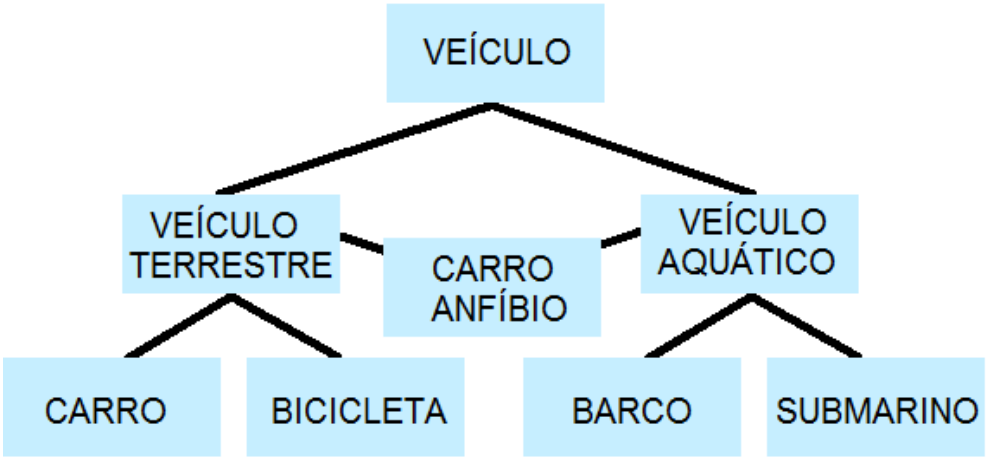
Exemplo 21:

```
Estudante fulano = new Estudante();

fulano.nome = "Fulano de Tal";
fulano.matricula = 13101659;
fulano.instituicao = "FATEC-SP";
fulano.curso = "Análise e Desenvolvimento de Sistemas";

fulano.setIdade(20);
fulano.falar();
fulano.apresentarSe();
```

4.4.2 Herança múltipla

Algumas linguagens orientadas a objetos como C++, Scala e Python também oferecem o mecanismo de herança múltipla, que permite que uma classe herde de mais de uma classe ao mesmo tempo.

Figura 3 – Representação do conceito de herança múltipla aplicado a classe CarroAnfibio.



Fonte: Elaborada pelo autor.

Contudo herança múltipla tem sido uma questão sensível por muitos anos, pois pode aumentar a complexidade na programação e gerar ambiguidade em situações onde mais de uma classe base possui atributos ou métodos como mesmo nome, embora hoje já haja abordagens que resolvem essa ambigüidade (i.e., em C++ por exemplo, é usado `classePai::AtributoAmbiguo` para explicitar de qual classe o atributo ambíguo será acessado). Ilustração no Exemplo 22 da implementação de herança múltipla em C++.

Exemplo 22:

```
class Estudante
{
    public:
        void apresentarSe() { ... }
    protected:
        int idade;
        string nome;
};

class Trabalhador
{
    public:
        void trabalhar() { ... }
    protected:
        double salario;
        string nome;
};

class Estagiario : public Estudante, public Trabalhador
{
    ...
};
```

4.5 Polimorfismo

A palavra polimorfismo vem do grego, πολύς (poli) que significa muitas e μορφή (morphos) que significa formas, ou seja, muitas formas. Em programação, é dito que um objeto é polimórfico quando o sistema de tipo da linguagem atribui mais de um tipo a este (GABBRIELLI e MARTINI, 2006).

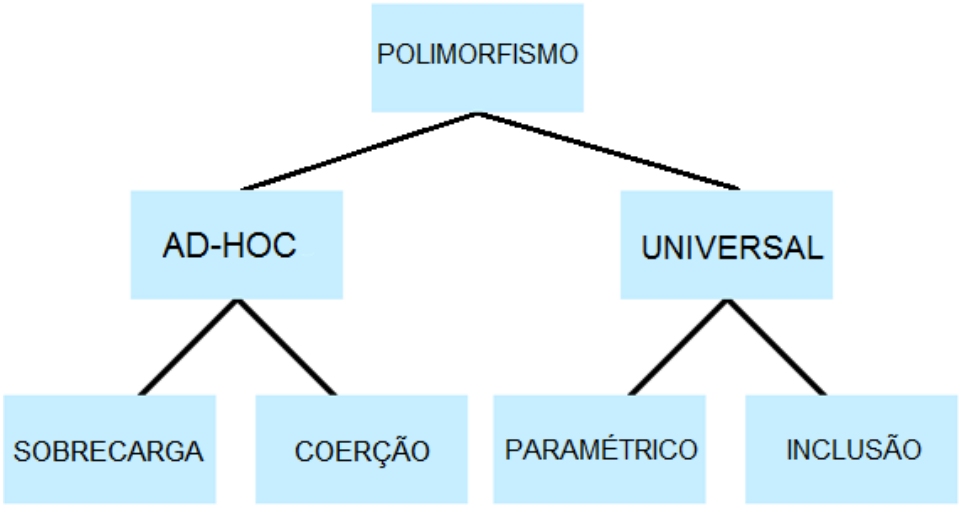
Linguagens tipadas convencionais, como Pascal, baseiam-se na ideia de que funções e procedimentos e, consequentemente, seus parâmetros, têm um tipo

único. Tais linguagens são ditas monomórficas, no sentido de que cada variável pode ser interpretada como sendo de um tipo. Linguagens de programação monomórficas contrastam com as linguagens polimórficas em que algumas variáveis podem ter mais de um tipo (CARDELLI e WEGNER, 1985).

Contudo, mesmo nas linguagens de programação mais convencionais há um certo grau de polimorfismo. Em muitas linguagens, por exemplo, o operador de soma (+), aceita tanto soma entre números inteiros (`int x int -> int`), quanto soma entre números reais (`float x float -> float`), e em algumas linguagens ainda, age como concatenador de strings (`string x string -> string`). Sendo assim, podemos dizer que operador de soma é polimórfico, pois possui mais de um tipo, e assume um deles de acordo com o contexto. Outro exemplo, seria a função `length()`, que retorna o número de elementos em um vetor de tipo `T` (`T[] -> int`), sendo `T` qualquer tipo.

Existem quatro tipos de polimorfismo que uma linguagem pode ter (note que nem toda linguagem orientada a objetos implementa todos os tipos de polimorfismo).

Figura 4 – Tipos de polimorfismo no paradigma orientado a objetos.



Fonte: Elaborada pelo autor.

4.5.1 Sobrecarga

Sobrecarga ocorre quando definimos numa mesma classe métodos com o mesmo nome, mas com assinaturas diferentes, ou seja, que recebem e/ou retornam parâmetros de tipos diferentes (GABBRIELLI e MARTINI, 2006). Esse tipo de

polimorfismo ocorre em tempo de compilação, pois conseguimos saber qual definição do método será invocada para um dado grupo de parâmetros antes mesmo de o código ser executado.

Exemplo 23:

```
public class Pessoa
{
    public void Almocar()
    {
        Console.Write("Estou almoçando");
    }

    public void Almocar(string comida)
    {
        Console.Write("Estou almoçando " + comida);
    }

    public void Almocar(Pessoa pessoa)
    {
        Console.Write("Estou almoçando na companhia de ");
        Console.Write(pessoa.nome);
    }
}
```

4.5.2 Coerção

Coerção permite ao programador converter de maneira implícita alguns tipos para outros tipos, omitindo assim algumas conversões de tipo semanticamente necessárias (GABBRIELLI e MARTINI, 2006).

Em C# por exemplo, é possível converter implicitamente um tipo `int` para `double`, como ilustrado no Exemplo 24.

Exemplo 24:

```
int a = 2;
double b = a;
```

Esse tipo de polimorfismo nos permite, por exemplo, definir uma função de soma, para o tipo `double`, e também usá-la para somar inteiros, bem como somar

variáveis dos dois tipos. O Exemplo 25 ilustra a declaração da função soma com parâmetros do tipo `double`, seguido de seus possíveis usos no Exemplo 26.

Exemplo 25:

```
double soma(double x, double y)
{
    return x + y;
}
```

Exemplo 26:

```
int a = 2;
double b = 3.0;

soma(a, a);
soma(a, b);
soma(b, a);
soma(b, b);
```

Note que para podermos usar a função soma como no exemplo anterior no polimorfismo de sobrecarga, seria necessário definir as quatro versões possíveis da função soma, como ilustra o Exemplo 27:

Exemplo 27:

```
int soma(int x, int y)
{
    return x + y;
}

double soma(int x, double y)
{
    return x + y;
}

double soma(double x, int y)
{
    return x + y;
}
```

```
double soma(double x, double y)
{
    return x + y;
}
```

4.5.3 Paramétrico

Polimorfismo paramétrico permite que uma função ou um tipo de dado seja escrito de forma genérica, de modo que ele possa lidar com valores uniformemente, sem dependendo do seu tipo (PIERCE, 2002).

O polimorfismo paramétrico é extremamente útil quando precisamos criar estruturas de dados genéricas, como listas, árvores, filas, pilhas, etc. Utilizando o polimorfismo paramétrico, podemos definir a estrutura de forma genérica, e especificar o tipo no momento da criação do objeto ou do uso da função.

O Exemplo 28 ilustra a criação da estrutura de dados de pilha genérica. O Exemplo 29 demonstra seu uso na criação de uma pilha para um tipo de objeto especifico.

Exemplo 28:

```
public class Pilha<T>
{
    private T[] elementos;
    private int ultimaPosicao;
    private int topo;

    public Pilha(int tamanho)
    {
        if (tamanho > 0)
        {
            elementos = new T[tamanho];
            ultimaPosicao = tamanho - 1;
            topo = -1;
        }
    }

    public void Push(T valor)
    {
        if (!estaCheia())
```

```
        {
            topo++;
            elementos[topo] = valor;
        }
        else
        {
            throw new Exception("Stack Overflow");
        }
    }

    public T Pop()
    {

        if (!estaVazia())
        {
            T valor = elementos[topo];
            topo--;
            return valor;
        }
        else
        {
            throw new Exception("Stack Underflow");
        }
    }

    public bool estaVazia()
    {
        return topo == -1;
    }

    public bool estaCheia()
    {
        return topo == ultimaPosicao;
    }
}
```

Exemplo 29:

```
Pilha<int> pilhaInteiros = new Pilha<int>();

pilhaInteiros.Push(1);
pilhaInteiros.Push(2);

Pilha<string> pilhaStrings = new Pilha<string>();
```

```
pilhaStrings.Push("Oi");
pilhaStrings.Push("Tchau");
```

Já na definição de uma função genérica, podemos, por exemplo, definir uma função que imprimirá os elementos da Pilha genérica por ordem de inserção na pilha. Assim, podemos imprimir os elementos de uma Pilha de qualquer tipo.

Exemplo 30:

```
void ImprimePilha<T>(Pilha<T> p)
{
    if (!p.estaVazia())
    {
        T elemento = p.Pop();
        ImprimePilha(p);
        Console.WriteLine(elemento);
    }
}
```

4.5.4 Inclusão

Objetos de um subtipo (classe derivada) podem ser manipulados como sendo objetos de algum de seus supertipos (classe base) (WATT, 2004). No Exemplo 22 onde foi visto herança múltipla, a classe `Estagiario` herdava da classe `Estudante` e da classe `Trabalhador`, sendo que `Estudante`, por sua vez, herdava de `Pessoa`. Sendo assim, um objeto da classe `Estagiario` é ao mesmo tempo, dos tipos `Estagiario`, `Estudante`, `Trabalhador` e `Pessoa`, podendo ser tratado como de um de seus supertipos quando necessário.

No Exemplo 31, criamos um objeto do tipo `Estudante`, que será tratado como um objeto do tipo `Pessoa`, uma vez que um estudante também é uma pessoa.

Exemplo 31:

```
Pessoa estudante = new Estudante();
```

Esse tipo de polimorfismo é muito útil quando queremos criar uma função que recebe como parâmetro um objeto de uma classe base (supertipo) ou de qualquer

uma de suas classes derivadas (subtipos), sendo, portanto, um método que aceita vários tipos de uma mesma hierarquia.

Para ilustrar esse exemplo, podemos criar um método que receba um objeto do tipo `Pessoa` e execute um de seus métodos. Esse método também receberá objetos que sejam subtipos de `Pessoa` (e.g., `Estudante`, `Trabalhador`, `Estagiario`, etc.), e que portanto também possuem os mesmos métodos definidos em `Pessoa`, ainda que com implementações diferentes para cada classe derivada.

Exemplo 32:

```
class Pessoa
{
    virtual public string falar()
    {
        return "Sou uma pessoa!";
    }
}

class Estudante : Pessoa
{
    override public string falar()
    {
        return "Sou um estudante!";
    }
}

class Trabalhador : Pessoa
{
    override public string falar()
    {
        return "Sou um trabalhador!";
    }
}

class Program
{
    static void ApresentarSe(Pessoa p)
    {
        Console.WriteLine(p.falar());
    }

    static void Main()
```

```
    {
        ApresentarSe(new Pessoa());
        ApresentarSe(new Estudante());
        ApresentarSe(new Trabalhador());
    }
}
```

4.6 Classe Abstrata

Classe abstrata é uma classe que não pode ser instanciada diretamente, mas que pode ser herdada por outras classes. Nela, definimos atributos, métodos e métodos abstratos, que são métodos com apenas a assinatura definida, sem implementação.

As classes derivadas (através de `override`) é que definirão a funcionalidade de cada atributo ou método abstrato. A ideia de uma classe abstrata é forçar que todas as suas classes derivadas implementem os atributos e métodos definidos nela.

Classes abstratas também podem herdar de outras classes abstratas e uma classe abstrata pode ser herdada por uma classe concreta (i.e., classes instanciáveis, como `Pessoa`, `Trabalhador`, `Estudante`, `Pilha<T>`, etc.).

Um bom exemplo de classe abstrata é a classe `Animal`, uma vez que esta classe em si é uma abstração de todas as espécies de animais, e consequentemente, não existe no mundo real um animal que seja apenas do tipo `Animal`, e sim “animais concretos” que são de alguma raça e espécie específica (e.g., pastor alemão, harpia, pônei, etc., note que esses animais também pertencem a outras classes abstratas, sendo estas cachorro, ave e cavalo, respectivamente, que também herdam de `Animal`), ou seja, que pertencem a alguma classe concreta derivada de `Animal`. O Exemplo 33 ilustra a implementação da classe abstrata `Animal`:

Exemplo 33:

```
abstract class Animal
{
    public bool EstaDormindo;
    public bool EstaFaminto;
```

```
    public abstract void Dormir();
    public abstract void Acordar();
    public abstract void Comer();
}

abstract class Cachorro : Animal
{
    public int Patas;
    public bool Calda;
    public bool Macho;

    public abstract void Latir();
    public abstract void Correr();
}

class PastorAlemao : Cachorro
{
    //implementação dos métodos abstratos de forma
    //especifica para esse tipo de Animal/Cachorro
    //bem como definição de atributos
    //e métodos próprios desse tipo
}
```

4.7 Interface

Na interface definimos atributos e métodos sem implementar nada do seu código (apenas assinaturas) e a classe que implementa a interface é obrigada a fornecer (implementar) o código definidos na interface. Vale também evidenciar que uma classe pode implementar mais de uma interface, sendo esta uma alternativa para compartilhamento de características entre classes em linguagens que não suportam herança múltipla.

Por exemplo, podemos pensar no código de barras de um produto como uma interface, uma vez que são estabelecidos padrões de código de barra, e é necessário que todo produto implemente esses padrões para poder ser identificado. O Exemplo 34 ilustra a implementação da interface `IcodigoBarras`, bem como sua implementação.

Exemplo 34:

```
public interface ICodigoBarras
{
```

```
    void LeituraPadrao3of9( );
    void LeituraPadraoEAN13( );
}

public class Produto : ICodigoBarras
{
    public void LeituraPadrao3of9( )
    {
        // Implementação
    }

    public void LeituraPadraoEAN13( )
    {
        // Implementação
    }
}
```

5 Paradigma Funcional

Como visto nos capítulos anteriores, o paradigma orientado a objeto baseia-se em conceitos similares àqueles do paradigma procedimental. Esse alto grau de similaridade entre paradigmas provém, em parte, de uma de suas bases comuns de projeto: a arquitetura de von Neumaan (GARANHANI, 2008). Consequentemente as linguagens imperativas são eficientes em computadores com essa arquitetura.

Contudo existem outros paradigmas que focam em resolver outros tipos de problemas ou voltados a alguma metodologia de programação particular em vez de uma execução eficiente em uma arquitetura particular de computador. Até agora, entretanto, a reduzida eficiência ao executar programas escritos nesses paradigmas tem impedido que eles se tornem tão populares quanto o paradigma imperativo.

O paradigma funcional baseia-se no conceito matemático de função, em que para cada elemento do seu conjunto domínio (entrada) há apenas um elemento no seu conjunto contradomínio (saída). Programas escritos nele são definições de funções e de especificações de aplicação destas, e as execuções consistem em avaliá-las. Isso resulta em uma abordagem à solução de problemas que difere fundamentalmente dos métodos usados nos paradigmas já vistos até então. Todos os exemplos deste paradigma serão escritos utilizando a linguagem de programação Haskell (exceto quando especificado), que dá suporte ao paradigma funcional (LIPOVAčA, 2011).

Em uma linguagem de programação puramente funcional não se usa variáveis ou instruções de atribuição, há apenas a definição de valores, os quais são constantes durante toda sua existência no escopo. Consequentemente, construções iterativas (e.g., `while`, `for`, `repeat`) não são possíveis já que elas são controladas por variáveis, em vez disso, são usadas funções recursivas. Sem variáveis, a execução de um programa puramente funcional não tem nenhum estado em termos de semântica operacional e denotacional (SEBESTA, 2006).

Os Exemplos 35 e 36 ilustram as definições recursivas das funções fatorial e fibonacci, respectivamente.

Exemplo 35:

```
fat 0 = 1
fat n = n * fat (n-1)
```

Exemplo 36:

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

5.1 Transparência Referencial

Assim como as funções matemáticas, a execução de uma função no paradigma funcional sempre produz o mesmo resultado quando dados os mesmos parâmetros (SEBESTA, 2006). Isso se chama transparência referencial e não só permite que compilador aplique alguns tipos de otimização (e.g., memoização e eliminação de subexpressões comuns), como também permite ao programador deduzir facilmente (e até mesmo provar) que uma função está correta e, em seguida, construir funções mais complexas juntando diversas funções simples.

Exemplo 37:

```
square x = x * x
```

A função `square`, no Exemplo 37, que recebe um número e devolve seu quadrado, possui transparência referencial, uma vez que o seu valor de retorno depende exclusivamente do parâmetro de entrada, e consequentemente, também retornará sempre o mesmo valor para invocações como os mesmos parâmetros.

5.2 Efeitos Colaterais

Em paradigmas que possuem o conceito de estado do programa, como o procedimental e orientado a objetos, se a avaliação de uma função ou expressão gera uma mudança de estado no programa sem ser decorrente do valor retornado pela função, é dito que a função causa efeitos colaterais (*side-effect*) (WATT, 2004). O efeito colateral mais comum é uma atribuição a uma variável global. Na presença de efeitos colaterais, o comportamento de uma função pode depender da sua ordem de execução no programa, uma vez que alguma outra expressão ou função pode

modificar o valor de alguma variável que está faz uso. Compreender uma função com efeitos colaterais requer conhecimento sobre o contexto em que ela está sendo executada, bem como conhecer bem as possíveis relações desta com o “mundo exterior”.

Imagine que dentro do escopo ao qual a função `square` foi definida, existe uma variável que representa a quantidade de vezes que precisamos calcular o quadrado de um número, e que dentro da função `square`, somamos um nessa variável “global”. A função `square`, portanto passa a ter efeitos colaterais uma vez que altera o estado do programa sem ser pelo seu retorno, contudo, ela não deixa de ter transparência referencial, pois o retorno ainda depende apenas do parâmetro de entrada.

O Exemplo 38 ilustra esse cenário; contudo, como Haskell é uma linguagem puramente funcional e não há o conceito de estado do programa, o código foi escrito em JavaScript.

Exemplo 38:

```
var quadradosCalculados = 0;

functionsquare(x)
{
    quadradosCalculados = quadradosCalculados + 1;
    return x * x;
}
```

Agora imagine que temos uma outra função `f` que retorna o número de quadradosCalculados até o momento, obviamente o retorno dessa função depende da sua ordem execução no programa. Por exemplo, se chamarmos `f` antes de executar `square`, `f` retornará `0`, e se executarmos imediatamente depois retornará `1`, e assim por diante. Sendo assim, `f` também possui efeitos colaterais e não tem transparência referencial.

5.3 Funções Puras

Uma função é considerada pura se possui transparência referencial e é livre de efeitos colaterais (HAVERBEKE, 2004). Em linguagens puramente funcionais, é

permitido apenas a definição de funções puras, uma vez que na matemática todas funções são naturalmente puras.

Funções puras possuem algumas propriedades úteis durante a programação:

1. se o resultado de uma função não é usado, a chamada da função pode ser removida sem afetar outras expressões;
2. uma vez que possui transparência referencial, o compilador pode utilizar técnicas de otimização;
3. se não há dependência de dados entre duas ou mais funções puras, a ordem de execução entre elas é irrelevante. O que permite naturalmente um processamento paralelo, uma vez que uma não interfere na outra;
4. Se uma linguagem de programação não permite efeitos colaterais, qualquer estratégia de avaliação de expressão pode ser usada, o que dá ao compilador maior liberdade para reordenar a avaliação das expressões no programa.

5.4 Funções de Primeira Classe e de Ordem Superior

Nas linguagens de programação em geral, um valor, entidade ou objeto é dito como cidadão de primeira classe (*first-class citizen*) se este puder ser passado como parâmetro para uma função, ser retornado por uma função, bem como ser atribuído a uma variável ou estrutura de dados (SCOTT, 2009). Um bom exemplo são inteiros e caracteres, que são objetos de primeira classe para a maioria das linguagens de programação.

No paradigma funcional funções são tratadas como cidadãos de primeira classe, e com isso podemos ter uma função que receba uma ou mais funções como argumento, ou ainda, que retorna outra função como resposta. Estas funções são chamadas funções de ordem superior.

Um exemplo simples de uma função de ordem superior é a função `map`, no Exemplo 39, que recebe como argumento uma função e uma lista, e retorna uma nova lista formada pela aplicação da função a cada elemento da lista passada a ela. Para uma linguagem suportar `map`, ela deve tratar funções como cidadãos de primeira classe.

Exemplo 39:

```
map f [] = []
map f (x:xs) = f x : map f xs
```

A primeira linha nos diz que se a função `map` receber uma função e uma lista vazia, deve retornar uma lista vazia, uma vez que não há elementos para aplicar a função `f`. A segunda linha diz que se `map` receber uma função e uma lista não vazia, ele deve entender `x` como o primeiro elemento da lista, e todo resto como `xs`, a partir disso, aplicará a função `f` à `x` e adicionará o novo valor gerado por `f` à lista gerada pelo uso recursivo da função `map`. No Exemplo 40, vamos criar uma função que retorna o quadrado de um número, e então passar essa função para `map` junto com uma lista de números. O programa retorna como resultado a seguinte lista: `[1,4,9,16,25]`.

Exemplo 40:

```
square x = x * x
map square [1,2,3,4,5]
```

Funções de ordem superior são extremamente úteis para a abstração de um modo geral, pois estas nos permitem não mais abstrair apenas sobre valores, mas também sobre ações. Segundo Hughes, estas funções aumentam a modularidade durante a programação (HUGHES, 1990), o qual este considera ter grande importância para uma programação bem sucedida. Por exemplo, considere que precisamos criar duas funções, uma que retorne a soma dos valores de uma lista e outra que retorne o produto também de uma lista. O Exemplo 41 ilustra esse cenário.

Exemplo 41:

```
sum [] = 0
sum (x:xs) = add x (sum xs)

prod [] = 1
prod (x:xs) = mul x (prodxs)
```

Contudo, percebemos que há um padrão na criação dessas funções e que possivelmente podemos criar outras funções seguindo essa mesma linha. É fácil

perceber que a função (`add`/`mul`) e o valor inicial (`0`/`1`) são elementos variáveis e poderíamos parametrizá-los. Com isso, é fácil criar uma função genérica, a qual chamaremos de `fold`.

Exemplo 42:

```
(fold f init) [] = init
(fold f init) (x:xs) = f x ((fold f init) xs)
```

Os parênteses em `(fold f init)` são desnecessários, e foram utilizados apenas para ênfase. A partir de `fold` podemos criar as definições para `sum` e `prod` de uma forma bem mais simples:

Exemplo 43:

```
sum = fold add 0
prod = fold mul 1
```

5.5 Expressão Lambda

Nas linguagens de programação funcionais podemos definir funções de várias formas. Um dos jeitos é definirmos expressões e em seguida nomeá-las, que é o jeito que fizemos até então, mas um outro jeito é criar uma expressão lambda.

Expressões lambdas são basicamente funções anônimas que criamos porque precisamos de uma função apenas uma vez. Geralmente, uma função lambda é escrita com o único propósito de passá-la para uma função de ordem superior (LIPOVAčA, 2011).

Para construir uma expressão lambda em Haskell, nós escrevemos o sinal de barra (`\`, que remete à letra grega lambda: λ) seguido dos parâmetros separados entre espaços. Após estes, escrevemos `->` e então o corpo da função.

Um bom exemplo de aplicação de expressão lambda é o Exemplo 40, onde definimos a função `square` apenas para com o intuito passá-la como parâmetro de `map`. Utilizando expressão lambda, o código fica bem mais simples, como demonstrado o Exemplo 44:

Exemplo 44:

```
map (\x -> x * x) [1,2,3,4,5]
```

Outro exemplo da utilidade das funções lambdas é usá-las para criar funções que serão retornadas por alguma função de ordem superior. Por exemplo, imaginemos um cenário onde precisamos definir algumas funções de segundo grau, conforme o Exemplo 45:

Exemplo 45:

```
f1 x = 3 * (x * x) + 4 * x + 1
f2 x = 6 * (x * x) + 7 * x + 5
f3 x = 5 * (x * x) + 3 * x + 2
```

É fácil perceber que há um padrão na criação das funções uma vez que todas elas são do mesmo tipo (funções de segundo grau), e que há também uma repetição de código, o que certamente facilita a criação de erros e dificulta a manutenção. Poderíamos resolver isso construindo uma função genérica que cria nossas funções de segundo grau, uma vez que indiquemos a ela quais são as constantes da função (no caso de `f1` seriam `3`,`4`,`1`). Vamos definir essa função como `quadraticFunction`, como mostra o Exemplo 46:

Exemplo 46:

```
quadraticFunction a b c = \x -> a * (x * x) + b * x + c
```

Com isso, podemos definir facilmente nossas funções de segundo grau, como ilustrado no Exemplo 47:

Exemplo 47:

```
nf1 = quadraticFunction 3 4 1
nf2 = quadraticFunction 6 7 5
nf3 = quadraticFunction 5 3 2
```

Vale ressaltar que `f1` e `nf1` possuem exatamente a mesma definição: ambas são funções que recebem um parâmetro `x` e o aplicam à expressão `3 * (x * x) + 4 * x + 1`, a única diferença é que em `f1`, sua definição é feita estaticamente, enquanto `nf1` foi criada dinamicamente. Mais uma vez também, podemos evidenciar a importância de funções como objetos de primeira classe e funções de ordem superior.

5.6 Polimorfismo Paramétrico

O polimorfismo paramétrico nos permite definir funções que recebem como parâmetro elementos pertencentes a uma determinada família de tipos (WATT, 2004). Através desse recurso, podemos definir funções mais genéricas e expressivas, mantendo a segurança da tipagem estática. Tais funções são chamadas de funções genéricas e junto com tipos de dados genéricos, formam a base da programação genérica.

Utilizando polimorfismo paramétrico, podemos definir uma função `length` que retorna o tamanho de uma lista, seja essa lista uma lista de caracteres, números, ou qualquer outro tipo de dado.

Exemplo 48:

```
length [] = 0
length (x:xs) = 1 + length xs
```

Por exemplo, quando criamos a função `sum`, definimos que ela receberá uma lista de números e retornará a soma destes, contudo, note que não falamos se a lista é de números naturais, inteiros, reais, etc. Ou seja, a função `sum`, foi definida de forma genérica para somar vários tipos de número.

Exemplo 49:

```
sum [] = 0
sum (x:xs) = x + sum xs
```

Outro exemplo é a função `map`, no Exemplo 50, que recebe como parâmetro uma função `f` e uma lista `l`, e retorna uma nova lista, gerada pela aplicação de `f` para cada elemento da lista `l`. Note que não falamos o tipo da função `f` (i.e., não informamos o tipo do parâmetro que `f` recebe, nem o tipo do retorno) bem como não falamos o tipo da lista (i.e., não falamos se será uma lista de números, de caracteres, de strings, etc.).

Exemplo 50:

```
map f [] = []
map f (x:xs) = f x : map f xs
```

Perceba, contudo, que o que é realizado com o parâmetro recebido pela função polimórfica determina quais tipos de dados podem ser passados à função. No caso da função `length`, não é realizada nenhuma expressão com nenhum elemento da lista, logo o tipo da lista não importa, então essa função funciona para qualquer tipo de lista. Já a função `sum`, aplica o operador/função de soma aos elementos da lista recebida como parâmetro, o que obriga a estes serem "somáveis". Por fim temos o `map`, que pode receber qualquer função `f` e qualquer lista `l`, desde que a `f` receba um valor do mesmo tipo dos elementos de `l`.

6 Paradigma Lógico

O paradigma lógico emergiu na década de 1970, sendo fundamentalmente distinto dos outros paradigmas. A abordagem do paradigma lógico é expressar programas na forma de lógica simbólica e usar um processo de inferência lógica para produzir resultados, sendo algumas de suas principais aplicações nas áreas de inteligência artificial, processamento de linguagem natural, criação de sistemas especialistas e prova de teoremas (TUCKER e NOONAN, 2007).

Linguagens de programação lógica permitem ao programador declarar um conjunto de fatos a partir do qual teoremas podem ser provados. O programador declara teoremas ou objetivos, e a implementação da linguagem tenta encontrar uma coleção de fatos e passos de inferência que, juntos, implicam no objetivo (SCOTT, 2009). Dentre as várias linguagens lógicas existentes, Prolog é de longe a mais popular, sendo também a maior responsável pela difusão do paradigma, por isso todos os exemplos na apresentação do paradigma serão escritos nesta linguagem (exceto quando especificado).

6.1 Fatos

A base de um programa lógico consiste de fatos, estes são proposições (declarações lógicas) que não possuem nenhuma regra para sua veracidade, e portanto, são incondicionalmente verdadeiras. A partir da declaração de fatos, podemos estruturar novos fatos, bem como estabelecer relações entre esses, tudo isso através da inferência lógica (SEBESTA, 2006). O Exemplo 51 ilustra a declaração de alguns fatos em Prolog.

Exemplo 51:

```
homem(paulo).
homem(lucas).
mulher(maria).
mulher(joana).
pai(paulo, lucas).
pai(paulo, joana).
mae(maria, lucas).
mae(maria, joana).
```

A primeira linha, por exemplo, afirma que `paulo` é `homem`, e a terceira que `maria` é `mulher`. Já a quinta linha estabelece que `paulo` é pai de `lucas`, ou seja, que a relação pai existe entre esses dois "objetos". Na programação lógica, identificadores de relacionamentos (e.g., `homem`, `mulher`, `pai`, `mae`, etc.) são denominados predicados e identificadores de objetos (e.g., `paulo`, `maria`, etc.) são denominados átomos. Em Prolog, ambos devem sempre iniciar com letra minúscula.

6.2 Consultas

Uma vez que temos a declaração de fatos em nosso programa, podemos fazer perguntas sobre esses ao sistema, a fim de saber se algum relacionamento entre objetos é verdadeiro, ou ainda quais os possíveis objetos de um relacionamento. A consulta `?- homem(paulo).` por exemplo, pergunta se `paulo` é `homem`.

Como há um fato que estabelece a relação que `paulo` é `homem`, o sistema nos retornará `yes`. Outro tipo de consulta é quando queremos saber quais os possíveis objetos de um relacionamento. Por exemplo, a consulta `?- pai(paulo, X).` pergunta ao sistema quem são os filhos de `paulo`, ou seja, de quem `paulo` é `pai`.

Por default, o sistema trará o primeiro valor satisfatório encontrado para a consulta, e portanto retornará `X = lucas`. Caso desejássemos saber outros possíveis valores para `X`, bastaria entrar com um ponto-e-vírgula, e o sistema retornaria também `X = joana`, e caso desejássemos encerrar a consulta, ao invés do ponto-e-vírgula apertaríamos enter. Note que `X` é uma variável, e que em Prolog deve começar com letra maiúscula.

6.3 Regras

Durante a construção de relações, podemos também adicionar regras / condições a esta para que a relação seja verdadeira. Por exemplo, imagine que queremos criar um predicado que relacione um número `X` como sendo par. É necessário então que na declaração desse predicado verifiquemos se o resto da

divisão entre X e 2 é 0 para então podermos concluir que a relação é verdadeira. O Exemplo 52 ilustra o programa:

Exemplo 52:

```
par(X) :- X mod 2 =:= 0.
```

O predicado acima pode ser lido como "X é par se X mod 2 for igual a 0", ou seja, para X ser considerado par, é necessário que cumpra essa condição. Outra coisa que regras também nos permitem é definir novas relações em termos de relações já existentes. Por exemplo, a partir das relações pai e mae podemos criar a relação pais, que relaciona X com sendo um dos pais (pai ou mãe) de Y; podemos ainda, com base na relação pais, definir a relação irmao, que será verdadeira se X e Y tiverem pais em comum. O programa é ilustrado no Exemplo 53.

Exemplo 53:

```
pais(X,Y) :- pai(X,Y).
pais(X,Y) :- mae(X,Y).

irmao(X,Y) :- pais(Z,X), pais(Z,Y), X \= Y.
```

Note que graças ao uso de regras, podemos facilmente escrever um programa que reproduza o clássico silogismo "Todo homem é mortal. Sócrates é homem. Logo, Sócrates é mortal.". Para ver o resultado computado pelo programa do Exemplo 54, o usuário deve fazer a consulta: `?- mortal(sócrates).` A resposta do sistema será `yes`.

Exemplo 54:

```
homem(sócrates).

mortal(X) :- homem(X).
```

6.4. Reversibilidade das Relações

Permite que o sistema não diferencie entre argumentos de entrada e saída de um predicado, podendo um determinado argumento ser ora parâmetro de entrada, ora de saída (PALAZZO, 1997). Ou seja, a execução pode ocorrer em qualquer

sentido, dependendo do contexto. Um exemplo simples de reversibilidade é quando definimos a relação `pai`, e a partir desta podemos fazer tanto consultas para descobrir quem são os filhos de um `pai`, como quem é o `pai` de um filho.

Para o programa ilustrado no Exemplo 55, por exemplo, a consulta `?- pai(paulo,Y)` retornará `joana`, e a consulta `?- pai(X, joana)` retornará `paulo`. Note que para o mesmo predicado, na primeira consulta o primeiro parâmetro era o parâmetro de entrada e o segundo de saída, enquanto que, na segunda consulta o primeiro parâmetro foi o de saída e o segundo o de entrada.

Exemplo 55:

```
pai(paulo, joana).
```

Um caso prático de utilidade dessa propriedade do paradigma lógico pode ser encontrado no processamento de listas. Imagine que precisamos definir um predicado que concatena duas listas para formar uma terceira. A consulta `?- concat([1,2],[3,4,5],X).` no programa ilustrado no Exemplo 56 retornará `X = [1,2,3,4,5].`

Exemplo 56:

```
concat([],Ys,Ys).
concat([X|Xs],Ys,[X|Z]) :- concat(Xs,Ys,Z).
```

A partir de `concat`, podemos definir outros dois predicados: `prefix` e `suffix`, que verificam se uma lista `Xs` está presente no começo de uma lista `Zs` e se uma lista `Ys` está presente no fim de uma lista `Zs`, respectivamente. Executar a consulta: `?- prefix([1,2,3],[1,2,3,4,5,6,7])` no programa ilustrado no Exemplo 57, retornaria `yes`.

Exemplo 57:

```
prefix(Xs, Zs) :- concat(Xs,Ys,Zs).
suffix(Ys, Zs) :- concat(Xs,Ys,Zs).
```

7 Análise Comparativa

Como foi apresentado ao longo deste trabalho, existem vários paradigmas de programação, cada um com uma abordagem fundamentalmente diferente das demais, determinando a maneira como o programador pensa sobre a tarefa de programar (PAPERT, 1991), através dos conceitos, técnicas, abstrações e limitações que fornece.

No decorrer desse capitulo compararemos os paradigmas vistos até então quanto algumas características geralmente desejadas para se atingir uma programação bem sucedida.

7.1 Expressividade

7.1.1 Legibilidade

Uma vez que os paradigmas procedimental e orientado a objetos são imperativos, nestes o programador precisa definir exatamente como quer que o programa seja computado, explicitando, além da lógica do algoritmo, todo o fluxo de controle e manipulação dos dados. Por sua vez, nas abordagens dos paradigmas funcional e lógico, que são declarativos, o programador delega ao sistema todo o fluxo de controle da execução do programa, permitindo o mesmo apenas declarar as estruturas e funções necessárias para a solução do problema.

Baseado nesse fato podemos argumentar que a abordagem imperativa faz com que um programa seja mais difícil de ler se comparado a declarativa, pois uma vez que é necessário explicitar sobre qual fluxo de controle o programa deve ser computado, parte do código passar a ser mais para a máquina do que para o ser humano, o que pode inclusive dificultar a compreensão deste quanto a ideia do algoritmo em si, uma vez que para o ser humano compreender o processo de resolução de um problema, nem sempre é preciso explicitar todo o fluxo, bastando uma abstração adequada. Como exemplo, podemos usar os Exemplos 1 e 2, que objetivam listar todos os binários de 3 dígitos.

Exemplo 1:

```
#include <stdio.h>

int main()
{
   for(int A=0; A<2; A++)
      for(int B=0; B<2; B++)
         for(int C=0; C<2; C++)
            printf("%d%d%d\n",A,B,C);
   return 0;
}
```

Exemplo 2:

```
dígito(0).
dígito(1).
binário(N) :- N=(A,B,C), dígito(A), dígito(B), dígito(C).
```

Vale ressaltar também que, como na programação declarativa os dados são imutáveis, o programa é livre de efeitos colaterais e possui transparência referencial, uma vez que você entende como algo funciona, você não precisa mais se preocupar se em algum momento especifico pode acontecer um erro naquele código caso o programa seja executado de alguma forma não planejada. Essa característica torna a leitura do código muito mais simples, uma vez que o resultado de todas as partes do programa não compartilham estados em comum e são independentes das demais partes (e de sua ordem de execução), permitindo o programador se concentrar em entender apenas a parte desejada, abstraindo as demais partes, sem receio de que o não conhecimento delas possa facilitar um erro.

7.1.2 Capacidade de Escrita

Podemos também analisar a expressividade dos paradigmas quanto à quantidade de esforço necessário para se escrever um programa nele, ou seja, por quão fácil é usar um determinado paradigma para representar um problema e sua solução.

Cada paradigma foi feito para expressar de forma simples e natural um tipo de problema, consequentimente um programa relativamente simples pode se tornar

complexo num paradigma inapropriado. O Exemplo 58, 59, 60 e 61 ilustram o clássico programa `Hello Word` em cada um dos paradigmas apresentados: procedimental, orientado a objetos, funcional e lógico, respectivamente.

Exemplo 58:

```
void main()
{
    printf("Hello, world");
}
```

Exemplo 59:

```
public class HelloWorld
{
    public static void main()
    {
        Console.Write("Hello, World");
    }
}
```

Exemplo 60:

```
print "Hello, Word"
```

Exemplo 61:

```
?- writeln('Hello World').
```

Note que nos paradigmas imperativos foi necessário declarar não só a instrução de mostrar a mensagem na tela, mas as também estruturas necessárias para o paradigma: para o procedimental uma função e para o orientado a objetos uma classe e um método, exigindo também conceitos de modificadores de acesso, que é um conceito plenamente desnecessário para um programa simplista como o `Hello Word`.

Como temos visto, o paradigma declarativo permite ao programador tratar a programação de forma mais abstrata, o que além de deixar o código mais expressivo e simples, também exige menos esforço durante a codificação. Como demonstração, o Exemplo 62, 63 e 64 ilustram como seria um programa que recebe um conjunto de números e retorna o maior nos paradigmas procedimental, funcional e lógico, respectivamente. Note que o paradigma orientado a objetos não aparece

no exemplo, pois seu código seria igual ao do procedimental, apenas estando dentro de uma classe (como no caso do programa `Hello Word`).

Exemplo 62:

```
int maxList(int[] array, int size)
{
    var aux = array[0];

    for (var i = 1; i < size; i++)
        if (array[i] > aux)
            aux = array[i];

    return aux;
}
```

Exemplo 63:

```
maxList (x:[]) = x
maxList (x:xs) = if x >= m then x else m
                 where m = maxList xs
```

Exemplo 64:

```
maxList([X],X).
maxList([X|Xs],X):- maxList(Xs,M), X >= M.
maxList([X|Xs],M):- maxList(Xs,M), M > X.
```

Note que além da definição nos paradigmas funcional e lógico ser mais concisos, o programa consegue não só para encontrar o maior valor de um conjunto de inteiros, mas também para qualquer tipo comparável (caractere, cadeia, reais, booleanos, etc.), pois usa polimorfismo paramétrico, que não é um conceito presente na programação procedimental (mas presente na orientação a objetos), nesta, o programador seria obrigado a criar outra função para cada tipo, dificultando assim a escrita do programa.

Conforme vimos no capitulo cinco e nos Exemplos 62, 63 e 64, muitas técnicas e conceitos do paradigma funcional simplificam consideravelmente a escrita do código, como: polimorfismo paramétrico (presente também na orientação a objetos, mas introduzido pelo paradigma funcional), funções como objetos de primeira classe e expressões lambdas.

Em algumas linguagens imperativas como a linguagem C, é possível simular funções de ordem superior através do uso de ponteiros. Abaixo os Exemplos 65, 66 e 67 demonstram essa situação em linguagem C, seguido de seus correspondentes em Haskell e Prolog, respectivamente.

Exemplo 65:

```
void map(int (*f)(int), int x[], int size)
{
    for (int i = 0; i < size; i++)
        x[i] = f(x[i]);
}

int f(int x)
{
    return 3 * x + 1;
}

int main()
{
    int list[] = {1, 2, 3, 4, 5};
    map(f, list, 5);
}
```

Exemplo 66:

```
map f [] = []
map f (x:xs) = f x : map f xs

map (\x -> 3 * x + 1) [1, 2, 3, 4, 5]
```

Exemplo 67:

```
f(X,Y) :- Y is X * 3 + 1.

map([],F,[]).
map([X|Xs],F,[Y|Ys]) :- call(F,X,Y), map(Xs,F,Ys).
```

Contudo, note que além do código do paradigma procedimental exigir muito mais esforço de escrita que seus correspondentes, ainda seria necessário definir uma função `map` para cada tipo, já que polimorfismo paramétrico é um conceito funcional. Já no paradigma orientado a objetos, seria possível definindo a função

`map` como uma função que recebe um vetor de T e um ponteiro de função que recebe T e devolve U, algo similar ao pseudocódigo abaixo:

Exemplo 68:

```
public class Program
{
    public static U[] map(T[] oldArray, U (*f)(T))
    {
        int tamanho = oldArray.length;

        U[] newArray = new U[tamanho];

        for(int i = 0; i < tamanho; i++)
            newArray[i] = f(oldArray[i]);

        return newArray;
    }

    public static int f(int x)
    {
        return 3 * x + 1;
    }

    public static void Main()
    {
        var list = new int[] {1, 2, 3, 4, 5};
        map(list, f);
    }
}
```

É fácil notar que ainda assim o código ficou extremamente maior que o do paradigma funcional e lógico. Outro ponto relevante é que o uso de funções de ordem superior por si só podem aumentar em muito a expressividade do código. Um bom exemplo é imaginar um cenário onde temos uma lista é precisamos aplicar um filtro qualquer sobre essa, o que é uma atividade comum em qualquer paradigma. Os Exemplos 69 e 70 demonstram como seria esse trecho de código nos paradigmas orientado a objetos e no funcional, respectivamente.

Exemplo 69:

```
public static List<int> getPares(List<int> numeros)
{
    var pares = new List<int>();
```

```
    for (int i = 0; i < numeros.Count; i++)
        if (numeros[i] % 2 == 0)
            pares.Add(numeros[i]);
}
```

Exemplo 70:

```
filter f [] = []
filter f (x:xs) = if f x then x : filter f xs else filter f xs

getPares numeros = filter (\x -> mod x 2 == 0) numeros
```

Perceba que além do código do paradigma funcional ter ficado muito mais simples e conciso do que o orientado a objetos, seria também muito mais fácil criar novas funções de filtro se necessário; se quiséssemos criar uma função `getImpares` por exemplo, no paradigma funcional só precisaríamos adicionar mais uma linha para a nova função, enquanto que no orientado a objetos teríamos que criar outra função quase idêntica a `getPares`, apenas mudando a comparação do `if`, ou então fazer uso de polimorfismo paramétrico combinado com ponteiros de função (similar ao Exemplo 68) para definir uma função de filtro genérica, o que ainda assim, tornaria o código ainda menos expressivo e complexo.

Como último exemplo de comparação, os Exemplos 71, 72 e 73 definirem o algoritmo `quicksort` de forma imperativa, funcional e lógica, respectivamente:

Exemplo 71:

```
void quicksort(ref T[] list, int size)
{
    if (size == 0)
        return;

    var smallerList = new T[size];
    var largerList = new T[size];

    T pivot = list[0];

    int numSmaller = 0, numLarger = 0;

    for (int i = 1; i < size; i++)
        if (list[i] < pivot)
```

```
            smallerList[numSmaller++] = list[i];
        else
            largerList[numLarger++] = list[i];

    quicksort(ref smallerList, numSmaller);
    quicksort(ref largerList, numLarger);

    int pos = 0;

    for (int i = 0; i < numSmaller; i++)
        list[pos++] = smallerList[i];

    list[pos++] = pivot;

    for (int j = 0; j < numLarger; j++)
        list[pos++] = largerList[j];
}
```

Exemplo 72:

```
filter f [] = []
filter f (x:xs) = if f x then x : filter f xs else filter f xs

quicksort [] = []
quicksort (x:xs) = quicksort lowers ++ [x] ++
                   quicksort largers
                   where lowers  = filter (>=x) xs
                         largers = filter (< x) xs
```

Exemplo 73:

```
concat([],Ys,Ys).
concat([X|Xs],Ys,[X|Z]) :- concat(Xs,Ys,Z).

partition([],Y,[],[]).
partition([X|Xs],Y,[X|Ls],Rs) :- X <= Y,partition(Xs,Y,Ls,Rs).
partition([X|Xs],Y,Ls,[X|Rs]) :- X  > Y,partition(Xs,Y,Ls,Rs).

quicksort([],[]).
quicksort([X|Xs],Ys) :-  partition(Xs,X,Left,Right),
                         quicksort(Left,Ls),
                         quicksort(Right,Rs),
                         append(Ls,[X|Rs],Ys).
```

Como demonstrado ao longo desta seção, fica evidente que programas declarativos, especialmente os funcionais, tendem a ser mais expressivos e concisos que seus correspondentes imperativos.

7.2 Confiabilidade

Pelo fato de programas desenvolvidos no paradigma imperativo terem seu estado representado por variáveis, e essas poderem ser compartilhadas entre diferentes partes do programa (ou seja, podem ter efeitos colaterais) é possível que um método altere o estado do programa e, de forma não planejada, acabe corrompendo a integridade do sistema, o que muito provavelmente pode levar a um erro em tempo de execução. Esse é comum problema comum no paradigma imperativo, e pode ser contornado com o uso dos tratadores de exceção `try-catch`, presentes em linguagens imperativas mais recentes como C++, Java, C#, JavaScript, Python, etc.

No paradigma orientado a objetos, ainda pode ser usado o conceito de encapsulamento para tornar o programa mais modular, como uma caixa preta. Assim, é fácil desenvolver um módulo, testá-lo, e uma vez testado, usá-lo em outras partes do projeto com total confiabilidade, bem como em outros projetos. Dessa forma, é possível aumentar a produtividade sem perder confiabilidade.

Já no paradigma funcional o problema citado não existe, uma vez que o programa não tem estados, e todos os dados definidos são imutáveis. Propriedades como transparência referencial e ausência de efeitos colaterais tornam a programação funcional muito mais modular e fácil do seu código ser analisado e testado de forma isolada, sendo um enorme trunfo quanto à confiabilidade do programa. O mesmo também vale para programas desenvolvidos no paradigma lógico, que assim como o funcional, também tem transparência referencial e ausência de efeitos colaterais.

Por fim, programas desenvolvidos no paradigma declarativo têm clara correspondência com à lógica matemática (CHAKRAVARTY, 1997), e por conta dessa característica muitos programas podem ser provados como livre de erros, o que é de extrema utilidade em um sistema para áreas médicas e aeroespaciais, por

exemplo. Vale ressaltar também que pouca legibilidade ou pouca facilidade de escrita também tendem a gerar programas pouco confiáveis.

7.3 Eficiência

O paradigma imperativo é baseado na maquina de Turing e na arquitetura von Neumann de computadores (SHERMAN, 2008), consequentemente esses programas são naturalmente eficientes quando executados nessa arquitetura; enquanto o paradigma declarativo é totalmente diferente deste modelo, sendo programas escritos neste geralmente menos eficientes no uso de CPU e memória que programas imperativos, como C, Pascal e Java (PAULSON, 1996).

Essa perda de eficiência se dá pelo fato de não existir definição de fluxo de controle na programação declarativa, de modo que qualquer iteração deve ser feita de forma recursiva. O problema é que quando uma função é chamada de forma recursiva, a pilha é usada para armazenar todos os dados de todas as instâncias dessa função, e se a função recursiva chama a si mesmo muitas vezes, a pilha pode ficar sem memória; enquanto que, em programas imperativos, para cada iteração são usadas as mesmas células de memória, tendo essas apenas seus valores alterados a cada ciclo do loop.

Em compensação, graças a propriedades como transparência referencial, ausência de efeitos colaterais e imutabilidade dos dados, o compilador pode aplicar técnicas de otimização (e.g., memoização e eliminação de subexpressões comuns) para remover computações desnecessárias; sendo difícil de aplicar essas técnicas em programas imperativos de forma a manter a integridade dos valores, e portanto a confiabilidade do software.

Além disso, avanços tecnológicos para a construção de microprocessadores têm permitido cada vez mais que sistemas ganhem maior desempenho executando de forma paralela. Contudo, não houve grandes avanços quanto ao desenvolvimento de paradigmas e linguagens para se aproveitar esse recurso, sendo agora o ponto mais crítico no desenvolvimento de um sistema paralelo não mais a modelagem do hardware, e sim o desenvolvimento do software (GARANHANI, 2008).

Muitos pesquisadores afirmam que o paradigma imperativo é inadequado para o processamento paralelo, já que seus conceitos são baseados no modelo von Neuman de “uma operação por vez” (HUDAK, 1986), bem como o fato da execução paralela de partes com dados compartilhados poder interferir uma na outra e alterar o resultado esperado. Sendo assim, torna-se uma tarefa complexa, e sem aproveitamento pleno escrever programas em C ou Java para aproveitar os benefícios de processadores multicore. Por outro lado, programas declarativos são naturalmente paralelos, uma vez que são compostos de funções puras e a ordem de execução entre elas é irrelevante. Nesse cenário, linguagens declarativas podem facilmente ser mais eficientes que as do paradigma imperativo.

7.4 Reusabilidade

Como vimos ao longo deste trabalho, nos paradigmas procedimental, funcional e orientado a objetos podemos escrever funções e / ou subrotinas, e por meio dessas encapsular uma determinada ação ou processo do sistema, bastando apenas chamá-la quando necessário; o que nos evita ter de reescrever o código em todos os lugares onde aquela ação se faz presente, aumentando assim altamente a reusabilidade em um projeto de *software*. Uma vez que já temos várias funções escritas, ainda podemos, a partir dessas, criar bibliotecas de funções, possibilitando assim a reusabilidade de código não apenas em várias partes de um mesmo sistema, mas também entre vários sistemas. No caso do paradigma lógico a ideia de reusabilidade é similar, sendo predicados os correspondentes a funções e criação de um banco de dados a parte com os predicados que serão reutilizáveis, correspondendo às bibliotecas.

O conceito de polimorfismo paramétrico adotado pelos paradigmas funcional e orientado a objeto também colaboram consideravelmente para uma maior reusabilidade durante o desenvolvimento de um projeto, pois é possível definir funções que aceitam mais de um tipo como parâmetro, o que nos permite escrever uma função genérica e usá-la para diversos tipos, ao invés de termos que reescrever o código para todos os tipos necessários.

Ainda no paradigma funcional o conceito de funções como objetos de primeira classe e funções de ordem superior também permitem aumentar em muito a reusabilidade de código, pois graças a essas é possível abstrair sobre ações e definir funções como `map` e `filter`, que além de extremamente úteis, permitem ao o programador, por exemplo, definir um filtro genérico e receber a função principal como parâmetro, ao invés de obrigá-lo a implementar uma versão especifica para cada tipo de filtro.

Já no paradigma orientado a objetos, além do polimorfismo, podemos destacar o conceito de herança, que tem como objetivo justamente definir novas classes a partir de outras já implementadas, herdando todas as características (atributos, métodos, formas de construção, etc.) da classe mãe, evitando assim que o programador tenha reescrever toda a estrutura mais genérica para cada nova especialização.

7.5 Manutenabilidade

Como discutido nas seções anteriores deste capitulo, programas declarativos tendem a ter um código mais simples e seguro. Essas características da programação declarativa facilitam a manutenção do sistema, uma vez que o programa é altamente modular e estes não possuem dependências implícitas.

Em contraste, a manutenção em programas imperativos tende a ser uma tarefa mais difícil já que esses possuem estados mutáveis com dependências implícitas e, possivelmente, subrotinas pouco expressivas. Uma forma de minimizar essa dificuldade é fazer uso do conceito de encapsulamento, ocultando assim toda parte “sensível” do programa (e.g., controle interno de um objeto, validações da regra de negócio, etc.), tornando assim o programa mais modular e por tanto, mais fácil de testar e manter. No paradigma procedimental, que não possui explicitamente um conceito de encapsulamento, podemos modularizar o código por subrotinas e bibliotecas.

Para simplificar o código em programas imperativos e minimizar a dificuldade de manutenção, existe uma série de recomendações/boas práticas a seguir durante o desenvolvimento, como por exemplo: usar nomes significativos para variáveis,

cada função ou subrotina deve realizar apenas uma tarefa, comentar o código quando for necessário evidenciar alguma regra do negócio, etc. (MARTIN, FEATHERS e OTTINGER, 2008).

8 Conclusão

Os paradigmas de programação estudados se mostraram bastantes diferentes entre si, não apenas na forma como o programa é estruturado, mas principalmente pelo conjunto de princípios, conceitos e técnicas que determinam a maneira como o programador pensa sobre a tarefa de programar.

Uma vez que cada paradigma surgiu a partir de necessidades distintas dos demais, naturalmente estes apresentam mais vantagens que os demais para o desenvolvimento de sistemas dentro do domínio para o qual foi originalmente proposto, oferecendo conceitos, técnicas e abstrações apropriadas para esses domínios de aplicação.

As principais contribuições deste trabalho para a comunidade de programadores foi o estudo dos principais conceitos de cada paradigma bem como avaliações comparativas entre estes em diversos pontos relevantes para um projeto de software, como expressividade do código, eficiência, confiabilidade, entre outros. Contribuindo assim para que este consiga decidir qual paradigma é o mais adequado para desenvolver um projeto específico.

Possíveis extensões naturais deste trabalho seriam o estudo e comparação dos seguintes estilos de programação: programação simbólica, meta programação e programação concatenativa.

Referências Bibliográficas

CARDELLI, L.; WEGNER, P. **On Understanding Types, Data Abstraction, and Polymorphism**. [S.l.]: AT&T Bell Laboratories, Murray Hill, 1985.

CHAKRAVARTY, M. M. T. **On the Massively Parallel Execution of Declarative Programs**. [S.l.]: na, 1997.

DEITEI, H. M.; DEITEI, P. J. **C++ - Como Programar**. [S.l.]: BOOKMAN, 2001.

DEITEL, P.; DEITEL, H. **Java - Como Programar - 8ª Ed**. São Paulo: Prentice Hall, 2010.

DIJKSTRA, E. W. **A Case againstthe GO TO Statement**. Eindhoven: Technological University, 1968.

GABBRIELLI, M.; MARTINI, S. **Programming Languages - Principles and Paradigms**. Bologna: Springer, 2006.

GARANHANI, C. E. C. **Um Ambiente Paralelo Para Implementação de Modelos Adaptativos**. São Paulo: EPUSP, 2008.

HAVERBEKE, M. **Eloquent JavaScript, 2nd Ed.:** A Modern Introduction to Programming. [S.l.]: No Starch Press, 2004.

HUDAK, P. **Para-functional Programming**. City of New Haven: Yale University, 1986.

HUGHES, J. **Why Functional Programming Matters**. Glasgow: The University, Glasgow, 1990.

KUHN, T. S. **The Structure of Scientific Revolutions**. Chicago : University of Chicago Press, 1991.

LIMA, E.; REIS, E. **C# e.NET - Guia do Desenvolvedor**. [S.l.]: Editora Campus Ltda, 2002.

LIPOVAčA, M. **Learn You a Haskell for Great Good!:** A Beginner's Guide. San Francisco: No Starch Press, 2011.

LUTZ, M.; ASCHER, D. **Learning Python**. Los Angeles: O'Reilly Media, 1999.

MARTIN, R. C.; FEATHERS, M. C.; OTTINGER, T. R. **Clean Code:** A Handbook of Agile Software Craftsmanship. [S.l.]: Prentice Hall PTR, 2008.

MILLS, H. D. **Mathematical Foundations for Structured Programming**. [S.l.]: University of Tennessee, Knoxville, 1972.

PALAZZO, L. A. M. **Introdução à Programação Prolog**. Pelotas: Editora da Universidade Católica de Pelotas , 1997.

PAPERT, S. **New Images Of Programming:** In Search Of An Educationally Powerful Concept Of Technological Fluency. (A Proposal To The National Science Foundation). Cambridge: MA: MIT Technology Laboratory, 1991.

PAULSON, L. C. **ML For The Working Programmer 2nd Edition**. Cambridge: Cambridge University Press, 1996.

PEREIRA, S. D. L. **Linguagem C++**. São Paulo: FATEC, 1999.

PIERCE, B. C. **Types and Programming Languages**. Cambridge: The MIT Press, 2002.

SCOTT, M. L. **Programming Language Pragmatics**. Burlington: Morgan Kaufmann Publishers , 2009.

SEBESTA, R. W. **Conceitos de Linguagem de Programação**. Colorado: Bookman, 2006.

SHERMAN, J. W. **Compiling Imperative and Functional Languages**. Florida: New College of Florida, 2008.

TUCKER, A. B.; NOONAN, R. E. **Programming Languages - Principles and Paradigms 2nd Edition**. New York: McGraw-Hill, 2007.

WATT, D. A. **Programming Language Design Concepts**. Chichester: John Wiley & Sons Ltd., 2004.

WIRTH, N. **Go To Statement Considered Harmful**. New York: Communications of the ACM, 1987.