Tecnologia da Informação

PARADIGMAS DE PROGRAMAÇÃO: UMA INTRODUÇÃO

Autores:
Sidnei Renato Silveira
Antônio Rodrigo Delepiane de Vit
Cristiano Bertolini
Fábio José Parreira
Guilherme Bernardino da Cunha
Nara Martini Bigolin

Compartilhando conhecimento

Tecnologia da Informação

PARADIGMAS DE PROGRAMAÇÃO: UMA INTRODUÇÃO

Autores:
Sidnei Renato Silveira
Antônio Rodrigo Delepiane de Vit
Cristiano Bertolini
Fábio José Parreira
Guilherme Bernardino da Cunha
Nara Martini Bigolin

Compartilhando conhecimento

Editor Chefe
Msc Washington Moreira Cavalcanti

Autores
Sidnei Renato Silveira
Antônio Rodrigo Delepiane de Vit
Cristiano Bertolini
Fábio José Parreira
Guilherme Bernardino da Cunha
Nara Martini Bigolin

Conselho Editorial
Msc Lais Brito Cangussu
Msc Rômulo Maziero
Msc Jorge dos Santos Mariano
Dr Jean Canestri

Projeto Gráfico e Diagramação
Departamento de arte Synapse Editora

Edição de Arte
Maria Aparecida Fernandes

Revisão
Os Autores

2021 *by Synapse Editora*
Copyright © Synapse Editora
Copyright do Texto © 2021 Os autores
Copyright da Edição © 2021 Synapse Editora
Direitos para esta edição cedidos à
Synapse Editora pelos autores.

O conteúdo dos artigos e seus dados em sua forma, correção e confiabilidade são de responsabilidade exclusiva dos autores, inclusive não representam necessariamente a posição oficial da Synapse Editora.

Permitido o download da obra e o compartilhamento desde que sejam atribuídos créditos aos autores, mas sem a possibilidade de alterá-la de nenhuma forma ou utilizá-la para fins comerciais.

A Synapse Editora não se responsabiliza por eventuais mudanças ocorridas nos endereços convencionais ou eletrônicos citados nesta obra.

Todos os manuscritos foram previamente submetidos à avaliação por parte dos membros do Conselho Editorial desta Editora e pareceristas convidados, tendo sido aprovados para a publicação.

Compartilhando conhecimento
2021

S587p Silveira, Sidnei Renato

Paradigmas de programação: Uma introdução
Autores: Sidnei Renato Silveira; Antônio Rodrigo Delepiane de Vit; Cristiano Bertolini; Fábio José Parreira; Guilherme Bernardino da Cunha; Nara Martini Bigolin.
Belo Horizonte, MG: Synapse Editora, 2021, 95 p.

Formato: PDF
Modo de acesso: World Wide Web
Inclui bibliografia

ISBN: 978-65-88890-08-0
DOI: doi.org/10.36599/editpa-2021_ppui

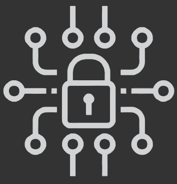
1. Programação, 2. Tecnologia da Informação, 3.Sistemas de Informação, 4. Alálise de Sistemas, 6. Desenvolvimento.

I. Paradigmas de programação: Uma introdução

CDD: 600
CDU: 60 - 681.3

SYNAPSE EDITORA
Belo Horizonte – Minas Gerais
CNPJ: 20.874.438/0001-06
Tel: + 55 31 98264-1586
www.editorasynapse.org
editorasynapse@gmail.com

Compartilhando conhecimento
2021



Apresentação

Este livro foi elaborado para servir como apoio aos processos de ensino e de aprendizagem de diferentes Paradigmas de Programação, podendo ser empregado em disciplinas de cursos superiores da área de Computação. Os conteúdos e ferramentas abordados foram selecionados de acordo com a experiência pedagógica dos autores desta obra, todos com ampla atuação na área da Educação em Informática. O livro aborda conceitos de linguagens de programação e estudos dos paradigmas lógico, funcional, orientado a eventos e concorrente, utilizando as linguagens de programação PROLOG, Scheme, Object Pascal e Java.

Compartilhando conhecimento
2021

Sumário

CAPÍTULO 01:
Conceitos de Linguagens de Programação 7

CAPÍTULO 02:
PROLOG: Programação em Lógica 40

CAPÍTULO 03:
Programação Funcional 58

CAPÍTULO 04:
Paradigma de Orientação a Eventos – *Lazarus* 67

CAPÍTULO 05:
Programação Concorrente em Java 75

Capitulo1

CONCEITOS DE LINGUAGENS DE PROGRAMAÇÃO

Este capítulo apresenta os conceitos relacionados aos diferentes paradigmas de programação, bem como aspectos envolvendo a implementação de linguagens de programação e um breve histórico das mesmas.

1.1 Paradigmas de Programação

No contexto de desenvolvimento de software, paradigma significa um modelo para estruturar e representar problemas, cuja solução deseja-se obter por meio de um programa, construído a partir de uma linguagem de programação (SEBESTA, 2018). Existem diferentes paradigmas para desenvolver programas, tais como: imperativo, lógico, funcional, orientado a eventos e orientado a objetos, entre outros. Em algumas linguagens de programação, dependendo dos recursos oferecidos, é possível programar em mais de um paradigma (por exemplo, é muito comum algumas linguagens permitirem o desenvolvimento baseado no paradigma imperativo e, também, orientado a objeto). Neste caso, dependerá do conhecimento do programador (desenvolvedor de software) a utilização dos recursos mais adequados para a solução de cada um dos problemas.

Os diferentes paradigmas também podem envolver o domínio da aplicação das linguagens de programação, tais como: desenvolvimento de sistemas para uso comercial (Sistemas de Informação), aplicações matemáticas e aprendizado de programação, entre outros.

Sebesta (2018) elenca alguns motivos para que um estudante da área de Informática estude conceitos e diferentes paradigmas de linguagens de programação:

- Capacidade intelectual influenciada pelo poder expressivo das linguagens nas quais nos comunicamos, ou seja, quanto mais linguagens conhecermos, maior será a nossa capacidade intelectual. Isso vale para as línguas nas quais nos comunicamos, por meio da linguagem natural (Português, Inglês, Italiano, Francês, etc.) e, também, por meio das diferentes linguagens de programação por meio das quais nós, desenvolvedores de software, podemos nos expressar para solucionar problemas;

- Limitar-se à linguagem natural limita a capacidade de expressar os pensamentos em termos de profundidade e abstração. A abstração nos permite, literalmente, abstrair detalhes, para que possamos nos focar na solução do problema, de forma mais genérica. À medida que vamos quebrando o problema em partes, podemos aprofundar os detalhes necessários para sua implementação. A abstração também compreende a definição de classes e métodos (no chamado POO - Paradigma Orientado a Objeto) e no uso de diferentes estruturas de dados;

- O conhecimento de uma variedade mais ampla de recursos de linguagens de programação reduz as limitações no desenvolvimento de software, ou seja, podemos encontrar diferentes formas de resolver problemas, mesmo que a linguagem de programação não ofereça os recursos necessários, de forma nativa. Por exemplo: vamos supor que precisemos utilizar uma estrutura de dados do tipo pilha em nosso programa. Entretanto, a linguagem de programação que estamos utilizando não suporta ponteiros. Podemos, então, criar uma pilha utilizando uma estrutura de dados mais simples, um vetor (ou matriz unidimensional);

- Maior conhecimento para a escolha de linguagens apropriadas: quanto mais linguagens de programação conhecemos, maior será a nossa capacidade de definir qual a linguagem mais adequada de acordo com o tipo de problema que precisamos resolver;

- Capacidade aumentada para aprender novas linguagens: quanto mais linguagens de programação conhecemos, maior será o nosso potencial para aprendermos outras linguagens de programação, pois nosso cérebro já estará preparado para buscar as características mais importantes e recursos diferenciados de cada linguagem.

1.2 Domínios de Programação

Atualmente, os computadores são usados em uma infinidade de áreas (desde o controle de usinas elétricas até à armazenagem de registros de talões de cheques). Ficaria inviável pensarmos em inúmeras atividades rotineiras sem o uso da Tecnologia da Informação: acesso à Internet, transações bancárias, compras com cartões de débito e/ou crédito, Educação a Distância, enfim, uma infinidade de tarefas diárias que necessitam do computador (e de programas, escritos em uma linguagem de programação). Devido a essa diversidade de aplicações, linguagens de programação com propósitos muito diferentes têm sido desenvolvidas, tais como linguagens específicas para o desenvolvimento de jogos. Entre os domínios de programação, destacam-se:

- Aplicações Científicas: os primeiros computadores digitais (década de 40) foram inventados para aplicações científicas: estruturas de dados simples com um grande número de operações aritméticas com ponto-flutuante. Entre as linguagens de programação para o desenvolvimento de aplicações científicas citam-se o Algol e o Fortran;

- Aplicações comerciais: o uso de computadores para aplicações comerciais iniciou-se na década de 50. As linguagens de programação destinadas ao desenvolvimento deste tipo de aplicação possuem facilidades para produzir relatórios e armazenar dados. Exemplos deste tipo de linguagem de programação são a linguagem COBOL (Common Business Oriented Language) e o Clipper (Clipper/dBase);

- Aplicações de IA (Inteligência Artificial): são aplicações computacionais caracterizadas pelo uso de computações simbólicas ao invés de numéricas. Uma lista, por exemplo, pode armazenar conhecimento sobre um determinado problema, ao invés de dados. Sendo assim, sistemas de IA processam conhecimento, não dados (LORENZI; SILVEIRA, 2011). Entre as linguagens que podem ser aplicadas no contexto da IA destacam-se LISP e PROLOG (PROgramming in LOgic);

- Programação de Sistemas: compreendem as linguagens de programação para o desenvolvimento de Sistemas Operacionais (software básico). Os Sistemas Operacionais precisam ter execução rápida e recursos de baixo nível para permitir a interface com os dispositivos externos. Entre as linguagens de programação com este propósito citam-se: PL/S, BLISS, Extended ALGOL e C. O Sistema Operacional Unix (precursor do Linux) foi desenvolvido em Linguagem C;

- Linguagens de Scripting: são linguagens utilizadas colocando-se uma lista de comandos (script) em um arquivo para serem executados. Um exemplo são os arquivos BATCH (arquivos com a extensão .BAT). No Sistema Operacional MS-DOS (Microsoft Disk Operating System) era muito com criarmos arquivos .BAT que continham diversos comandos, tais como comandos de configuração do sistema, que deviam ser executado ao ligarmos o computador. Isto era possível por meio de um arquivo denominado AUTOEXEC.BAT (um arquivo autoexecutável). Existem outros tipos de linguagens de script, tais como: Action Script (executada no Adobe Flash), Open Script (executada no Multimedia ToolBook Instructor) e a linguagem Python, entre outras;
- Linguagens de programação para propósitos especiais, tais como linguagens para a geração de relatórios, criação de jogos e simulação de sistemas. Por exemplo, a linguagem RPG (Report Program Generator), para geração de relatórios comerciais e a linguagem de simulação de sistemas GPSS (General Purpose Simulation System).

1.3 Critérios de Avaliação de uma Linguagem de Programação

Os critérios de avaliação destacados nesta seção são baseados em Sebesta (2018).

1.3.1 Legibilidade

A legibilidade é um critério de avaliação que diz respeito à facilidade com que os programas podem ser lidos e entendidos. Este entendimento é importante para a manutenção dos programas. Antes de 1970 o desenvolvimento de software era baseado na escrita de código, ou seja, o mais importante era, efetivamente, escrever os programas. Na década de 70 a codificação passou para o segundo plano, dando lugar à manutenção (a facilidade de manutenção é determinada pela legibilidade dos programas). A legibilidade pode ser avaliada por meio das seguintes características: 1) simplicidade global; 2) ortogonalidade; 3) instruções de controle; 4) tipos de dados e estruturas e 5) sintaxe (SEBESTA, 2018).

Legibilidade: Simplicidade Global

A simplicidade global significa que, uma linguagem com um grande número de componentes básicos é mais difícil de ser aprendida do que uma com poucos desses componentes. Geralmente, os desenvolvedores de software que precisam

usar uma linguagem grande tendem a aprender um subconjunto dela e ignorar seus outros recursos. Este é um exemplo do que acontece com linguagens de programação mais complexas, como é o caso de Java (SEBESTA, 2018).

Ocorrem problemas de legibilidade sempre que o autor do programa tenha aprendido um subconjunto diferente daquele com o qual o leitor está familiarizado. Isso fica claro quando dizemos que programar é uma atividade criativa, ou seja, não é algo que pode ser, a princípio, automatizado, pois cada um de nós pode criar programas diferentes, com recursos diferentes, para resolver um mesmo problema (SEBESTA, 2018).

Algumas características que podem reduzir a legibilidade são:

1) a multiplicidade de recursos e ;

2) a sobrecarga.

A multiplicidade de recursos diz respeito à existência de mais de uma maneira de realizar uma mesma operação. Por exemplo, uma operação de incrementar um contador, que pode ser feita de várias maneiras (algumas com resultados diferentes inclusive):

contador = contador + 1

contador += 1

contador++

++contador

A sobrecarga (overloading) de operador acontece quando um único símbolo tem mais de um significado. Por exemplo: o sinal de adição (+) pode ser usado para adição de números inteiros e para matrizes (arrays), como mostram os exemplos abaixo:

10+20 ´resulta 30

“10”+”20” ´resulta “1020”

Outro exemplo de sobrecarga ocorre no paradigma de Orientação a Objetos, quando descrevemos um mesmo método com diferentes assinaturas. Quando usamos a herança, podemos sobrescrever um método da classe-pai (BERTAGNOLLI, 2009).

Legibilidade: Ortogonalidade

A característica da ortogonalidade significa que um conjunto relativamente pequeno de construções primitivas pode ser combinado em um número relativamente pequeno de maneiras para construir as estruturas de controle e

de dados da linguagem. Por exemplo, supondo que uma linguagem de programação tenha 4 tipos de dados primitivos: *integer, float, double e character* e 2 operadores de tipo: *array e pointer* (SEBESTA, 2018). Se os 2 operadores de tipo puderem ser aplicados a si mesmos e aos quatro tipos de dados primitivos, um grande número de estruturas de dados poderá ser definido, tais como:

Arrays de integer, float, double e character

Pointer (ponteiros) para *integer, float, double, character e arrays*

Porém, se não for permitido aos ponteiros apontar para *arrays,* muitas dessas possibilidades seriam eliminadas. A ortogonalidade parte de uma simetria de relações entre primitivas. Os ponteiros devem ser capazes de apontar para qualquer tipo de variável ou estrutura de dados para que uma linguagem de programação seja ortogonal. A falta de ortogonalidade acarreta exceções às regras de uma linguagem de programação, fazendo com que os desenvolvedores de *software* tenham que encontrar outras formas e/ou recursos para realizar suas implementações.

Legibilidade: Instruções de Controle

As instruções de controle que envolvem desvios (os chamados "*gotos*") dificultam a legibilidade de um programa. Para aumentar a legibilidade deve-se utilizar a programação estruturada e eliminar comandos de desvio (goto). Isso pode ser feito utilizando-se sub-rotinas (procedimentos e funções) e, em paradigmas como o da Orientação a Objeto, por meio de métodos (SEBESTA, 2018).

Sebesta (2018) coloca que pode-se restringir o uso de instruções goto para tornar os programas mais legíveis, destacando-se que elas devem preceder seus alvos, exceto quando usadas para formar laços; seus alvos nunca devem estar muito distantes; e seu número dever ser limitado (ou até mesmo nulo, se possível).

Legibilidade: Tipos de Dados e Estruturas

A presença de facilidades adequadas para definir tipos de dados e estruturas de dados em uma linguagem de programação é outro auxílio significativo para a legibilidade. Algumas linguagens de programação mais modernas possuem estruturas de dados já implementadas, para que os desenvolvedores possam utilizar pilhas, filas, listas e outras estruturas. Em algumas linguagens é preciso construir métodos para implementar estas estruturas, geralmente utilizando-se ponteiros (pointers – alocação dinâmica de memória) (SEBESTA, 2018).

Legibilidade: Considerações sobre a Sintaxe

As regras de sintaxe de uma linguagem de programação também podem contribuir (ou não) para a legibilidade de um programa. Seguem alguns exemplos (SEBESTA, 2018) de regras de sintaxe que podem dificultar a legibilidade de um programa:

- Restringir os *identificadores* a tamanhos muito pequenos. Na linguagem FORTRAN 77, por exemplo, os identificadores podiam ter apenas 6 caracteres no máximo;
- *Palavras especiais ou reservadas*: a legibilidade de um programa é fortemente influenciada pelas formas das palavras especiais, tais como o método para formar instruções compostas ou grupos de instruções (ex.: *begin – end*, abrir e fechar chaves {}). Utilizar os mesmos símbolos não é adequado. O mais correto seria utilizar *end for, end if, end while*, ao invés de usar somente fechar chaves, por exemplo;
- Em algumas linguagens, como no FORTRAN 90, palavras especiais como DO e END são nomes válidos de variáveis, o que dificulta o entendimento: como saber se está sendo utilizado um identificador (variável) ou um comando da linguagem?

Ainda considerando a sintaxe, destacam-se as características que envolvem a forma e o significado. Neste sentido, os desenvolvedores de linguagens de programação devem projetar instruções a fim de que sua aparência indique, pelo menos parcialmente, ou seja, sua finalidade é um auxílio para a legibilidade. A semântica (significado) deve seguir diretamente da sintaxe ou da forma.

Esse princípio é violado por duas construções de linguagem idênticas ou similares quanto à aparência, mas com significados diferentes, dependendo, por exemplo, do contexto.

1.3.2 Capacidade de Escrita (Writability)

A capacidade de escrita, ou *writability*, é a medida de quão facilmente uma linguagem pode ser utilizada para criar programas para um domínio de problema escolhido. A maioria das características da linguagem que afeta a legibilidade também afeta a capacidade de escrita (escrever um programa exige uma releitura frequente da parte que já foi escrita pelo programador) e é essencial para a manutenção de sistemas (SEBESTA, 2018).

A capacidade de escrita pode ser considerada com base nas características de 1) simplicidade e ortogonalidade; 2) suporte para abstração; e 3) expressividade (SEBESTA, 2018).

Simplicidade e Ortogonalidade

Caso uma linguagem de programação tenha um grande número de diferentes construções, alguns programadores podem não estar familiarizados com todas elas. Isso pode levar ao uso inadequado de alguns recursos e ao desuso de outros que podem ser mais elegantes ou mais eficientes. Um número menor de construções primitivas e um conjunto consistente de regras para combiná-las (ortogonalidade) é muito melhor do que existir um grande número de primitivas (SEBESTA, 2018).

Apesar disso, uma ortogonalidade demasiada pode resultar em prejuízo para a capacidade de escrita, pois erros ao escrever programas podem não ser detectados, uma vez que quase todas as combinações de primitivas são permitidas. É preciso existir um equilíbrio.

Suporte para Abstração

A abstração é a capacidade de definir e, depois, de usar estruturas ou operações complexas de uma maneira que permita ignorar muitos dos detalhes de implementação. Isto é possível, por exemplo, por meio da implementação de métodos. Se criarmos métodos para manipular uma estrutura de dados do tipo pilha (*inserir dados na pilha – empilhar, retirar dados da pilha – desempilhar e assim por diante*) outros desenvolvedores poderão, posteriormente, manipular uma pilha apenas chamando estes métodos, sem necessidade de conhecer os detalhes da nossa implementação (SEBESTA, 2018).

O grau de abstração permitido por uma linguagem de programação é muito importante para a sua capacidade de escrita (*writability*). As linguagens de programação podem suportar duas categorias distintas de abstração: processo e dados. Um exemplo de abstração de processo é o uso de subprogramas (métodos) para que o mesmo código não precise ser replicado muitas vezes. Basta chamar o método para executá-lo quantas vezes for necessário.

A abstração de dados envolve a aplicação de diferentes estruturas de dados e formas de manipulá-las em uma linguagem de programação (por exemplo, diferentes formas para manipular uma árvore binária).

Expressividade

A *expressividade* é a capacidade de que operadores muito poderosos permitam que uma grande quantidade de computação seja realizada com um programa muito pequeno. Por exemplo:

contador = contador +1

contador++ ´o operador ++ faz a mesma operação do que as instruções anteriores

Um exemplo de expressividade, utilizando a linguagem *Python,* é o operador de atribuição (sinal de igualdade =). Podemos realizar atribuições múltiplas em *Python,* tais como:

x, y = 10, 20 # nesta mesma linha, a variável x está recebendo o valor 10 e a variável y o valor 20, respectivamente

Confiabilidade

Um programa é considerado confiável se ele se comportar de acordo com as suas especificações sob todas as condições. Isto pode ser verificado por meio de inúmeros testes que devem ser realizados antes do software ser disponibilizado aos usuários. A confiabilidade pode ser medida por meio de 1) verificação de tipos; 2) manipulação de exceções; e 3) *aliasing* (SEBESTA, 2018).

Confiabilidade: verificação de tipos

A verificação de tipos pode ser realizada por meio de testes, para identificar se existem erros de tipo em um programa, por meio do compilador ou durante a execução do programa. A verificação em tempo de compilação é desejável pois, quanto mais cedo forem detectados erros em um programa, menos dispendioso serão as suas correções. Um exemplo de erro bem comum é quando declaramos uma variável do tipo inteiro e, o usuário ao entrar com os dados, digita um caracter alfabético. Este é um erro de semântica, pois a variável (identificar) declarado não permite o armazenamento de caracteres que não sejam numéricos.

Confiabilidade: Manipulação de Exceções

A manipulação de exceções (*exceptions*) é a capacidade de um programa interceptar erros em tempo de execução, pôr em prática medidas corretivas e, depois, prosseguir com a execução. Esta possibilidade é um grande auxílio para a confiabilidade dos programas. Um exemplo de manipulação de exceções é o bloco *Try...Catch* da linguagem de programação *Java.*

Confiabilidade: Aliasing

O conceito de *aliasing* é o de existirem dois ou mais métodos, ou nomes, distintos para fazer referência à mesma célula da memória. Isto se configura como um recurso perigoso em uma linguagem de programação. Por exemplo, se

criarmos um método que receberá um parâmetro com passagem por referência (apontando para a mesma área de memória), estaremos modificando o valor desta área de memória dentro do método – este é um problema e um exemplo de *aliasing*.

Em algumas linguagens o *aliasing* é usado para superar deficiências nas facilidades de abstração de dados. Entretanto, algumas linguagens o restringem muito, para aumentarem sua confiabilidade.

1.3.3 Custo

A última característica a ser avaliada é o custo de uma linguagem de programação. O custo final de uma linguagem de programação é uma função de muitas de suas características (SEBESTA, 2018):

- custo para treinar desenvolvedores, compreendendo as características ligadas à simplicidade e à ortogonalidade;
- custo para escrever (*writability*) programas na linguagem;
- custo para compilar programas na linguagem;
- custo para executar programas (influenciado pelo projeto da linguagem);
- custo do sistema de implementação da linguagem;
- custo da má confiabilidade;
- custo da manutenção de programas.

1.4 Influências sobre o Projeto de uma Linguagem de Programação

Ao projetar uma linguagem de programação o desenvolvedor (ou uma equipe de desenvolvedores) deve levar em consideração aspectos ligados à 1) arquitetura do computador e 2) metodologia de programação (SEBESTA, 2018).

A arquitetura dos computadores exerceu um efeito crucial sobre o projeto das linguagens de programação, em especial a arquitetura de *von Neumann*. Nesta arquitetura, dados e programas são armazenados na mesma memória e as variáveis (identificadores) são os recursos centrais dos programas. As linguagens de programação que seguem esta arquitetura, na sua grande maioria, são chamadas de imperativas, tais como *C*, *COBOL, Pascal* e *Clipper*, entre outras.

Quanto às metodologias de programação, pode-se implementar linguagens que usam comandos de desvio (*goto*), baseadas em programação estruturada (projeto *top-down*) ou baseadas no Paradigma de Orientação a Objetos.

A seguir temos um exemplo de um trecho de código-fonte escrito em linguagem BASIC, utilizando comandos de desvio (*goto*). Para utilizar comandos de desvio precisamos que as linhas do código-fonte sejam numeradas (para podermos indicar para qual linha a execução deve ser desviada) ou que o trecho do código seja nomeado com um rótulo (*label*). No exemplo temos o uso do comando *GOTO* nas linhas 30 e 70. Caso o usuário não pressione a tecla S ele receberá a mensagem *Pressione a tecla S* e o programa voltará à execução para a linha 10. Caso o usuário *pressione a tecla S* o programa encerrará a execução (*END* na linha 50).

```
10 PRINT “Pressione uma tecla:”
20 TECLA$=INPUT$(1)
30 IF TECLA$<>”S” AND TECLA$<>”s” THEN GOTO 60
50 END
60 PRINT “Pressione a tecla S”
70 GOTO 10
```

1.5 Categorias de Linguagens de Programação

Existem quatro categorias principais de linguagens de programação, compreendendo, então, quatro paradigmas principais: 1) imperativas, 2) funcionais, 3) lógicas e 4) orientadas a objeto. Existem outras categorias, tais como as linguagens orientadas a eventos. Algumas vezes uma linguagem de programação é baseada em um dos quatro paradigmas principais e possui recursos de outro paradigma. Por exemplo, Java é uma linguagem de programação orientada a objeto mas, quando utilizamos componentes da interface gráfica (tais com janelas e botões), também será preciso desenvolver código de acordo com a ocorrência de eventos (paradigma de orientação a eventos), tais como clique com o *mouse* sobre um botão.

Seguem alguns exemplos de linguagens de programação de acordo com as principais categorias:

1) Imperativas: *C, Pascal, COBOL, Clipper;*
2) Funcionais: LISP, *Scheme;*
3) Lógicas: PROLOG;
4) Orientadas a Objeto: *C#, Java, SmallTalk.*

1.6 Métodos de Implementação de Linguagens de Programação

A linguagem de máquina do computador é seu conjunto de macroinstruções. A sua própria linguagem de máquina é a única que a maioria dos computadores compreende para executar as instruções de um programa (SEBESTA, 2018). Para que um desenvolvedor possa criar programas em uma linguagem de programação de alto nível (mais próxima da nossa linguagem natural), é preciso converter o código-fonte em um código que possa ser executado e entendido pelo computador. Essa conversão pode ser feita por um programa compilador, por exemplo.

Um computador poderia ser projetado e construído com uma linguagem de alto nível particular como sua linguagem de máquina, mas seria complexo, caro e inflexível. Seria muito difícil utilizar outras linguagens de alto nível neste mesmo computador. Antigamente, muitos computadores eram desenvolvidos desta forma e só podiam ser programados com uma única linguagem (como se fosse uma linguagem proprietária). Atualmente isso seria inviável, devido ao alto custo e dificuldades de aprendizado de inúmeras linguagens de programação.

A opção de projeto de máquina mais prática implementa, em hardware, uma linguagem de nível muito mais baixo, que oferece as operações primitivas mais comumente necessárias e exige que o *software* de sistema (por exemplo, o Sistema Operacional) crie uma interface com os programas de nível mais elevado, tais como as IDEs (*Integrated Development Environment*), que são ambientes integrados para desenvolvimento de software (por exemplo, o *NetBeans* e o *Visual Studio*).

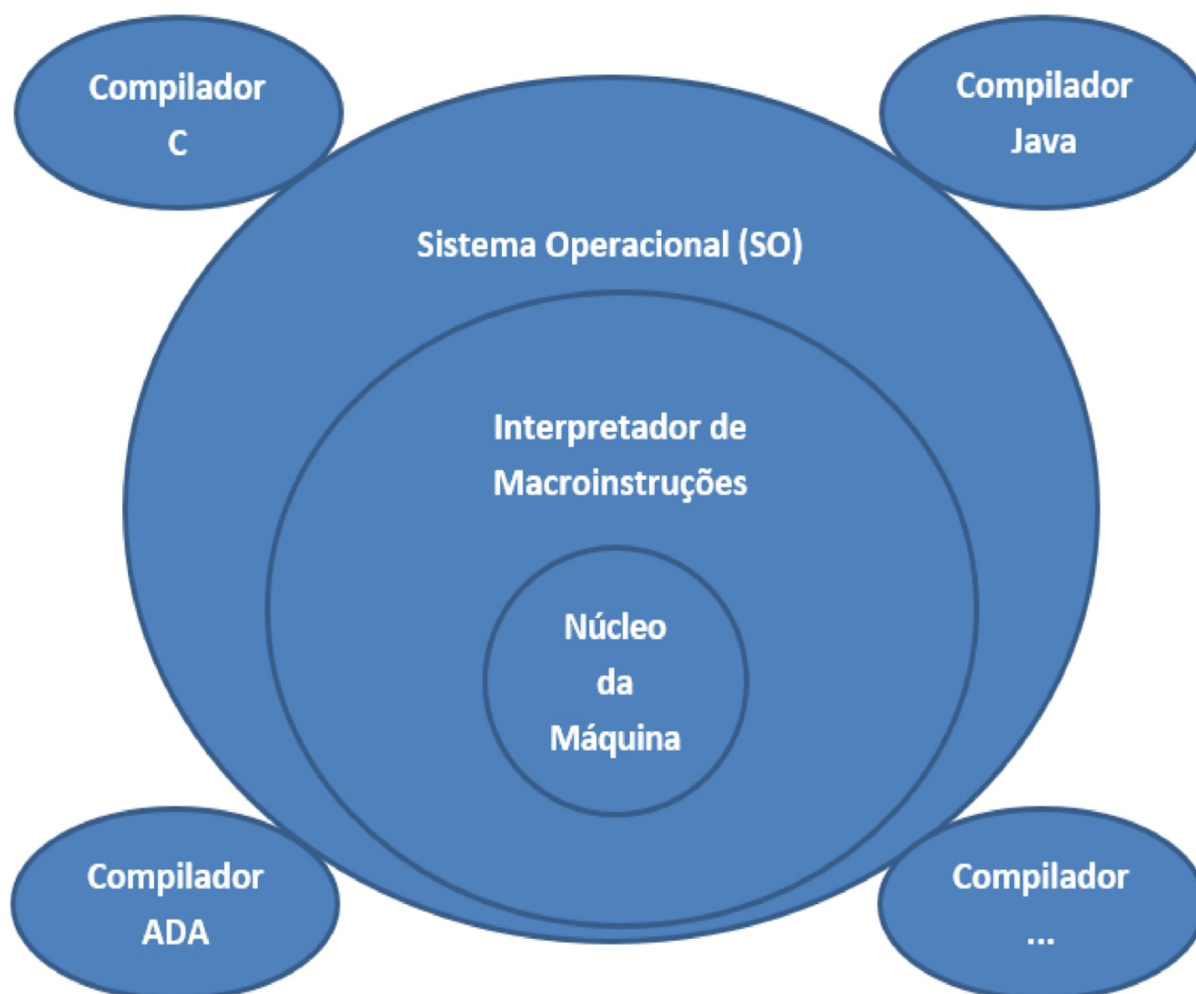
Um sistema de implementação de uma linguagem de programação não pode ser o único software em um computador. Faz-se necessário um grande conjunto de programas, que compõem o Sistema Operacional, que fornece primitivas de mais alto nível do que as de linguagem de máquina, facilitando o acesso aos recursos do computador (SEBESTA, 2018).

As primitivas de um Sistema Operacional (SO) oferecem gerenciamento de recursos do sistema, operações de entrada e saída, um sistema de gerenciamento de arquivos, editores de texto, entre outros recursos necessários para que um programa possa ser executado, interagindo com o usuário e com os dispositivos de *hardware*.

Como os sistemas de implementação de linguagens de programação necessitam de muitas das facilidades do SO, eles comunicam-se com o SO ao invés de diretamente com o processador (em linguagem de máquina).

O SO e as implementações são dispostos em camadas sobre a interface da linguagem de máquina de um computador. Essas camadas podem ser abstraídas como computadores virtuais, que oferecem interfaces de alto nível para os usuários, neste caso, os desenvolvedores de *software.* A Figura 1 apresenta esta abstração de camadas.

Figura 1: Abstração das Camadas de Interface de um Computador



Fonte: Adaptada de SEBESTA (2018)

1.6.1 Compilação

A implementação de uma linguagem de programação *compilada* significa que os programas podem ser traduzidos para linguagem de máquina, que pode ser executada diretamente no computador. Este método tem a vantagem de uma execução de programa muito rápida, pois é gerado um arquivo *executável*

durante o processo de compilação. O processo de compilação é dividido em diferentes fases: 1) análise léxica; 2) análise sintática; 3) geração do código intermediário e análise semântica (SEBESTA, 2018):

- O analisador léxico reúne os caracteres do código-fonte em unidades léxicas que são os identificadores, as palavras especiais, os operadores e os símbolos de pontuação. Os comentários são ignorados;
- O analisador sintático recebe, como entrada, as unidades do analisador léxico (*tokens*) e usa-as para construir estruturas hierárquicas chamadas árvores de análise (*parse trees*), as quais representam a estrutura sintática de um programa. A análise sintática verifica se o código-fonte foi construído de forma a respeitar as regras de sintaxe da linguagem de programação;
- O gerador de código intermediário produz um programa em uma linguagem diferente, no nível intermediário entre o código-fonte e a saída final do compilador, que é um programa em linguagem de máquina. As linguagens intermediárias se parecem com as linguagens *Assembly*. O analisador semântico faz parte do gerador de código intermediário e verifica se há erros difíceis de serem detectados durante a análise sintática, tais como erros de tipo.

A primeira fase da análise, a análise léxica, divide o código-fonte em *tokens*, a partir de categorias, tais como *identificadores e palavras reservadas*. Por exemplo, supondo que o analisador léxico irá decompor a seguinte instrução (escrita em pseudocódigo):

*calculo = 5 * quadrado + 10*

Lexemas	*Tokens*
calculo	identificador
=	sinal de igual
5	literal inteiro
*	operador de multiplicação
quadrado	identificador
+	operador de adição
10	literal inteiro

Esta lista de *tokens* será a entrada para o analisador sintático que, por meio de árvores de análise, verificará se as regras de sintaxe estão sendo cumpridas. Antigamente era preciso digitar todo o código-fonte e submetê-lo ao processo de compilação para verificar se existiam ou não erros de sintaxe. Atualmente, as IDEs são mais *inteligentes* e, à medida que o desenvolvedor de *software* vai

digitando o código-fonte, já vão sendo mostrados possíveis erros de sintaxe e/ou mensagens de aviso (*warnings*), tais como variáveis que são declaradas e não são efetivamente utilizadas no programa.

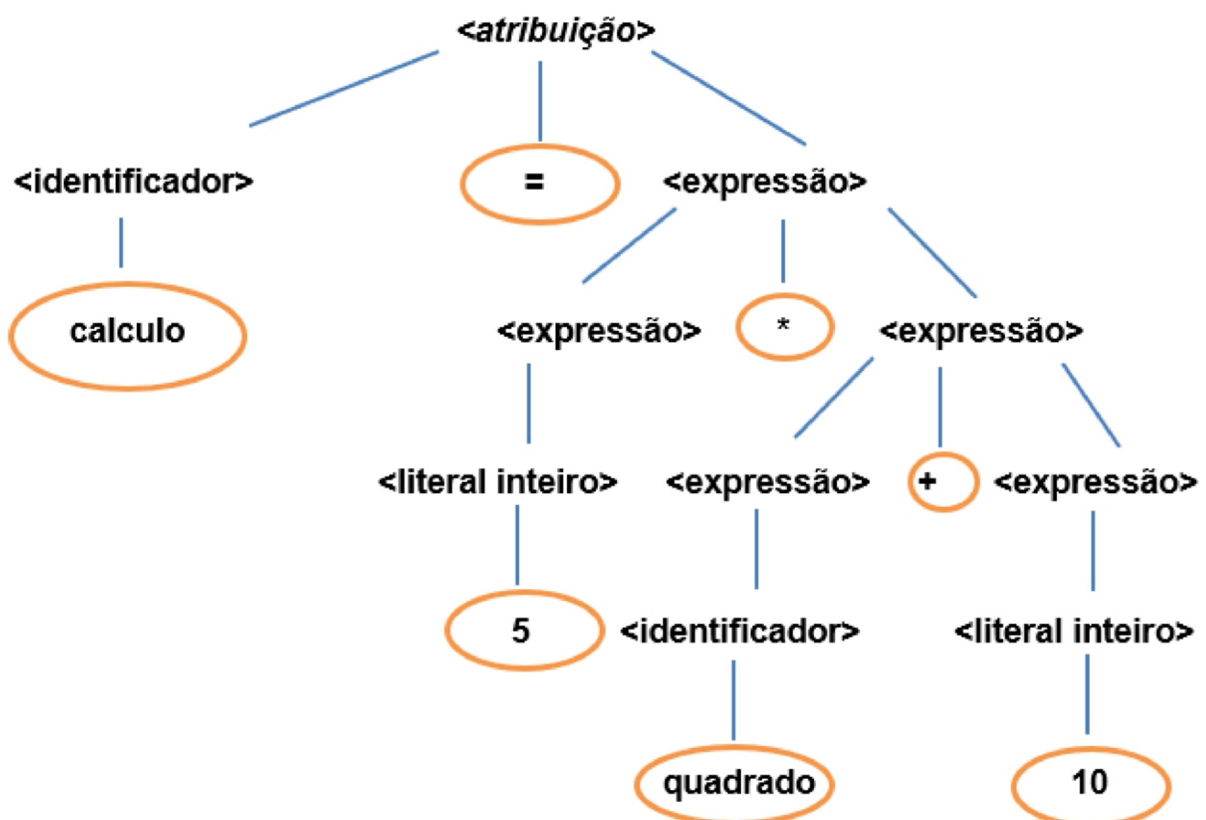
A segunda fase é a análise sintática, por meio de árvores de análise sintática ou *parse trees*. Supondo que vamos criar uma árvore da mesma instrução utilizada no exemplo da análise léxica: *calculo = 5 * quadrado + 10*. Podemos, de acordo com as regras de sintaxe da linguagem de programação, abstrairmos esta instrução da seguinte forma:

<atribuição> <- <identificador> = <expressão>

Esta abstração significa que, um comando de atribuição, segundo as regras de sintaxe de uma linguagem de programação, pode ser representado por um *identificador* (variável criada pelo programador) *recebendo o valor de um expressão aritmética*.

A árvore de análise, para este comando, poderia ser construída como mostra a Figura 2.

Figura 2: Exemplo de Árvore de Análise Sintática



Fonte: Elaborado pelos autores

A verificação da sintaxe pode ser realizada a partir da *leitura* dos nodos (nós) folha da árvore (destacados com uma elipse ao seu redor): *calculo = 5 * quadrado +10*.

A geração de código intermediário é a criação de um programa em linguagem semelhante à linguagem *Assembly*, que utiliza *mnemônicos* para acessar os registradores do processador (tais como AX, BX, CX). A Figura 3 apresenta exemplos de instruções escritas em *Assembly*.

Figura 3: Exemplo de Instruções em *Assembly*

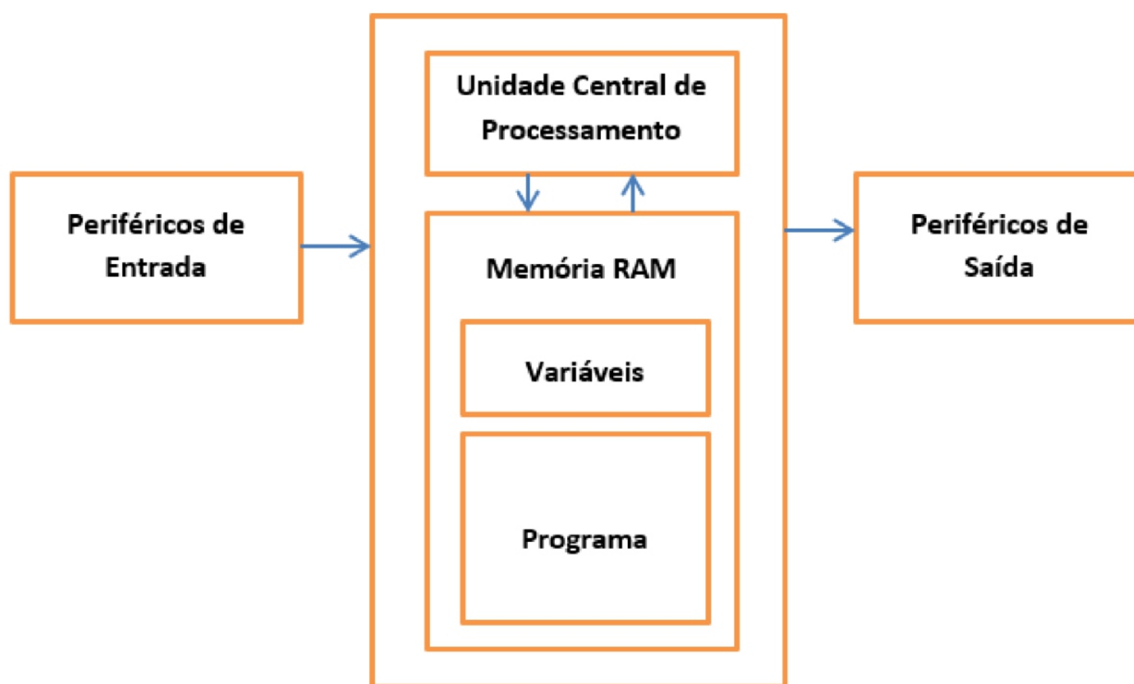
```
MOV CX, 0
MOV DX, AX
INC CX
MUL CX
CMP CX, 10
```

Fonte: Elaborado pelos autores

Ao final do processo de compilação temos o código executável de um programa, em linguagem de máquina. Apesar de a linguagem de máquina gerada por um compilador possa ser executada diretamente no *hardware,* quase sempre ela deve ser executada com algum outro código, tais como programas do Sistema Operacional, que permitem o acesso a diferentes recursos da máquina. Para isso é necessário vincular (ou ligar) os programas de usuários aos programas de sistema e, também, a bibliotecas, por meio de um *linkeditor* (SEBESTA, 2018).

A execução de um programa em código de máquina (arquivo executável) em um computador com arquitetura de *von Neumann* ocorre em um processo chamado **ciclo buscar-executar**. Os programas residem na memória volátil (ou memória *RAM – Random Access Memory*) mas são executados na CPU (Central *Process Unit* ou *UCP* – Unidade Central de Processamento). Cada instrução a ser executada precisa ser transferida da memória para o processador. O endereço da instrução seguinte a ser executada é mantido em um registro chamado *contador de programa* (SEBESTA, 2018). A Figura 4 apresenta uma abstração da arquitetura de *von Neumann*.

Figura 4: Abstração da Arquitetura de *von Neumann*



Fonte: Elaborado pelos autores

De acordo com a Figura 4, temos:

- Periféricos de entrada: que permitem a interação do usuário com o programa que está sendo executado;
- Periféricos de saída: que permitem que o usuário receba os resultados (respostas) do programa em execução;
- Unidade Central de Processamento: que faz a execução das instruções do programa, buscando, na memória, cada linha do programa que deve ser executado e, também, os valores armazenados nas variáveis. O fluxo entre a UCP e a memória é de mão dupla, pois as instruções são lidas na memória, enviadas para a UCP e os resultados são devolvidos para a memória.

Por exemplo, supondo a execução da instrução *calculo = 5 * quadrado = 10*:

1. a UCP, por meio do contador de programa, irá buscar esta instrução na memória;
2. a UCP irá buscar o valor armazenado na variável quadrado;
3. a UCP irá realizar o cálculo, primeiro da multiplicação (pela ordem de precedência), 5 * quadrado e, depois, irá somar este resultado ao número inteiro 10;
4. a UCP irá devolver o resultado para a memória, armazenando-a na variável *cálculo*.

1.6.2 Interpretação Pura

Na extremidade oposta à compilação, os programas podem ser interpretados por outro programa chamado *interpretador*, sem nenhuma conversão. Exemplos de linguagens interpretadas são o *dBase* (precursor da linguagem de programação *Clipper*) e a linguagem PROLOG, entre outras (SEBESTA, 2018).

O programa interpretador funciona como um simulador de *software* de uma máquina cujo ciclo buscar-executar lida com instruções de programa em linguagem de alto nível ao invés de instruções em código de máquina. Essa simulação de *software* fornece uma máquina virtual para a linguagem. Como vantagem ao processo de compilação, ao ocorrer um erro de execução, pode-se apontar a linha do código fonte onde tal erro ocorreu (SEBESTA, 2018).

Como desvantagem, a interpretação pode ser de 10 a 100 vezes mais lenta do que a compilação. A principal causa dessa lentidão é a decodificação mais lenta das instruções em linguagem de alto nível, bem mais complexas do que as instruções em linguagem de máquina. Além disso, a interpretação também exige mais espaço de memória, pois, além do código-fonte, a tabela de símbolos deve estar presente na interpretação, para que o interpretador faça a análise sintática, instrução a instrução (SEBESTA, 2018).

1.6.3 Sistemas de Implementação Híbridos

Alguns sistemas de implementação de linguagens são um meio-termo entre os compiladores e os interpretadores. Estes sistemas convertem programas em linguagem de alto nível para uma linguagem intermediária projetada para permitir uma fácil interpretação. Ao invés de traduzir o código fonte em uma linguagem intermediária (tal como a linguagem *Assembly*), este tipo de sistema interpreta o código intermediário, ou seja, a entrada para o sistema híbrido é o código intermediário (SEBESTA, 2018).

A linguagem de programação *Per*l utiliza este tipo de sistema. Além disso, as implementações iniciais da linguagem de programação *Java* eram todas híbridas e, para executar um programa, era necessário contar com um *software* denominado máquina virtual *Java* (SEBESTA, 2018).

1.7 Considerações sobre Sintaxe e Semântica

O estudo das linguagens de programação pode ser dividido em sintaxe e semântica. Sintaxe significa que uma instrução (ou comando) da linguagem está construída corretamente, de acordo com as regras da respectiva linguagem. A

semântica tem relação com o sentido, ou seja, se aquela instrução faz sentido. Por exemplo, em linguagem natural, se escrevermos: *Apague a luz*. Esta frase está escrita corretamente (sintaxe) e faz sentido (semântica). E se escrevêssemos: *Apague a porta?* A sintaxe continua correta mas o sentido não (a não ser que estejamos desenhando uma casa e queremos, literalmente, apagar a porta). Em programação de computadores isto também acontece. Por exemplo:

a = 10 + 20

Esta instrução está escrita corretamente, do ponto de vista da sintaxe e da semântica. A variável a irá receber a soma dos valores 10 e 20, resultando no valor 30. E se escrevêssemos a seguinte instrução:

a = 10 + "a"

A sintaxe desta instrução está adequada, ou seja, uma variável recebe o resultado de uma expressão aritmética (de uma soma). O problema é que não faz sentido (semântica), somar o número inteiro 10 com a letra (caracter alfabético e não uma variável) a. Em algumas linguagens de programação, como a linguagem PHP, o resultado de uma expressão destas seria "10a" mas, na maioria das linguagens, esta instrução contém um erro de semântica.

A sintaxe de uma linguagem de programação é a forma de suas expressões, de suas instruções e de suas unidades de programa. A semântica é o significado destas três características (SEBESTA, 2018). Por exemplo, vamos analisar a sintaxe de uma instrução *if* (seleção simples) de acordo com as regras da linguagem C:

if (<expressão>) <instrução>

As informações destacadas entre os sinais de maior e menor <> indicam que, nestes locais, será introduzido o código-fonte criado pelo desenvolvedor. Em <*expressão*> será colocada uma expressão (lógica ou relacional), que será validada. Caso a expressão resulte verdadeiro (*true*), a <*instrução*> logo após o *if* será executado. Esta é a semântica, o significado deste comando: se o valor atual da <*expressão*> for verdadeiro, a <*instrução*> será selecionada para execução.

A sintaxe e a semântica estão estreitamente relacionadas, ou seja, em uma linguagem de programação bem projetada, a semântica deve seguir-se diretamente da sintaxe. Isto quer dizer que a forma de uma instrução deve sugerir fortemente o que esta pretende realizar. Descrever a sintaxe é mais fácil do que a semântica, em parte porque uma notação concisa e aceita universalmente está disponível para a descrição da sintaxe, mas nenhuma foi

desenvolvida ainda para a semântica (SEBESTA, 2018). A semântica (o sentido) é a parte criativa da programação, ou seja, diferentes programadores podem, utilizando as regras de sintaxe, construírem programas diferentes que solucionam um mesmo problema.

As linguagens, sejam naturais, tais como o Português, ou linguagens de programação, como a linguagem *Java,* são conjuntos de sequências de caracteres de algum alfabeto. As sequências são chamadas de sentenças ou instruções. As regras de sintaxe especificam quais sequências de caracteres do alfabeto da linguagem podem ser utilizadas no desenvolvimento de programas na respectiva linguagem (SEBESTA, 2018).

As descrições formais das linguagens de programação, em nome da simplicidade, não incluem descrições das unidades sintáticas de nível mais baixo (identificadores, literais, operadores e palavras especiais).

Em 1959 foi criado um documento que introduziu uma nova notação formal para especificar a sintaxe das linguagens de programação, conhecido como Forma de *Backus-Naur* ou simplesmente BNF, a partir de estudos anteriores dos pesquisadores *John Backus* (ALGOL 58) e *Peter Naur* (ALGOL 60). Essa notação é uma metalinguagem, ou seja, uma linguagem usada para descrever outra linguagem. Assim, a BNF é uma metalinguagem para as linguagens de programação (SEBESTA, 2018).

A BNF usa abstrações para estruturas sintáticas, representadas entre os sinais de maior e menor <>. Uma simples instrução de atribuição poderia ser representada pela abstração <*atribuição*>. No entanto, a definição real de <*atribuição*> poderia ser dada por:

<atribuição> → <var> = <expressão>

onde, o símbolo à esquerda da seta (LHS – *left-hand side*) é a abstração que está sendo definida e o texto à direita (RHS) é a definição do LHS, consistido em uma mistura de *token*s, lexemas e de referências a outras abstrações. O exemplo significa que a abstração <*atribuição*> deriva em uma abstração <*var*> que recebe uma abstração <*expressão*>. Essa é uma definição da BNF, também chamada de regra ou produção (SEBESTA, 2018).

As abstrações em uma descrição BNF são chamadas símbolos não-terminais (nodos que precisam ser expandidos em uma árvore sintática) e os *tokens* e os lexemas são chamados de símbolos terminais (nodos folha de uma árvore sintática).

Os símbolos não-terminais podem ter mais de uma definição, representando diferentes possibilidades de escrita daquela instrução em uma determinada

linguagem de programação. Por exemplo, a instrução *if* (ou *comando_if*) pode ser construído de diferentes formas:

> *<comando_if> → if <expressão_lógica> then <comando>*
>
> *<comando_if> → if <expressão_lógica> then <comando> else <comando>*

Estas diferentes possibilidades podem ser representadas pelo símbolo | (que representa um ou):

> *<commando_if> → if <expressão_lógica> then <comando>*
>
> *| if <expressão_lógica> then <comando> else <comando>*

Para descrevermos uma lista de itens usando a BNF, ao invés de usarmos a forma Matemática, onde listas de tamanho variável são escritas usando-se três pontos: 1, 2, …, devemos utilizar a recursão (ou recursividade). Uma regra é recursiva se o LHS aparecer em seu RHS, como no exemplo abaixo, onde a abstração <*lista_de_identificadores*> aparece no RHS, remetendo ao LHS.

> *<lista_de_identificadores> → identificador*
>
> *| identificador, <lista_de_identificadores>*

Vamos utilizar um exemplo de uma BNF com um número reduzido de regras, para compreendermos o processo de derivação de um pequeno programa. A BNF tem 5 regras (5 abstrações): <*programa*>, <*lista_de_comandos*>, <*comando*>, <*variável*> e <*expressão*>.

> *<programa> → início <lista_de_comandos> fim*
>
> *<lista_de_comandos> → <comando>*
>
> *| <comando>; <lista_de_comandos>*
>
> *<comando> → <variável> := <expressão>*
>
> *<variável> → A | B | C*
>
> *<expressão> → <variável> + <variável>*
>
> *| <variável> - <variável>*
>
> *| <variável>*

Analisando-se esta pequena gramática, de uma linguagem de programação hipotética, temos que um programa (representado pela abstração <*programa*>) é uma lista de comandos, precedida pela palavra reservada início e terminada pela palavra reservada fim:

> *<programa> → início <lista_de_comandos> fim*

Uma lista de comandos, representada pela abstração <*lista_de_comandos*> pode ser apenas uma instrução (abstração <*comando*>) ou uma lista de diferentes comandos que existem nesta linguagem hipotética, separados por um ponto-e-vírgula (;).

<lista_de_comandos> → <comando>
| <comando>; <lista_de_comandos>

Esta linguagem tem um único comando, que é o comando de atribuição:

<comando> → <variável> := <expressão>

Além disso, por ser uma linguagem bastante enxuta, só podem ser usadas as variáveis pré-determinadas (pela abstração <*variável*>) A, B ou C.

<variável> → A | B | C

As expressões permitidas (por meio da abstração <*expressão*>) podem ser a soma de duas variáveis, a subtração de duas variáveis ou atribuir o valor de uma variável à outra. Estas expressões são definidas pela abstração <*expressão*>.

<expressão> → <variável> + <var>
| <variável> - <variável>
| <variável>

Conhecendo a gramática da linguagem, é possível saber o que podemos escrever com a mesma, ou seja, quais programas são permitidos. Por exemplo, seguindo a gramática anterior é possível escrevermos a sentença A:=A+B?

E a sentença A:=A*C? Esta sentença é permitida?

Para validarmos estas sentenças, ou seja, verificarmos se estão escritas corretamente de acordo com a sintaxe da linguagem, podemos criar as árvores de análise sintaticamente correspondentes (ou *parse trees*), como vimos, anteriormente, na seção 1.6.1.

1.7.1Parse Trees

Vamos fazer, então, o processo de derivação de um programa, seguindo a gramática da linguagem hipotética da seção 1.7, para verificarmos se é possível escrevemos o seguinte programa:

```
programa
inicio
  A:=A+B;
fim
```

O processo de derivação começa com o símbolo de iniciar <*programa*>, e o símbolo => representa "deriva". Cada abstração deve ser derivada sequencialmente, substituindo um dos não-terminais (símbolos entre os sinais de maior e menor <>) por uma definição do mesmo, constante da gramática da linguagem. Nessa definição o não-terminal substituído sempre é o da extrema esquerda na forma sentencial anterior (derivação à esquerda). A derivação prossegue até que a forma sentencial não contenha nenhum símbolo não-terminal. Vejamos o exemplo:

<programa>=> início <lista_de_comandos> fim
=> início <comando> fim
=> início <variável> := <expressão> fim
=> início A := <expressão> fim
=> início A:= <variável> + <variável> fim
=> início A:= A + <variável> fim
=> início A:= A + B fim

Vamos analisar o processo de derivação passo-a-passo. Começamos pela abstração <*programa*>, que é uma lista de comandos, que começam com a palavra reservada início e terminam com a palavra reservada *fim*:

<programa> => início <lista_de_comandos> fim

O próximo passo da derivação é resolver a abstração <*lista_de_comandos*>. Como queremos apenas um comando, pois o nosso programa de exemplo só tem uma atribuição, vamos derivar <*lista_de_comandos*> como <*comando*>:

=> início <comando> fim

A próxima derivação é escolher um comando permitido pela linguagem. Na nossa linguagem hipotética, o único comando existente é o de atribuição (atribuir uma expressão a uma variável):

=> início <variável> := <expressão> fim

Seguindo o processo de derivação, devemos derivar a primeira abstração <*variável*> e, pelo nosso programa de exemplo, precisamos derivar novamente em A:

=> início A := <expressão> fim

Agora precisamos derivar, primeiro, a abstração mais à esquerda, que é <expressão> e verificarmos se existe a expressão que queremos utilizar (no caso a soma de duas variáveis):

=> início A:= <variável> + <variável> fim

Ainda seguindo o processo de derivação, devemos derivar a primeira abstração <*variável*> e, pelo nosso programa de exemplo, precisamos derivar novamente em A:

=> início A:= A + <variável> fim

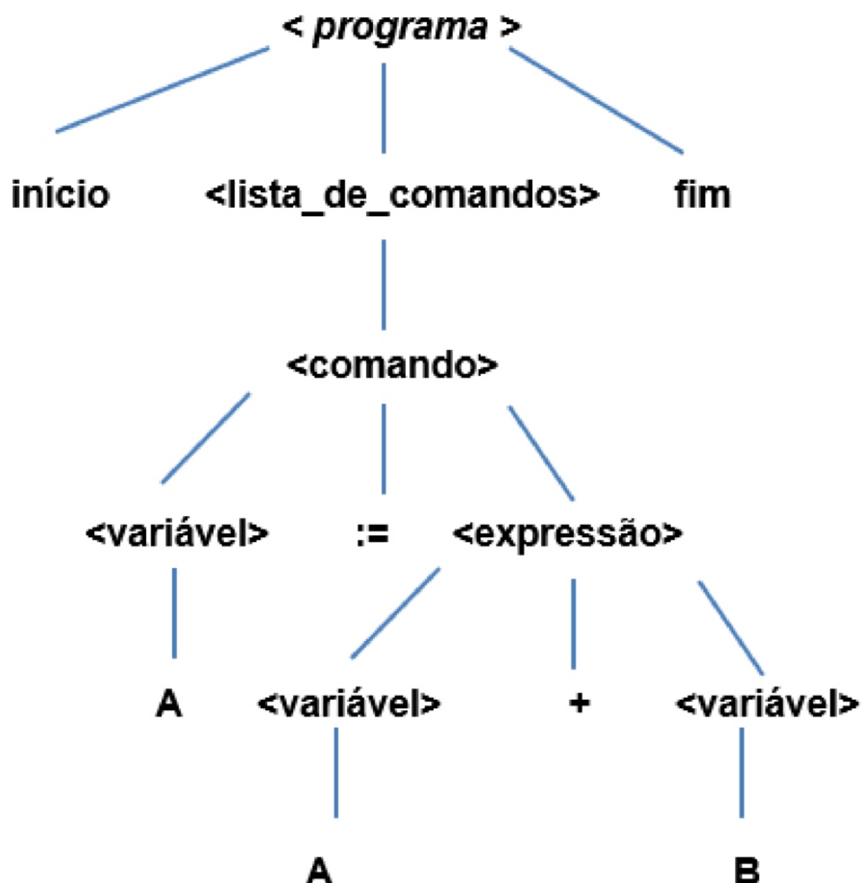
Agora vamos derivar a segunda abstração <*variável*> e, pelo nosso programa de exemplo, precisamos derivar em B:

=> início A:= A + B fim

Como não temos mais nenhuma abstração (nenhum símbolo não-terminal), significa que conseguimos fazer a derivação do nosso programa de exemplo, comprovando que é possível escrevê-lo com a gramática desta linguagem hipotética. E como ficaria a árvore de análise deste programa? Vejamos na Figura 5.

Segundo Sebesta (2018), a gramática de uma linguagem descreve a estrutura sintática hierárquica da linguagem que define, sendo que todo vértice interno de uma árvore de análise é um símbolo não-terminal, toda folha é um símbolo terminal e toda árvore secundária de uma análise descreve uma instância de uma abstração na instrução.

Figura 5: Exemplo de Árvore de Análise Sintática



Fonte: Elaborado pelos autores

1.8 Ambientes de Programação

Um ambiente de programação ou IDE (*Integrated Development Environment*) é um conjunto de ferramentas usadas no desenvolvimento de *software*. Um ambiente de programação pode ser um editor de texto para criação do código-fonte de um programa e um compilador e, também, pode ser uma grande coleção de ferramentas integradas (SEBESTA, 2018), tais como os ambientes de programação mais recentes, entre os quais se destacam o *NetBeans* e o *Visual Studio*.

Neste livro vamos utilizar diferentes ambientes de programação, tais como: *SWI PROLOG* (para desenvolvermos os conceitos da programação em lógica), *Kawa* (programação funcional, utilizando o dialeto *Scheme*), *Lazarus* (IDE para desenvolvimento de aplicações utilizando a linguagem de programação *Object Pascal*) e o *NetBeans* (programação concorrente em *Java*).

1.9 Evolução das Linguagens de Programação

Com base nos estudos apresentados por Sebesta (2018), vamos apresentar uma breve evolução das linguagens de programação:

- Pseudocódigos: a programação de computadores, no final da década de 40 e início da década de 50, compreendia o desenvolvimento em código de máquina, também conhecido como programação de *hardware* mínima. Nesta época o desenvolvimento de *software* era muito difícil (baixo nível, muito ligado ao *hardware*), pois era necessário utilizar códigos numéricos para especificar instruções, além de ser preciso utilizar o endereçamento absoluto das posições de memória. Estas dificuldades foram as principais motivações para que fosse criada a linguagem *Assembly*;

- Linguagem de Programação FORTRAN: esta linguagem foi desenvolvida pela IBM (*International Business Machines*), juntamente com o sistema IBM 704 com instruções de ponto-flutuante em *hardware.* Até então as operações com ponto-flutuante eram simuladas em *software*, o que consumia muito tempo de execução. Na computação científica, a criação do IBM 704 marcou o fim da era interpretativa na computação científica, pois o FORTRAN é a primeira linguagem de programação de alto nível compilada, tendo sido lançada em abril de 1957;

- Programação Funcional: a primeira linguagem de programação funcional foi criada para oferecer recursos de linguagem para processamento de listas, cuja necessidade surgiu a partir das primeiras

aplicações na área de Inteligência Artificial, em meados da década de 50. Sendo assim, foi criada a linguagem de programação *LISP*, uma linguagem voltada ao processamento de listas. Era uma tentativa de unir os esforços de linguistas (para realizar o processamento de linguagem natural), de psicólogos (para modelar os processos fundamentais do cérebro) e de matemáticos (visando automatizar processos). A intenção era permitir que os computadores processassem dados simbólicos em listas encadeadas já que, na época, a computação fundamentava-se em processar dados numéricos armazenador em *arrays.* A linguagem *LISP* foi projetada como uma linguagem de programação funcional, sendo que os programas são compostos por funções e argumentos, não existindo instruções de atribuição e variáveis. Entre os dialetos da linguagem *LISP* destacam-se o *Scheme* (que será estudado neste livro) e o *Common LISP*;

- ALGOL (*ALGOrithmic Language*): a criação da linguagem de programação ALGOL compreende os esforços para projetar uma linguagem de programação universal. A linguagem ALGOL permitia a implementação de procedimentos recursivos, passagem de parâmetros por valor e por referência e a utilização de *arrays* dinâmicos e foi a primeira linguagem de programação com sintaxe descrita formalmente, utilizando a BNF. Entretanto, não possuía instruções de entrada/saída padrão, sendo as mesmas dependentes da implementação, tornando os programas difíceis de serem portados para outros computadores. Algumas linguagens de programação descendentes do ALGOL são: C*, Pascal* e *Ada*, entre outras;

- COBOL: linguagem de programação voltada a aplicações comerciais, traduzindo o desejo de utilizar a Língua Inglesa o máximo possível, visando ser fácil de utilizar, para ampliar o número de programadores. Foi a primeira linguagem de programação que permitia nomes longos (até 30 caracteres), bem como traços para conexão de palavras;

- BASIC (*Beginner´s All-purpose Simbolic Instruction Code*): linguagem muito popular nos microcomputadores no final da década de 70 e início da década de 80, sendo de fácil aprendizado. Seus dialetos menores podiam ser implementados em computadores com memória muito pequena, tais como o CP200 da Prológica, que possuía apenas 64Kb de memória RAM. A linguagem *BASIC* ressurgiu com o *Visual BASIC* (década de 90) e o *Visual BASIC.Net* (anos 2000);

- PL/I: foi a primeira tentativa em grande escala de projetar uma linguagem que poderia ser usada para um amplo espectro de áreas de aplicação, tanto científicas como comerciais. Foi desenvolvida, como o FORTRAN, como um produto da IBM. Foi a primeira linguagem de programação que permitiu: que os programas criassem tarefas executadas concorrentemente; detectar exceções e erros em tempo de execução e incluir ponteiros como tipos de dados;
- Simula 67: foi a linguagem de programação que deu origem à abstração de dados, permitindo a construção de **classe** (estrutura de dados);
- Linguagem C: linguagem de programação projetada com completa ausência de verificação de tipos o que a torna bastante flexível (ou insegura). Um importante motivo para o grande crescimento da popularidade obtido na década de 80 é que a linguagem C fazia parte do Sistema Operacional UNIX;
- PROLOG: programação baseada em lógica, utilizando uma notação lógica formal para comunicar processos computacionais a um computador: fatos e regras;
- ADA: esta linguagem de programação foi o resultado do mais extensivo e dispendioso esforço de projeto de uma linguagem de programação já realizado, sendo desenvolvida para o Departamento de Defesa dos EUA e tinha algumas características especiais, tais como a execução concorrente de unidades de programas especiais, chamadas tarefas;
- *SmallTalk*: linguagem de programação baseada no paradigma de Orientação a Objetos: a maturidade da programação OO foi atingida com essa linguagem. Os sistemas de janelas (*GUI – Graphical User Interface*) desenvolveram-se a partir do *SmallTalk*;
- Linguagem *C++*: linguagem com uma combinação de recursos imperativos e orientados a objeto, oferecendo um conjunto de classes predefinidas, juntamente com a possibilidade de criação de classes pelo usuário;
- Linguagem *Java*: foi inicialmente projetada para a programação de dispositivos eletrônicos incorporados de consumo (micro-ondas, torradeiras, televisores interativos). A partir de 1993 (com a proliferação dos *browsers* para acesso à *WWW*) a linguagem Java passou a ser considerada útil para a programação para a *web.*

Exercícios do Capítulo 1

Exercícios de Revisão: com base no que foi estudado no Capítulo 1, responda às perguntas:

Qual é a denominação de um conjunto consistente de regras para combinar as construções primitivas?

Qual a vantagem em utilizar tratamento de exceções?

Cite:

- Um exemplo de *aliasing:*
- Exemplos de linguagens para propósitos especiais, tais como a criação de Jogos:
- Exemplo de linguagem com um número pequeno de componentes básicos:
- Exemplo de linguagem com um grande número de componentes básicos:
- Exemplo de sobrecarga (*overloading*) de operador:

Conceitos de Linguagens de Programação

Associe as listas indicando, entre parênteses, na segunda lista, a letra correspondente à resposta que mais se encaixa na questão apresentada.

(a) Linguagem de programação que dominou a computação científica por mais de 40 anos

(b) Medida de quão facilmente uma linguagem pode ser usada para criar programas para um domínio de problema escolhido

(c) Linguagem de programação que dominou as aplicações comerciais nos últimos 40 anos

(d) Linguagem de programação utilizada em Inteligência Artificial

(e) Linguagem de programação na qual o UNIX foi escrito

(f) Exemplo de falta de ortogonalidade em uma linguagem de programação

(g) Conceito de *aliasing*

(h) Capacidade de definir e, depois, de usar estruturas ou operações complicadas de uma maneira que permita ignorar muitos dos detalhes

(i) Nome da categoria de linguagens de programação cuja estrutura é determinada pela arquitetura de *von Neumann*

(j) Critério de avaliação de uma linguagem de programação que indica que existem operadores muito poderosos que permitem que uma grande quantidade de computação seja realizada com um programa muito pequeno

(k) Recursos fundamentais de uma linguagem de programação orientada a objeto

(l) Primeira linguagem de programação a suportar os três recursos fundamentais da programação orientada a objeto

(m) Métodos gerais para implementar uma linguagem de programação

(n) Critério de avaliação de uma linguagem de programação que indica que um programa se comporta de acordo com as suas especificações sob todas as condições

(o) Produz uma execução de programa mais rápida do que um interpretador

(p) Função de um *linkeditor*

(q) Capacidade de um programa de interceptar erros em tempo de execução, pôr em prática medidas corretivas e prosseguir com a execução

(r) Gargalo de *von Neumann*

(s) Critério que indica a possibilidade de combinação de construções primitivas de uma linguagem para construir as estruturas de controle e de dados

(t) Tipos de instruções que preenchem um banco de dados Prolog

(u) Unidades de programa concorrentes da linguagem Ada

(v) Primeira aplicação proposta para a linguagem *Java*

(w) Nome dado para a coleção de métodos que os compiladores podem usar para diminuir o tamanho e/ou aumentar a velocidade de execução do código produzido

(x) Linguagem de programação que desenvolveu o conceito de construção de classe

(y) Critério que indica a facilidade de leitura de um programa

(z) Linguagem de programação cujo principal meio de fazer computações é aplicando funções a determinados parâmetros

Segunda lista

( ) *SmallTalk*

( ) Linguagem C

( ) Ter dois ou mais métodos ou nomes, distintos, para fazer referência à mesma célula da memória

( ) Herança, vinculação dinâmica, abstração de dados (encapsulamento)

( ) Fatos e regras

( ) Vinculação de bibliotecas e outros programas aos programas do usuário

( ) Legibilidade

( ) PROLOG

( ) Expressividade

( ) Imperativas

( ) LISP

( ) Compilação e interpretação

( ) Linguagem desenvolvida para a programação de dispositivos eletrônicos (TV, micro-ondas)

( ) Tratamento de exceções

( ) SIMULA 67

( ) FORTRAN

( ) Confiabilidade

( ) Linguagem de programação hipotética em que um ponteiro não pode apontar para um *array*

( ) Abstração

( ) Compilador

( ) Ortogonalidade

( ) *Writability*

( ) As instruções podem ser executadas muito mais rápido do que podem ser transferidas para o processador

( ) Otimização de código

( ) Tarefas

( ) COBOL

Conceitos de Linguagens de Programação

Responda verdadeiro (V) ou falso (F) para cada uma das afirmações abaixo:

( ) Existem quatro categorias principais de linguagens de programação: imperativas, funcionais, lógicas e orientadas a objeto

( ) Uma linguagem de programação lógica é baseada em funções

) Uma linguagem de marcação, como o HTML (*HyperText Markup Language*), é uma linguagem de programação

( ) Um computador pode ser projetado e construído com uma linguagem de alto nível própria

( ) Um sistema de implementação de linguagem é o único software necessário para o funcionamento do computador

( ) O processo de compilação traduz o código-fonte de um programa em linguagem de máquina

( ) As implementações iniciais da linguagem Java eram compiladas

( ) O UNIX é um dos ambientes de programação mais antigos

( ) Na década de 40 não existia nenhuma linguagem de programação de alto nível e a programação era feita em código de máquina

( ) A linguagem BASIC foi uma das mais utilizadas para o ensino de programação, especialmente na década de 70

( ) Prolog é um exemplo de linguagem de programação funcional

( ) A linguagem de programação C originou-se de um projeto anterior, denominado de linguagem B

( ) Tarefas são as unidades de programas concorrentes da linguagem Ada

( ) As linguagens PL/I, Simula 67, C, Pascal, Ada, C++ e Java são exemplos de linguagens descendentes da linguagem ALGOL

( ) A linguagem COBOL era destinada a aplicações científicas

( ) Ada foi a 1ª linguagem de programação que permitiu que os programas criassem tarefas executadas concorrentemente

( ) A origem da abstração de dados, com a possibilidade da definição de classes, surgiu com a linguagem de programação *SmallTalk*

( ) A linguagem de programação C foi utilizada para a implementação do Sistema Operacional UNIX

( ) A linguagem C foi o resultado do mais extensivo e dispendioso esforço de projeto de uma linguagem de programação já realizado

( ) Os sistemas de janelas (GUI – *Graphical User Interface*) desenvolveram-se a partir da linguagem *SmallTalk*

( ) A linguagem Java foi inicialmente projetada para a programação de dispositivos eletrônicos incorporados de consumo (micro-ondas, TVs, etc.)

( ) A sintaxe de uma linguagem de programação é o significado das expressões, instruções e unidades de programa

( ) Uma metalinguagem é uma linguagem utilizada para descrever outra linguagem

( ) A BNF é uma metalinguagem para descrever a semântica de uma linguagem de programação

( ) Símbolos como o colchete [ ] para indicar itens opcionais e chaves { } para indicar repetições fazem parte da BNF estendida

Conceitos de Linguagens de Programação: Gramáticas e Árvores de Análise Sintática

1) Escreva a derivação da sentença A := B + C * A de acordo com a gramática abaixo:

```
<atribuição>  →  <id> := <expressão>
<id> → A | B | C
<expressão>   →  <expressão> + <expressão>
                 | <termo>
<termo>  →  <termo> * <fator>
            | <fator>
<fator>  →  ( <expressão> )
            | <id>
```

2) Desenhe a árvore de análise (*parse tree*) da sentença A := B + C * A

3) Escreva a derivação da sentença ((a * b) + (a * a)) de acordo com a gramática abaixo:

<expressão> → <expressão> * <expressão>
| <expressão> + <expressão>
| (<expressão>)
| a
| b

4) Desenhe a árvore de análise (*parse tree*) da sentença ((a * b) + (a * a))

Capitulo 2

PROLOG: PROGRAMAÇÃO EM LÓGICA

Este capítulo apresenta conceitos do Paradigma de Programação Lógica, utilizando a linguagem PROLOG (*PROgramming in LOgic)*. Os exemplos práticos foram executados no ambiente SWI-PROLOG, disponível para *download* em https://www.swi-prolog.org/download/stable (SWI-PROLOG.ORG, 2021).

2.1 Programação em Lógica

A Programação em Lógica baseia-se na ideia de que um problema pode ser descrito por meio de relações sobre um conjunto de objetos, a partir as quais outras relações podem ser obtidas empregando-se regras de dedução. Para isso são utilizadas linguagens de programação declarativas (ou assercionais), que se constituem de notações que permitem especificar em que consiste um dado problema, sem precisar como resolvê-lo. Neste contexto, um programa é uma sequência de relações entre objetos (valores) e cada uma destas relações é definida em termos de condições e restrições, as quais, quando satisfeitas, implicam que a relação existe (CHARÃO, 2013; PEREIRA, 2019).

Uma equação algébrica, por exemplo, expressa uma relação entre os valores de suas variáveis. Assim, pode-se resolvê-la para qualquer uma das variáveis, desde que sejam conhecidos os valores das outras. De forma análoga, um programa escrito no paradigma lógico descreve uma relação e pode ser usado para encontrar valores não conhecidos, a partir dos já conhecidos. Neste contexto, um programa lógico compreende a definição de uma relação, onde cada cláusula especifica que a relação existe sob determinadas condições (lógica matemática dedutiva) (CHARÃO, 2013; PEREIRA, 2019).

A programação em lógica (ou programação lógica) diferencia-se da programação imperativa por requerer a descrição da estrutura lógica do problema, ao invés da maneira pela qual o computador deve solucioná-lo (algoritmo). Para isso, o programador precisa ter uma ideia clara das relações que envolvem o problema cuja solução está sendo procurada.

Um programa lógico é constituído de sentenças que expressam o conhecimento relevante ao problema que o programa pretende resolver. A representação do conhecimento é feita com base em objetos discretos (números, figuras geométricas, etc) e relações entre estes objetos (CHARÃO, 2013).

Por exemplo:

> Doki é um cachorro.
> Todo cachorro é um mamífero.
> Logo, Doki é um mamífero. *Isto é uma dedução*.

Para criamos um programa baseado no paradigma lógico, devemos definir fatos e regras sobre indivíduos e/ou objetos, bem como suas relações. A execução do programa se dá por meio de consultas sobre o relacionamento definido, ou seja, o usuário faz perguntas (consultas) para verificar as relações existem. Neste contexto, um programa escrito no paradigma lógico é conhecimento como uma **base de conhecimento** (CHARÃO, 2013; PEREIRA, 2019).

Um exemplo bastante clássico é o da construção de uma base de conhecimento sobre uma família. Para construirmos esta base de conhecimento, iniciamos definindo relações de parentesco, tais como:

> *pai(pedro, maria)* – Pedro é pai de Maria
> *mãe(maria, luiz)* – Maria é mãe de Luiz

Estas cláusulas (sem condições) definem **fatos** (verdades) sobre o domínio do problema. Podemos definir, também, **regras**, que são cláusulas com condições:

> *avô_materno(pedro,luiz) :- pai(pedro,maria), mãe(maria,luiz).*

Esta regra especifica que Pedro é avô materno de Luiz se Pedro é pai de Maria e Maria é mãe de Luiz. Esta conclusão ocorre porque as condições são verdadeiras. A conclusão é chamada de **cabeça** da cláusula e as condições constituem o **corpo** da cláusula

Podemos definir, também, **regras genéricas**, utilizando variáveis ao invés de objetos particulares. Por exemplo:

> avô_materno(X,Y) :- pai(X,Z), mãe(Z,Y)

Esta regra genética estabelece que qualquer X é avô materno de Y se Z é filho de X e Z é mãe de Y. Estas duas últimas partes da regra são as condições.

Outro exemplo envolve criarmos regras para a relação de *irmão* (por parte de pai e/ou por parte de mãe):

```
irmão(X,Y) :- pai(Z,X), pai(Z,Y)
irmão(X,Y) :- mãe(Z,X), mãe(Z,Y)
```

Esta regra estabelece que X e Y são irmãos, se existe um indivíduo Z que seja pai de X e de Y ou se existe um indivíduo Z que seja mãe de X e de Y. Neste exemplo X, Y e Z são variáveis e terão seus valores instanciados (definidos) no momento em que o programa for executado, ou seja, no momento em que a base de conhecimento for consultada pelo usuário.

Podemos, desta forma, construir uma base de conhecimento de fatos e regras que descreve todas as possíveis relações de família: tio, tia, primo, avô, etc. A partir desta base de conhecimento pode-se questionar o programa (consultar). **Consultas** são cláusulas expressas como condições (relações) a partir das quais se deseja obter uma conclusão. Sendo assim, são cláusulas sem cabeça, ou seja, sem conclusão.

Algumas consultas requerem uma resposta simples, tal como sim ou não. Outras requerem que um ou mais objetos sejam fornecidos como resposta. Por exemplo:

Pedro é primo de Luiz?

:- primo(pedro,luiz)

De quem Maria é mãe?

:- mãe(maria,X)

Quem são os pais de Maria?

:- mãe(X,maria), pai (Y,maria)

Quais são os relacionamentos tio-sobrinho existentes na base de conhecimento?

:- tio(X,Y)

Existem diferentes notações para expressar relações. A base da linguagem Prolog são as cláusulas de Horn. Existem 3 tipos de cláusulas Horn: fatos, regras e consultas (CHARÃO, 2013; PEREIRA, 2019):

- **Fatos:** são cláusulas que contêm apenas a conclusão (ou cabeça), por exemplo: pai(pedro,maria).

- **Regras:** Cada regra é formada por uma conjunção de uma ou mais condições, separadas por vírgulas (as quais são lidas como “e”), que implica em uma conclusão. Por exemplo: avô_paterno(X,Y) :- pai(X,Z), pai(Z,Y).
- **Consultas:** são cláusulas (sem conclusão) formadas por conjunções de uma ou mais condições. Por exemplo: :- mãe(X, Maria), pai(Y,Maria).

2.2 Ciclo de Programação Prolog

Para criarmos um programa segundo o paradigma lógico precisamos, inicialmente, fornecer a base de conhecimento, contendo os fatos e as regras e armazená-los em um arquivo. A execução do programa deve ser realizada por meio de consultas, visando verificar proposições, de acordo com o conhecimento armazenado na base.

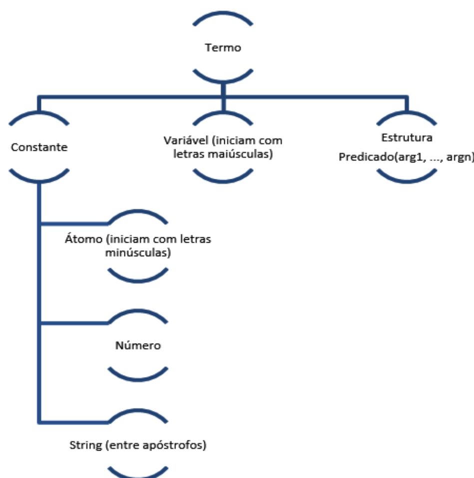
O programa é um *mundo* fechado. Isso significa que um fato é considerado falso quando não está presente na base de conhecimento do programa ou não é dedutível a partir das regras estabelecidas (relações) (CHARÃO, 2013).

Por exemplo, supondo que criamos uma base de conhecimento sobre *família* e declaramos fatos e regras que envolvem os parentescos *pai, mãe* e *irmãos*. Se quisermos fazer uma consulta sobre tios, a base de conhecimento não está organizada de forma a nos permitir encontrar esta relação.

2.3 Instruções em PROLOG

As instruções em PROLOG podem ser divididas em constantes (átomos, números e strings), variáveis e estruturas (predicados), como mostra o esquema gráfico da Figura 6 (CHARÃO, 2013).

Figura 6: Tipos de Instruções em PROLOG



Fonte: Adaptada de CHARÃO (2013)

Por exemplo:

Pedro é pai de João: pedro e joão são átomos.
pai(pedro,joão).

De quem Pedro é pai? X é uma variável.
pai(pedro,X).

Maria tem 35 anos.
idade(maria,35).

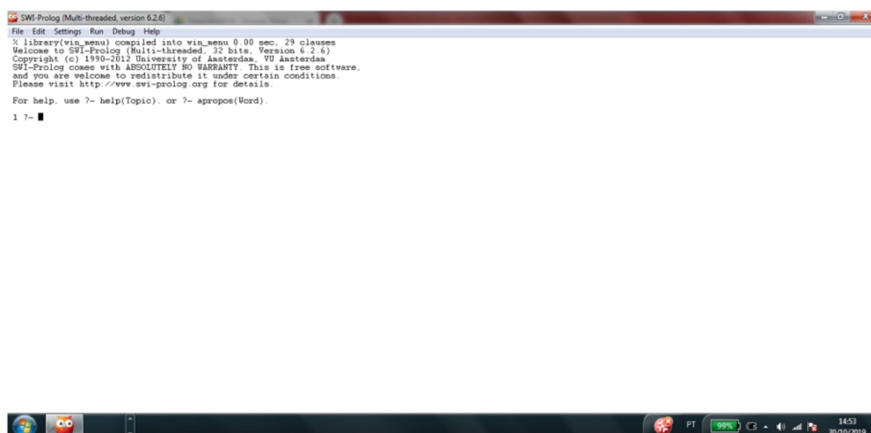
Maria mora na cidade de Frederico Westphalen.
mora(maria,'Frederico Westphalen').

Podemos fazer um teste inicial, criando um programa em Prolog, utilizando o ambiente SWI-Prolog. Vamos digitar a seguinte base de conhecimento para teste:

```
pai(pedro,joão).
pai(pedro,maria).
idade(pedro,80).
idade(joão,45).
idade(maria,35).
mora(maria,'Frederico Westphalen').
mora(joão,'Porto Alegre').
mora(pedro,'Porto Alegre').
```

A Figura 7 mostra a tela inicial do ambiente SWI-Prolog. Nesta tela vemos o *prompt* (sinal de interrogação), onde poderemos consultar um programa (executar). Para isso, devemos usar o comando *consult* e indicarmos o arquivo com a extensão *pl* (arquivo .pl) onde está armazenado o código-fonte do nosso programa (a base de conhecimento).

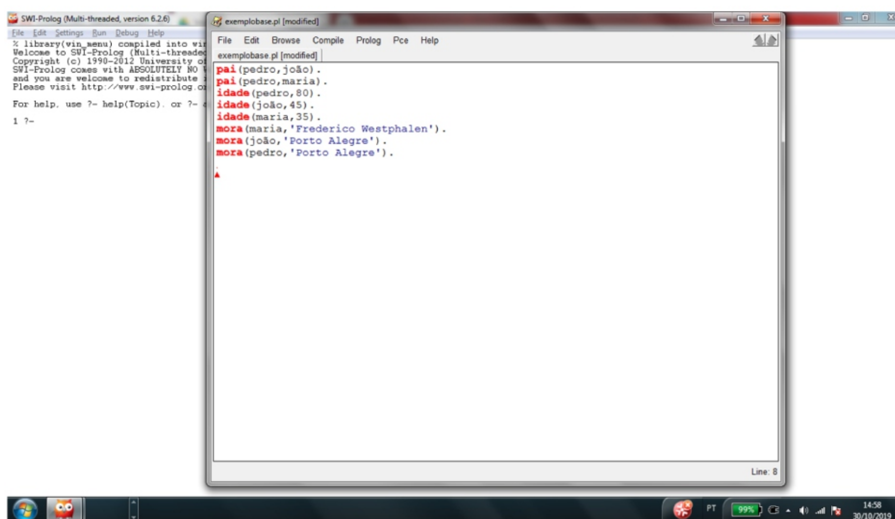
Figura 7 – Tela Inicial do SWI-Prolog



Fonte: Elaborado pelos autores

Para criarmos um novo programa, devemos acessar o menu *File* e selecionarmos a opção *New.* Será apresentada, inicialmente, uma tela para definirmos o local (pasta) e o nome do arquivo do tipo *Prolog Source* (arquivo com a extensão *pl*). Em nosso exemplo, criamos um arquivo denominado *exemplobase*. A seguir será mostrada uma tela onde poderemos digitar e editar o nosso código-fonte, ou seja, a nossa base de conhecimento, como mostra a Figura 8 (já contendo o código-fonte digitado).

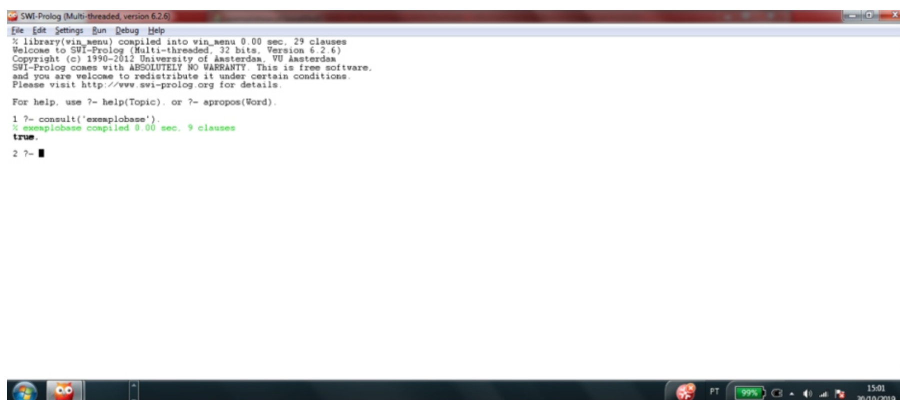
Figura 8 – Exemplo de código-fonte em Prolog (base de conhecimento)



Fonte: Elaborado pelos autores

Após digitarmos o código-fonte (base de conhecimento), devemos selecionar a opção *Save Buffer*, disponível no menu *File* da janela de edição. Para executarmos as consultas, ou seja, executarmos o programa, devemos voltar para a janela contendo o *prompt* e digitarmos o comando *consult('exemplobase')*, como mostra a Figura 9.

Figura 9 – Execução (consulta a uma base de conhecimento)



Fonte: Elaborado pelos autores

Após carregarmos o arquivo *exemplobase* para consulta, por meio do comando *consult,* podemos realizar diferentes consultas, de acordo com as

cláusulas existentes em nossa base de conhecimento, como mostram alguns exemplos da Figura 10.

Figura 10 – Exemplos de Consultas

```
SWI-Prolog (Multi-threaded, version 6.2.6)
File  Edit  Settings  Run  Debug  Help
% library(win_menu) compiled into win_menu 0.00 sec, 29 clauses
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 6.2.6)
Copyright (c) 1990-2012 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?- consult('exemplobase').
% exemplobase compiled 0.00 sec, 9 clauses
true.

2 ?- pai(pedro,joão).
true .

3 ?- pai(pedro,X).
X = joão ;
X = maria.

4 ?- pai(pedro,josé).
false.

5 ?- pai(X,Y).
X = pedro,
Y = joão ;
X = pedro,
Y = maria.

6 ?- idade(maria,X).
X = 35.

7 ?- idade(maria,34).
false.

8 ?- mora(maria,X).
X = 'Frederico Westphalen'.

9 ?- mora(X,'Frederico Westphalen').
X = maria.

10 ?-
```

Fonte: Elaborado pelos autores

Vamos analisar as consultas realizadas e as respostas obtidas:

Na primeira consulta, estamos verificando se *Pedro é pai de João*, ou seja, se esta relação está definida (ou pode ser deduzida) a partir das informações armazenadas em nossa base de conhecimento. O resultado é *true,* já que existe este predicado *pai(pedro,joão)* em nossa base.

?- pai(pedro,joão).

true .

Na segunda consulta estamos trabalhando com uma variável (X). Estamos perguntando, então, de quem *Pedro* é pai. Será mostrado, se existir, a primeira relação que satisfaz à consulta. Se pressionarmos a tecla <*Enter*> serão mostradas as próximas relações. Neste caso, *Pedro*, conforme a nossa base de conhecimento, é pai de *João* e de *Maria*, que serão os valores a serem instanciados na variável X.

?- pai(pedro,X).

X = joão ;

X = maria.

A próxima consulta não envolve variáveis. Estamos querendo verificar se *Pedro* é pai de *José*, ou seja, verificar se esta relação existe (ou poderia ser deduzida) a partir das cláusulas existentes na nossa base de conhecimento. Neste caso o resultado é false, já que esta relação não existe.

?- pai(pedro,josé).

false.

A consulta a seguir envolve duas variáveis, X e Y. Sendo assim, a intenção é mostrar *quem é pai de quem*, de acordo com a base de conhecimento.

?- pai(X,Y).
X = pedro,
Y = joão ;
X = pedro,
Y = maria.

Na próxima consulta perguntamos qual é a idade de *Maria*.

?- idade(maria,X).
X = 35.

E, na consulta seguinte, queremos verificar se *Maria* tem 34 anos de idade (no caso é falso).

?- idade(maria,34).
false.

As próximas consultas envolvem o predicado *mora*. Na primeira consulta, queremos saber onde *mora Maria* e, na segunda, quem é que mora em *Frederico Westphalen*.

?- mora(maria,X).
X = 'Frederico Westphalen'.
?- mora(X,'Frederico Westphalen').
X = maria.

2.3.1 Variáveis

Na programação lógica, as variáveis não são como variáveis de linguagens imperativas, pois as mesmas não são declaradas e não são vinculadas a tipos. A vinculação de valor (instanciação, de forma dinâmica) ocorre no processo de resolução, para verificar se uma meta é verdadeira (CHARÃO, 2013). Por exemplo, considerando a regra abaixo:

ave(X) :- papagaio(X).

Esta regra especifica que todo objeto instanciado em uma variável X que for um *papagaio* é uma ave.

Supondo a seguinte base de conhecimento:

papagaio(‘Caco’).
ave(X) :- papagaio(X).

Se realizarmos a seguinte consulta: ave(X), a resposta será X=’Caco’.

2.3.2 Declaração de Fatos

A cláusula a seguir é um fato, onde *pai* é um predicado.

pai(pedro,joao).

Na cláusula, *pedro* e *joao* são argumentos, separados por vírgulas e são, também, constantes (átomos).

Um fato pode ser universal, de acordo com a sintaxe: fato universal(const1, ..., var1). Por exemplo, podemos dizer que todos os seres humanos são ancestrais de *Adão,* por meio do fato universal *ancestral("Adão",X)* que indica que *para todo X, Adão é ancestral de X*. A declaração de um fato universal emprega variáveis e constantes (CHARÃO, 2013).

Devemos seguir convenções quanto ao número de argumentos, ordem e nomes, para que nossa base de conhecimento, bem como as respostas para as consultas, seja coerente (CHARÃO, 2013). Por exemplo:

pai(joão, maria, josé) é diferente de pai(joão,maria).
pai(josé,paulo) é diferente de pai(paulo,josé).
pai(joão,maria) é diferente de pAi(joão,maria).

2.3.3 Declaração de Propriedades

Para declarar uma propriedade (característica, atributo), devemos seguir a sintaxe: propriedade(objeto_constante). Uma propriedade é um predicado. A declaração de uma propriedade deve conter somente constantes (CHARÃO, 2013). Por exemplo:

cachorro(doky).
magro(doky).

Estas propriedades indicam fatos: *Doky* é um cachorro e *Doky é magro.*

2.3.4 Declaração de Relações

A declaração de uma relação deve seguir a sintaxe *relação(objeto_constante1, ..., objeto_constanteN)* e deve empregar somente constantes (CHARÃO, 2013). Por exemplo:

pai(joão,maria).
João é pai de Maria.

2.4 Formas de Consultas

Ao executar um programa escrito no paradigma lógico, estamos realizando uma consulta. Uma consulta pode ser para: 1) validar uma proposição (regra) ou 2) verificar a existência de um fato (CHARÃO, 2013).

Por exemplo, supondo que queiramos fazer uma consulta para validar uma proposição. Podemos, no *prompt* do *SWI-Prolog,* utilizarmos a sintaxe predicado(arg1,...,argn). O resultado será *true* ou false, indicando ou não a validação da referida proposição. Por exemplo, supondo a existência de dois fatos na base de conhecimento:

```
cachorro(doky).
cachorro(rex).
```

E as seguintes consultas:

```
?- cachorro(doky).
true
?- cachorro(pluto).
false
```

A primeira consulta resultou *true,* pois existe um cachorro chamado *doky* na base de conhecimento. Entretanto, a segunda consulta retornou false, pois o cachorro consultado *(pluto)* não existe. Se quisermos que esse segundo cachorro seja incluído na base de conhecimento, basta abrirmos o código-fonte do nosso programa em Prolog e incluirmos o predicado *cachorro(pluto)*.

Podemos usar as consultas para verificarmos a existência de um fato ou de uma relação, como por exemplo:

```
cachorro(doky).
cachorro(dudu).
gordo(doky).
?- gordo(X).
X=doky
?- cachorro(X).
X=doky;
X=dudu.
```

O ponto-e-vírgula (;) verifica se existe outra resposta na base de conhecimento.

2.5 Uso de Operadores Relacionais

Os operadores relacionais podem ser usados na elaboração de regras, na base de conhecimento, ou nas consultas. Os operadores relacionais em Prolog são os seguintes:

Operador	Significado
>	Maior
<	Menor
>=	Maior ou igual
=<	Menor ou igual
=:=	Igual
=\=	Diferente

Vejamos os exemplos abaixo, supondo a seguinte base de conhecimento:

```
idade(pedro,35).
idade(ana,30).
idade(paulo,27).
```

Com base nestas cláusulas, podemos realizar algumas consultas com operadores relacionais:

A primeira consulta vai mostrar o nome da pessoa e a sua respectiva idade, desde que a mesma seja menor ou igual a 30:

```
?- idade(N,X), X =< 30.
N=ana,
X=30;
N=paulo,
X=27.
```

No segundo exemplo de consulta serão mostradas apenas as pessoas e respectivas idades, desde que sejam apenas menores do que 30.

```
?- idade(N,X), X>25, X<30.

N=paulo,

X=27.
```

Podemos utilizar os operadores relacionais para verificar o resultado de operações aritméticas. Por exemplo, nesta consulta estamos perguntando se o resultada da soma de 1 mais 2 é diferente da soma de 2 mais 1 (resulta *false*).

```
?- 1+2 =\= 2+1.
False
```

A mesma consulta, com o operador de comparação de igualdade, resulta *true.*

```
?-1+2 =:= 2+1.
True
```

2.6 Declaração de Regras

Vamos verificar a construção de uma regra por meio de um exemplo em que, todo X (uma variável) que for um papagaio então é uma *ave* (é uma dedução por meio da regra). Temos, então, dois predicados nesta regra: *ave* e *papagaio* e uma variável (X).

```
ave(X) :- papagaio(X).
```

As regras são cláusulas com condições e tem, como forma geral: consequente :- antecedente. Os símbolos :- devem ser lidos como um se (como se fosse um comando *if* em outras linguagens de programação).

As regras podem ser compostas, utilizando os conetivos e (representado por uma vírgula - ,) e *ou* (representado por um ponto-e-vírgula ;). Por exemplo: se X é pai de Y e Y é pai de Z, então X é avô de Z.

```
avo(X,Z) :- pai(X,Y), pai(Y,Z).
```

Podemos verificar um exemplo com o conetivo *ou*: se X é um papagaio ou uma coruja, então X é uma ave:

```
ave(X) :- papagaio(X); coruja(X).
```

2.7 Aritmética Simples

Em Prolog podemos utilizar o operador *is* para implementarmos regras contendo operações aritméticas. Por exemplo: a regra *soma* tem 3 variáveis, A, B e C, sendo que a variável A irá armazenar a soma dos valores que forem instanciados para as variáveis B e C:

```
soma(A,B,C) :- A is B + C.
?- soma(A,1,4).
5
```

Devemos lembrar que este *operador não deve ser usado como uma atribuição*. Por exemplo:

soma(A,B,C) :- A is A+B+C.

Esta cláusula está escrita de forma errada, já que a variável A já está instanciada antes do *is* e não pode estar, também, do lado direito. Também não podemos utilizar um predicado no lugar da expressão aritmética. Neste sentido, a sentença A *is soma(A,3,4)* está incorreta.

Exercícios do Capítulo 2

1) Expresse por meio de fatos e regras Prolog as informações contidas na seguinte frase (Adaptado de LIMA, 2019):

   Doky é um cachorro. Garfield é um gato. Nemo é um peixe. Dudu é um pássaro. João é uma pessoa. Maria é uma pessoa. Doky é magro. Garfield é gordo. Gatos gostam de peixes. Gatos gostam de pássaros. Cachorros gostam de pessoas. Gatos gostam de pessoas. Os gatos comem tudo que gostam, exceto pessoas.

   Após criar a base de conhecimento, faça consultas para mostrar as seguintes informações:

   - quais são os cachorros que existem na base de conhecimento?
   - quais são os gatos que existem na base de conhecimento?
   - quais são os peixes que existem na base de conhecimento?
   - quais são os pássaros que existem na base de conhecimento?
   - quais são as pessoas que existem na base de conhecimento?
   - quem é magro?
   - quem é gordo?
   - quem gosta de quem?
   - o que os gatos comem?

2) Expresse por meio de fatos e regras Prolog as informações contidas na seguinte frase (Adaptado de LIMA, 2019):

   Ana é bonita. Marcos é rico e bonito. Maria é rica e forte. Rodrigo é forte e bonito. Sílvio é amável e forte. Todos os homens gostam de mulheres bonitas. Todos os homens ricos são felizes. Qualquer homem que gosta de uma mulher que gosta dele é feliz. Qualquer mulher que gosta de um homem que gosta dela é feliz. Maria gosta de qualquer homem que gosta dela. Ana gosta de qualquer homem que gosta dela, desde que ele seja rico e amável ou bonito e forte.

   Após criar a base de conhecimento, faça consultas para mostrar as seguintes informações:

- quem é bonito(a)?
- quem é rico(a)?
- quem é feliz?
- de quem Maria gosta?
- de quem Ana gosta?

3) Considere a seguinte base de fatos em Prolog (Adaptado de LIMA, 2019):

```
aluno(joao,calculo).
aluno(maria,calculo).
aluno(joel,programacao).
aluno(joel,estrutura).
frequenta(joao,ufsm).
frequenta(maria,ufsm).
frequenta(joel,uri).
professor(carlos,calculo).
professor(ana_paula,estrutura).
professor(pedro,programacao).
funcionario(pedro,ufsm).
funcionario(ana_paula,ufsm).
funcionario(carlos,uri).
```

Escreva as seguintes consultas em Prolog:

a) Quem são os alunos do professor X?

b) Quem são as pessoas que estão associadas a uma universidade X? (alunos e professores)

4) Suponha os seguintes fatos (Adaptado de LIMA, 2019):

```
nota(joao,5.0).
nota(maria,6.0).
nota(joana,8.0).
nota(mariana,9.0).
nota(cleuza,8.5).
nota(jose,6.5).
nota(joaquim,4.5).
nota(mara,4.0).
nota(mary,10.0).
```

Considerando que:

Nota de 7.0 a 10.0 = aprovado

Nota de 5.0 a 6.9 = recuperação

Nota de 0.0 a 4.9 = reprovado

Escreva uma regra em Prolog para identificar a situação de um determinado aluno (aprovado, em recuperação ou reprovado).

5) Imagine um contexto definido pelos seguintes predicados (PEREIRA, 2019):

O predicado mora, que relaciona uma pessoa ao bairro em que ela mora

```
mora(pessoa,bairro).
```

O predicado pertence, que relaciona um bairro à região ao qual ele pertence

```
pertence(bairro,região).
```

O predicado amigo, que relaciona duas pessoas que são amigas

```
amigo(pessoa,pessoa).
```

O predicado tem_carro, que define as pessoas que têm carro

```
tem_carro(pessoa).
```

a) Crie uma coleção de fatos a respeito desses predicados, ou seja, popule a base de conhecimento

b) Defina uma regra estabelecendo que uma pessoa pode dar carona a outra se essa pessoa tem carro e ambas moram em bairros que ficam na mesma região

6) Crie uma base de conhecimento em Prolog declarando os fatos representados na seguinte tabela, que representa um catálogo de filmes (Adaptado de LIMA, 2019):

Título	Gênero	Diretor	Ano	Minutos
Amnésia	Suspense	Nolan	2000	113
Babel	Drama	Inarritu	2006	142
Capote	Drama	Miller	2005	98
Casablanca	Romance	Curtiz	1942	102
Matrix	Ficção	Wachowsk	1999	136
Rebecca	Suspense	Hitchcock	1940	130
Shrek	Aventura	Adamson	2001	90
Sinais	Ficção	Shymalen	2002	106
Spartacus	Ação	Kubrik	1960	184
Superman	Aventura	Donner	1978	143
Titanic	Romance	Cameron	1997	194
Tubarão	Suspense	Spielberg	1975	124
Volverine	Drama	Almodovar	2006	121

De acordo com esta base de conhecimento, e escreva consultas que possam responder às seguintes perguntas:

a) Quem dirigiu o filme Titanic?
b) Quais são os filmes de suspense?
c) Quais os filmes dirigidos por Donner?
d) Em que ano foi lançado o filme Sinais?
e) Quais os filmes com duração inferior a 100 minutos?
f) Quais os filmes lançados entre 2000 e 2005?
g) Quais os filmes lançados antes de 1980?

7) Construa, no SWI-PROLOG, a seguinte base de conhecimento, sobre automóveis:

```
modelo(grand_livina).
modelo(spin).
modelo(fiesta_hatch).
modelo(fiesta_sedan).
modelo(gol).
marca(nissan).
marca(chevrolet).
marca(ford).
marca(volkswagen).
fabrica(nissan,grand_livina).
```

```
fabrica(chevrolet,spin).
fabrica(ford,fiesta_hatch).
fabrica(ford,fiesta_sedan).
fabrica(volkswagen,gol).
lugares(grand_livina,7).
lugares(spin,7).
lugares(fiesta_hatch,5).
lugares(fiesta_sedan,5).
lugares(gol,5).
portamalas(grand_livina,grande).
porta-malas(spin,grande).
porta_malas(fiesta_sedan,médio).
porta_malas(fiesta_hatch,pequeno).
porta_malas(gol,pequeno).
```

De acordo com esta base de conhecimento, construa consultas para responder às seguintes perguntas:

a) Quais modelos existem na base de conhecimento?
b) Quais marcas existem?
c) Quais marcas fabricam quais modelos de carros?
d) Quais são os carros fabricados pela Ford?
e) Quantos lugares têm cada um dos modelos de carros?
f) Quais carros têm 7 lugares?
g) Qual é o tamanho do porta-malas de cada modelo de carro?
h) Quais são os carros que têm o porta-malas pequeno?

Ainda de acordo com esta base de conhecimento, crie regras para:

a) Mostrar qual(is) é(são) o(s) modelo(s) de carro(s) mais indicado(s) para uma família grande

b) Entre 2 carros (2 variáveis diferentes), qual é o carro que tem mais lugares?

Capitulo 3

PROGRAMAÇÃO FUNCIONAL

Neste capítulo vamos estudar o paradigma de Programação Funcional, onde os programas são formados, exclusivamente, por funções. Os exemplos práticos serão realizados utilizando o ambiente *Kawa* (disponível no *link* https://www.gnu.org/software/kawa/) por meio da linguagem *Scheme,* um dialeto do LISP.

3.1 Conceitos de Programação Funcional

No paradigma de programação funcional os programas são formados, exclusivamente, por funções. O próprio programa principal é uma função que recebe dados de entrada como argumentos e produz um resultado (PELLEGRINI, 2013).

A função principal é definida em termos de outras funções, as quais, por sua vez, são definidas em termos de outras mais simples, até o nível em que as funções são primitivas da linguagem (como se fossem comandos – palavras reservadas) (PELLEGRINI, 2013).

Entre algumas das principais características de um programa escrito no paradigma funcional, destacamos (PELLEGRINI, 2013):

- Programas funcionais não contêm comandos de atribuição: uma vez que uma variável assume um dado valor, este nunca muda;
- Programas funcionais não produzem quaisquer efeitos colaterais: uma chamada de função não produz outro efeito que computar o seu resultado. Isto faz com que a ordem de execução seja irrelevante – o programador não precisa descrever o fluxo de controle.

Um efeito colateral ocorre quando uma função altera um parâmetro de entrada/saída ou uma variável não local, por exemplo, quando alteramos um valor de uma variável global (escopo global) (PELLEGRINI, 2013). Por exemplo:

```
a=10
b= a + funcao(&a)
```

Temos, no exemplo, uma variável a com o valor 10 e uma variável *b* que recebe o valor da variável a somado ao resultado da função denominada *funcao*, que recebe o endereço da variável *a (&a)* como argumento de entrada, ou seja, as modificações que forem feitas nesta variável *a*, dentro da função, serão refletidas fora da mesma (escopo global), pois está sendo utilizada a passagem de parâmetros por referência (ao invés de ser criada uma cópia da variável *a* – passagem de parâmetros por valor), a função utilizará a mesma área de memória da variável a no escopo global.

Assumindo que ***funcao*** retorna o parâmetro dividido por 2 e modifica o valor para 20:

Qual é o valor de *b* se:

- a for avaliado primeiro: ??? **15**
- *funcao* for avaliada primeiro: ??? **25**

Como expressões podem ser avaliadas em qualquer momento, pode-se livremente substituir variáveis por seus valores e vice-versa. Isso significa que os programas funcionais são referencialmente transparentes (transparência referencial). Esta liberdade faz com que os programas funcionais sejam, matematicamente, mais facilmente manipuláveis do que os procedimentais. Quando uma expressão de uma linguagem é substituída por um valor e, esse valor é o resultado da avaliação da expressão, a semântica (sentido) do programa não se altera (PELLEGRINI, 2013). Por exemplo: supondo o seguinte trecho de código escrito no paradigma funcional:

```
x=1
inc(a) = a + 1
2 + inc(x) + inc(x)   O resultado será 2 + 2 + 2 = 6.
```

Em linguagens imperativas:

```
int x=1
int inc(a) = a + 1
2 + inc(x) + inc(x)    O resultado será  2 + 2 + 3 = 7.
```

O resultado, em uma linguagem imperativa, é diferente, pois, ao executar o segundo comando *inc(x)* o valor de x já terá sido alterado para 2 e, 2+1 resulta em 3. No paradigma funcional, por conta da transparência referencial, sempre que executarmos *inc(x)*, sendo x = 1, a função sempre retornará 2.

A programação funcional facilita a programação modular, já que permite construir funções complexas a partir de funções simples e programas a partir da composição de outros programas, ou seja, a partir da combinação de funções. Uma função é uma regra de correspondência, como uma função matemática. Uma função matemática é uma correspondência biunívoca de membros de um conjunto, chamado conjunto domínio, com outro conjunto, o conjunto imagem (PELLEGRINI, 2013).

As definições de funções frequentemente são escritas como um nome de função, seguido de uma lista de parâmetros entre parênteses, seguida da expressão de correspondência. Por exemplo, a função cubo, que recebe um parâmetro *x*, retorna o cubo de *x*:

cubo(x) = x * x * x, em que x é um número real

Nessa definição, os conjuntos domínio e imagem são os números reais. Utilizando a sintaxe da linguagem *Scheme*, a função cubo seria escrita assim:

(define (cubo x) (x x x))*

Veremos as questões de sintaxe da linguagem *Scheme* a seguir, neste mesmo capítulo.

O parâmetro x pode representar qualquer membro do conjunto domínio, mas é fixado para representar um elemento específico durante a avaliação da expressão da função (PELLEGRINI, 2013).

A aplicação de uma função, ou seja, sua execução, é realizada quando informamos o nome da função com um elemento particular do conjunto domínio. O elemento do conjunto imagem é obtido avaliando-se a expressão de correspondência da função com os do conjunto domínio, substituído pelas ocorrências do parâmetro (PELLEGRINI, 2013). Por exemplo, a chamada da função *cubo(2)* cubo(2,0) resulta no valor 8.

Toda ocorrência de um parâmetro é vinculada a um valor do conjunto domínio e considerada uma constante durante a avaliação, o que difere a programação funcional da programação imperativa, em que os valores dos parâmetros são variáveis (PELLEGRINI, 2013).

Neste contexto, o objetivo de uma linguagem de programação funcional é imitar as funções matemáticas no maior grau possível e os programas são definições de

funções e de especificações de aplicação destas. A execução de um programa consiste em avaliar as funções construídas. A execução de uma função sempre produz o mesmo resultado quando dados os mesmos parâmetros, que é a característica que já estudamos, a transparência referencial (PELLEGRINI, 2013).

Uma linguagem funcional oferece: 1) um conjunto de funções primitivas (como se fossem instruções ou comandos – palavras reservadas); 2) um conjunto de formas funcionais para construir funções complexas a partir das primitivas (combinação de funções); 3) uma operação de aplicação das funções e 4) alguma estrutura ou estruturas para representar dados (PELLEGRINI, 2013).

3.2 Linguagem *Scheme*

A linguagem de programação Scheme é um dialeto do LISP. As funções *Scheme* podem ser os valores de expressões e elementos de listas (estruturas de dados), e podem ser aplicadas em variáveis e passadas como parâmetros (PELLEGRINI, 2013).

3.2.1 Funções Primitivas

Os nomes em *Scheme* (nomes de uma função, de parâmetros) podem consistir em letras, dígitos e caracteres especiais (excetuando-se parênteses). A linguagem não é case *sensitive*, ou seja, não há distinção entre maiúsculas e minúsculas. Devemos lembrar que os nomes não devem iniciar por um dígito.

O interpretador *Scheme* é um laço infinito de leitura-avaliação-escrita. O interpretador lê repetidamente uma expressão digitada pelo usuário, interpreta-a e mostra o valor resultante. Sendo assim, utilizando o interpretador Kawa não teremos a opção de salvar o código-fonte de nosso programa (as funções construídas e suas aplicações – execução das mesmas).

As expressões com chamadas a funções primitivas são avaliadas da seguinte maneira:

- cada uma das expressões parâmetro é avaliada (sem nenhuma ordem particular);
- a função primitiva é aplicada aos valores do parâmetro e o valor resultante é exibido.

Vamos ver alguns exemplos de primitivas para operações aritméticas básicas de adição (+), subtração (-), multiplicação (*) e divisão (/):

Expressão	Valor
42	42
(* 3 7)	21
(+ 5 7 8)	20
(- 5 6)	-1
(- 15 7 2)	6
(-24 (* 4 3))	12

Devemos destacar que as operações aritméticas são escritas no formato pré-fixado, em que a operação aritmética deve ser colocada mais à esquerda. Por exemplo: (* 3 7), significa que queremos multiplicar 3 por 7: 3 * 7, que resulta 21.

Se tivermos mais de uma operação aritmética na mesma função, será executada, primeiro, a operação mais interna (parênteses mais internos). Por exemplo, expressão (-24 (* 4 3)) temos uma operação de subtração e uma de multiplicação. O interpretador *Scheme* irá executar, primeiro, a multiplicação de 4 por 3, resultando 12. Em seguida, irá realizar a subtração de 24 menos 12, resultando 12.

Podemos, também, verificar se o resultado de uma expressão é *true* ou *false,* podendo aplicar esta expressão em um comando condicional. Por exemplo, se digitarmos a seguinte expressão no interpretador *Kawa*:

```
(= (* 3 4)(* 2 6))
```

O interpretador responde *#t* – *true*: primeiro serão executadas as operações aritméticas dos parênteses mais internos: * 3 4 = multiplicar 3 por 4, resultando 12, * 2 6 = multiplicar 2 por 6, resultando, também 12. Posteriormente será realizada a operação de comparação, por meio do símbolo de igualdade (=) que retornará *true*, já que as duas operações resultaram no número 12.

Algumas Funções Primitivas

sqrt – retorna a raiz quadrada do argumento

random – produz um número randômico entre 0 e -1

display – avalia e mostra na tela o valor do argumento

newline – a próxima operação da tela inicia na linha seguinte

atom? – retorna #t se o argumento é um átomo, senão retorna nulo

null? – retorna #t se o argumento é um nulo, senão retorna nulo

eq? – retorna #t se os argumentos simbólicos são iguais, senão retorna nulo

= - retorna #t se os argumentos numéricos são iguais, senão retorna nulo

if: produz o efeito de avaliação condicional

(if predicado expr_consequente expr_alternativa)

cond (case)

(cond (predicado expressão)
(predicado expressão)
.....
(else expressão))

and: avalia os argumentos até que algum resulte em #f e então retorna #f (se nenhum resultar #f, retorna o valor do último argumento):

or: avalia os argumentos até que algum resulte em um valor diferente de #f e então retorna este valor; senão retorna #f

Algumas funções primitivas: exemplos

(eq? 'a 'b) #f
(eq? 'a 'a) #t
(atom? 5) #t
(atom? nil) #t

*Exemplo de utilização da primitiva **if***

(define (duplo valor)(* valor valor))

(if (=(+ 3 6)(duplo 3)) (sqrt 25) (sqrt 81))

*Exemplo de utilização da primitiva **cond***

(cond ((= 0 1) 3) (**(or (< 5 4)**(= 2 9)) 4)(else 5))

Qual o resultado que será obtido com a avaliação desta função?

(cond ((= 0 1) 3) (**(or (< 4 5)**(= 2 9)) 4)(else 5))

Qual o resultado que será obtido com a avaliação desta função?

(cond ((= 1 1) 3) (**(or (< 4 5)**(= 2 9)) 4)(else 5))

Qual o resultado que será obtido com a avaliação desta função?

Exemplos de utilização das primitivas *and* e *or*

(and (= 1 1)(> 5 4)) retorna #t

(and 1 2 3) retorna 3

(or (= 1 1)(> 5 4)) retorna #t

3.2.2 Escrevendo Funções

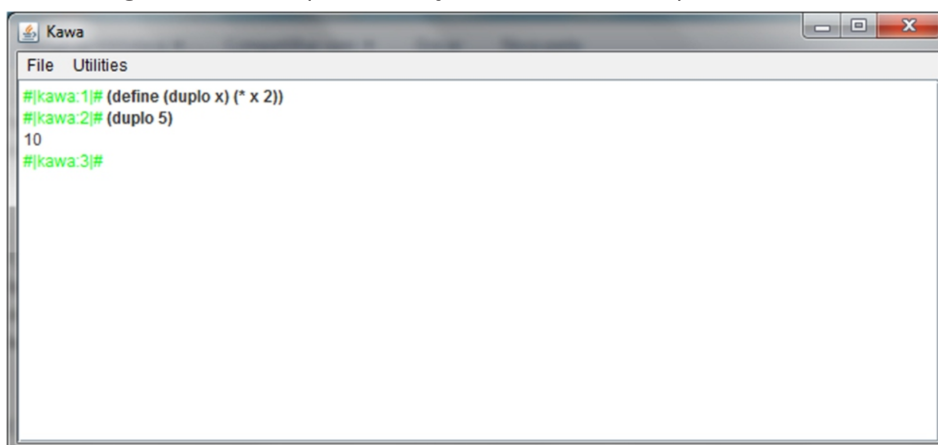
Um programa *Scheme* é uma sequência de definições e aplicações de funções. As definições e aplicações de funções são expressões escritas entre parênteses. A aplicação de uma função f a um conjunto de argumentos a1, a2, ..., an é representada por uma expressão, entre parênteses, em notação pré-fixada: (f a1 a2 ... an).

A definição de uma função inicia pelo símbolo define, seguido do nome da função, seus argumentos e, finalmente, pelo corpo da função. Por exemplo: definimos a função denominada *duplo* que, a partir de um parâmetro x, retorna o valor de x multiplicado por 2:

(define (duplo x) (2 x))*

Para utilizarmos a função, ou seja, executarmos o programa, aplicando a função, basta digitarmos no *prompt* do interpretador *Scheme* a referida função, seguida da aplicação da mesma. Por exemplo: se digitarmos (duplo 5) a resposta será 10, como mostra a Figura 11.

Figura 11 – Exemplo de Função *Scheme* no Interpretador *Kawa*



Fonte: Elaborado pelos autores

Vamos escrever um exemplo de função que retorne o quadrado de um número:

(define (quadrado numero)(numero numero))*

Após a definição, a função já pode ser aplicada, escrevendo o nome da função e o valor desejado:

(quadrado 5)

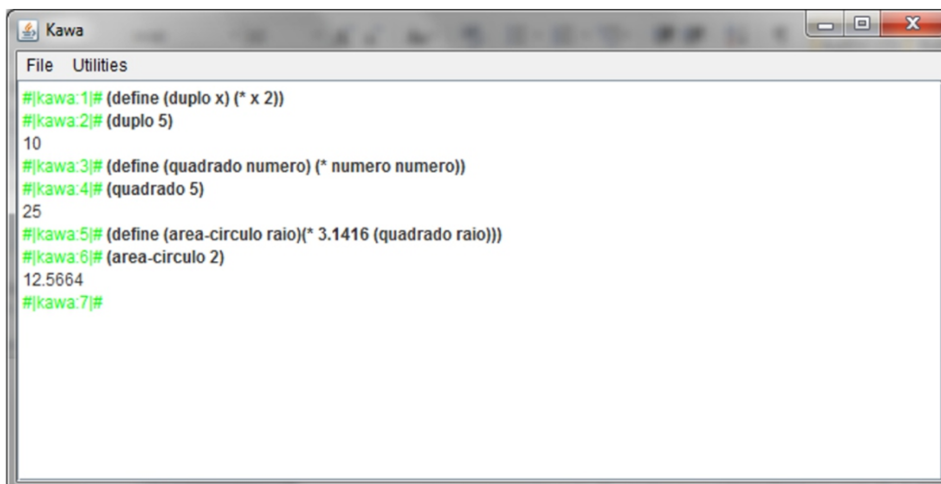
A função quadrado pode ser usada na definição de outras funções, por exemplo:

(define (area-circulo raio)(3.1416 (quadrado raio)))*

(area-circulo 2)

A Figura 12 apresenta estes exemplos criados e executados no interpretador *Kawa.*

Figura 12 – Exemplos de Funções *Scheme* no Interpretador *Kawa*



Fonte: Elaborado pelos autores

3.2.3 Tipos de Dados

A linguagem *Scheme* trabalha com objetos simples (átomos): números, *strings,* objetos *booleanos* e símbolos e com objetos compostos: pares e listas. Os números podem ser inteiros ou reais, positivos ou negativos: 0 20 5. 5.0 3.1416 -5.0 -5.3 *Strings* são objetos textuais delimitados por aspas: "Exemplo de *string*".

O tipo booleano pode assumir os valores #t (true) e #f (*false*): a avaliação da expressão (= 2 3) resulta #f. A avaliação da expressão (< 3 4) resulta #t.

Os símbolos são nomes usados para referenciar construções: objetos e funções e devem iniciar por uma letra e não devem conter parênteses ou aspas. Por exemplo:

(define a 50) – amarra o símbolo a ao inteiro 50

(define (operacao x y)(+ x y) – amarra o símbolo operacao ao código (+ x y)

Capítulo 3 – Exercícios

1) Escreva a definição de uma função que, a partir de dois números, escreva qual é o maior?

2) Escreva a definição de uma função que, a partir de dois números, escreva se são ou não múltiplos?

3) Um supermercado deseja reajustar os preços de seus produtos, para mais ou para menos, de acordo com os critérios mostrados na tabela abaixo. Escrever uma definição de função que, a partir dos valores do preço atual e da venda média mensal do produto, calcule qual o preço reajustado?

Venda média mensal	Preço Atual	% de Aumento	% de Diminuição
< 500	< R$30,00	10	-
>= 500 e < 1200	>= R$30,00 e < R$80,00	15	-
>= 1200	<= R$80,00	-	20

4) Escreva a definição de funções que, a partir de um parâmetro representando a o tempo necessário para fabricar um produto expresso em segundos, mostre-o expresso em horas, minutos e segundos (dica: utilize as primitivas *quotient* para extrair a parte inteira de uma divisão e *remainder* para extrair o resto da divisão).

5) Escreva a definição de uma função que conte, a partir do número informado como parâmetro, até chegar a zero. Enquanto não chegar a zero, mostrar o número correspondente. Quando a contagem chegar a zero, mostrar a mensagem "Encerrado".

6) Escreva a definição de funções que, a partir de um parâmetro representando a idade de uma pessoa expressa em dias, mostre-a expressa em anos, meses e dias (dica: utilize as primitivas *quotient* para extrair a parte inteira de uma divisão e *remainder* para extrair o resto da divisão).

7) Escreva a definição de uma função que, a partir de três parâmetros (dois números e um caracter representando a operação aritmética desejada + - * /) calcula o resultado da operação aritmética (dica: utilize a primitiva *cond*).

8) Escreva a definição de uma função que, a partir de um parâmetro numérico representando o salário bruto de uma pessoa, calcule o valor a ser pago referente ao INSS, de acordo com as seguintes faixas de valores: até R$1200,00 2% do salário bruto, acima de R$1200,00 até R$2500,00 5% do salário bruto e acim de R$2500,00 8% do salário bruto (dica: utilize a primitiva *cond*).

9) Escreva a definição de uma função que calcule o fatorial de um número.

Capitulo 4
PARADIGMA DE ORIENTAÇÃO A EVENTOS – *LAZARUS*

Neste capítulo vamos estudar o Paradigma de Orientação a Eventos, muito utilizado em linguagens de programação que permitem a construção de interfaces gráficas baseadas em janelas, tais como as do Sistema *Operacional Microsoft Windows* (SILVEIRA, 2006). As atividades de exemplo prático foram elaboradas com a IDE *Lazarus*, similar ao *Delphi*, utilizando a linguagem de programação *Object Pascal*. Apesar desta linguagem utilizar o paradigma de Orientação a Objetos, neste capítulo vamos focar o estudo do paradigma orientado a eventos.

4.1 Conceitos do Paradigma de Orientação a Eventos

No Paradigma de Orientação a Eventos utilizamos IDEs que permitem o desenvolvimento de *software* orientado pelo desenho de formulários (janelas ou *forms*), contendo diferentes componentes visuais, também chamados de controles (tais como menus e botões, por exemplo). O desenvolvimento é conhecido como RAD (*Rapid Application Development*) (SILVEIRA, 2006).

Os controles, ou componentes possuem algumas características: são pré-desenvolvidos; possuem propriedades (atributos) que podem ser alteradas em tempo de desenvolvimento e/ou de execução; possuem métodos associados e respondem a eventos (SILVEIRA, 2006).

Os eventos são mensagens que cada componente pode responde, tendo associado a eles um procedimento de evento. Alguns exemplos de eventos compreendem:

- Clicar com o botão esquerdo do *mouse*;
- Clicar com o botão direito do *mouse*;
- Pressionar uma tecla sobre um controle;

- Passar o cursor do *mouse* sobre um controle;
- Duplo clique sobre um controle;
- Entrar ou sair de um formulário (janela).

4.2 Componentes de um Programa

Um programa, neste contexto, é um Projeto (*Project*) contendo formulários (*form*) e código-fonte (*units* em *Object Pascal*).

Para nomearmos os controles devemos seguir uma nomenclatura. Utilizaremos, neste livro, a seguinte nomenclatura para os controles:

- Primeiro utilizaremos letras minúsculas (preferencialmente sem vogais) para identificar o tipo do componente (controle) e o restante do nome para identificar a função do mesmo. Por exemplo:
    - btnSair (Botão Sair)
    - btnRelVendas (Botão Relatório de Vendas).

Para iniciarmos a criação de um programa, utilizaremos os recursos da IDE *Lazarus*. Na janela *Form* Designer vamos inserir os componentes (controles) da interface do nosso programa. Na janela P*rogram Editor* vamos inserir o código-fonte, escrito em *Object Pascal*.

Algumas características importantes sobre o código-fonte que escreveremos na linguagem de programação *Object Pascal*:

- Os comentários devem ficar entre chaves {}, entre parênteses e asteriscos (* *) ou iniciarem por duas barras //;
- As instruções são separadas por ponto-e-vírgula. Veja, por exemplo: *ShowMessage*('Teste'); é uma instrução que mostra uma caixa de mensagem como a mensagem Teste;

Vamos programar de acordo com a ocorrência de eventos. Seguem alguns exemplos de eventos que podemos utilizar:

OnEnter – o componente recebe o foco;

OnClick – o componente é acionado (clicado);

OnDlbClick – duplo-clique no componente;

OnExit – o componente perde o foco;

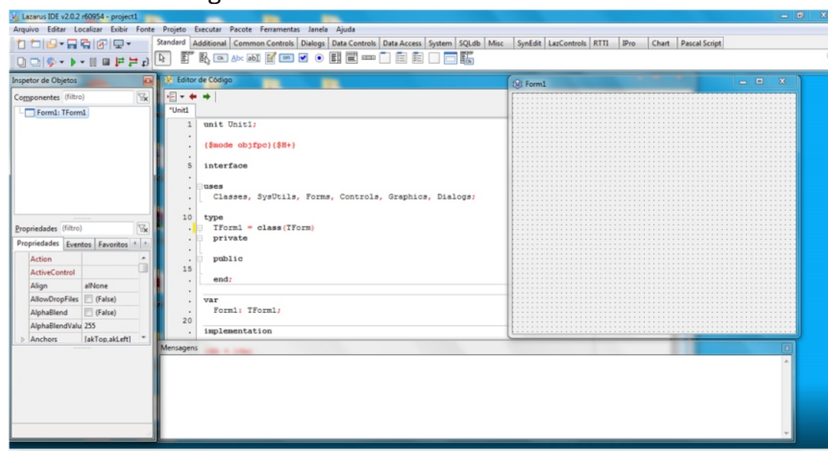
OnKeyPress– uma tela é pressionada e solta.

Além dos eventos, também podemos utilizar métodos previamente definidos na linguagem *Object Pascal*, tais como:

- *Hide* – esconde um componente;
- *Show* – mostra um componente.

A Figura 13 apresenta a tela de abertura da IDE *Lazarus*.

Figura 13 – Tela de Abertura do *Lazarus*



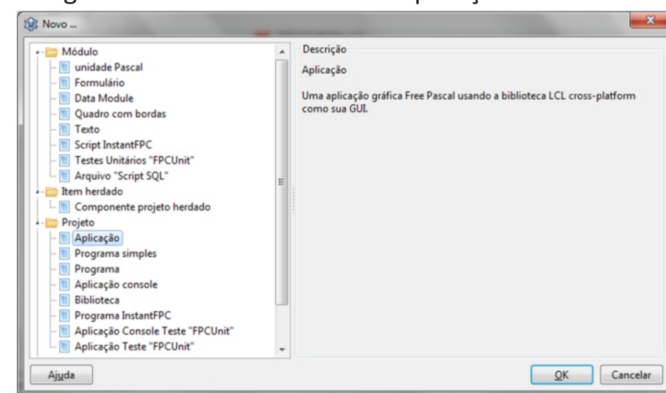
Fonte: Os autores

Na Figura 13 podemos observar, do lado esquerdo, o *Inspetor de Objetos*, que apresenta os componentes do nosso programa (aplicação). Por enquanto temos um formulário, denominado *Form1*, pertencente à classe *TForm*. Abaixo do *Inspetor de Objetos* temos a janela contendo as propriedades (características ou atributos) dos componentes. Na parte central da tela temos o *Editor de Código* (onde iremos escrever o código dos métodos, utilizando a linguagem de programação *Object Pascal.*

4.3 Exemplos de Programação utilizando o Paradigma de Orientação a Eventos

Vamos iniciar nossos exemplos criando uma nova aplicação na IDE *Lazarus*, selecionado a opção *Novo...* no menu *Arquivo* e escolhendo *Aplicação*, como mostra a Figura 14. Após escolhermos a opção *Aplicação* será mostrada a tela da Figura 15.

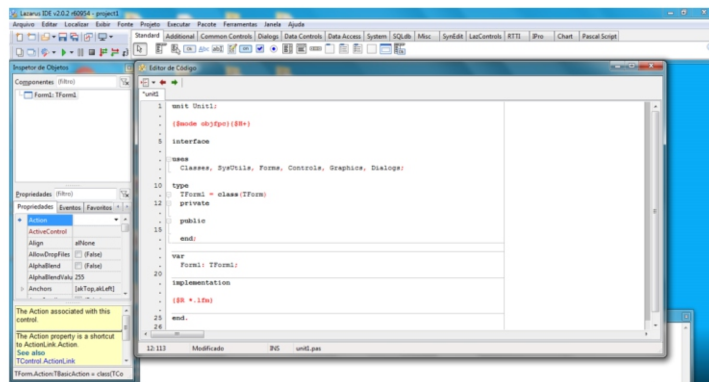
Figura 14: Criando uma Nova Aplicação no Lazarus



Fonte: Os autores

Na Figura 15 temos, do lado direito da tela, as janelas *Inspetor de Objetos* e *Propriedades*. No centro da tela temos o *Editor de Código*.

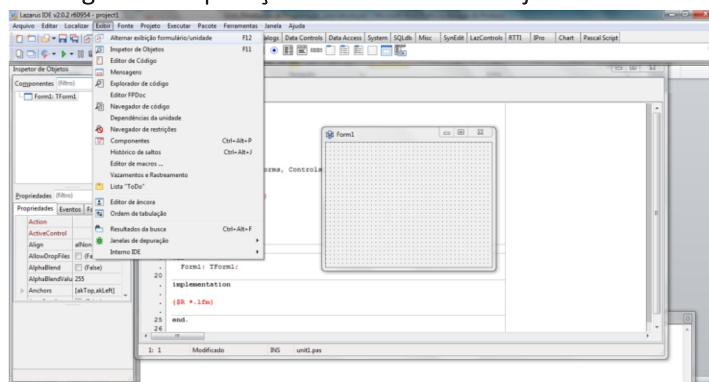
Figura 15: Aplicação no *Lazarus*



Fonte: Os autores

Para exibir o formulário (*form*), janela onde iremos inserir os objetos da interface da nossa aplicação, devemos selecionar a opção *Alternar exibição de formulário/unidade* ou pressionarmos a tecla F12. Será, mostrada, então, a janela do *form*, como mostra a Figura 16.

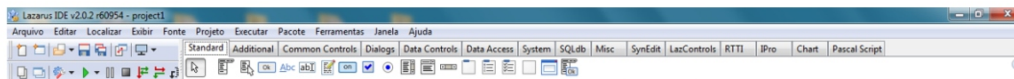
Figura 16: Aplicação no *Lazarus* com a janela *Form*



Fonte: Os autores

Vamos fazer um exemplo utilizando alguns controles básicos, disponíveis na aba *Standard*, mostrada na Figura 17.

Figura 17: Aba *Standard*

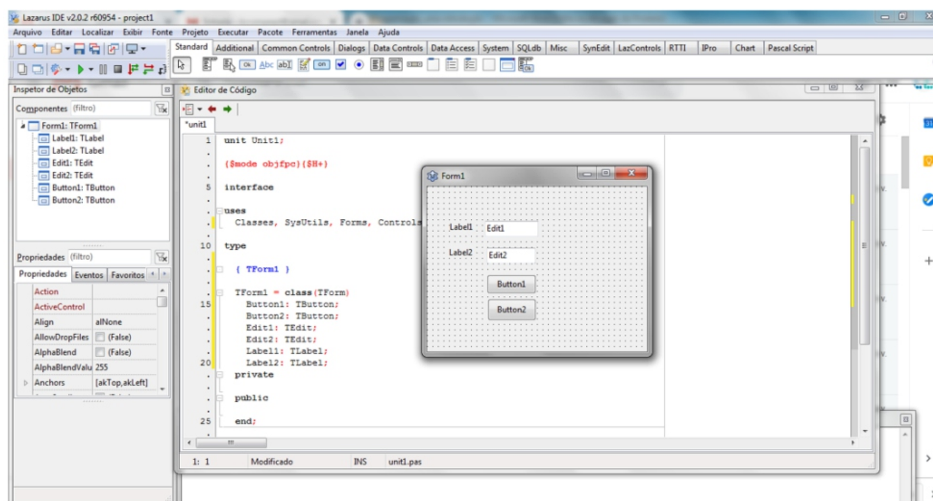


Fonte: Os autores

Vamos inserir no formulário dois controles do tipo *TLabel*, dois controles do tipo *TEdit* e dois controles do tipo *TButton*. O controle TLabel é utilizado para mostrar uma mensagem na tela (representa um rótulo). O controle T*Edit* é utilizado para a entrada de dados (representa uma caixa de texto ou uma caixa editável) e o controle *TButton* representa um botão. Para inserir os controles basta

selecionar o tipo de controle desejado na aba *Standard* e desenhá-los (arrastando e soltando o *mouse*) no *form*. A Figura 18 apresenta estes controles inseridos no formulário.

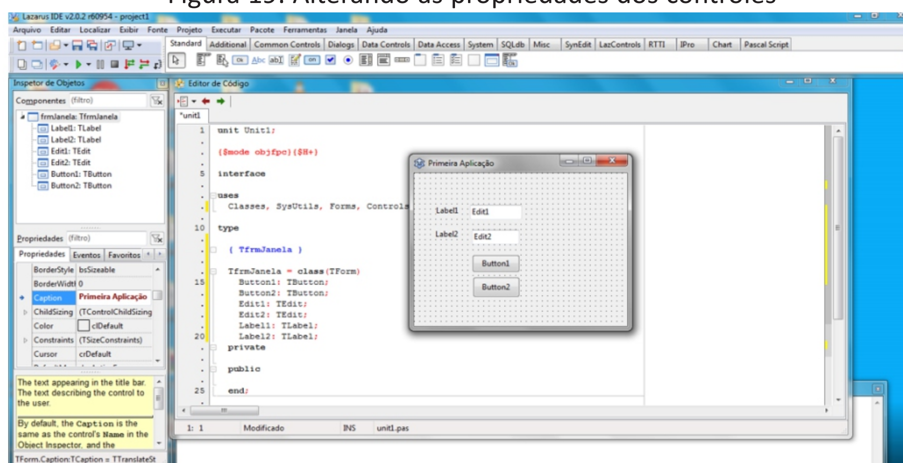
Figura 18: Controles inseridos no formulário



Fonte: os autores

Agora vamos alterar as propriedades dos controles, incluindo as propriedades do formulário (janela do nosso programa), para que possamos modificar alguns atributos visuais e, também, seus nomes (como se fossem nomes de variáveis, para que possamos utilizar os controles no nosso código-fonte). Para alterar as propriedades de um controle devemos selecioná-lo com o mouse e selecionar a(s) propriedade(s) desejada no lado esquerdo da tela, na caixa Propriedades. Vamos iniciar selecionando o *form* e alterando as propriedades *Name* (nome do formulário) e *Caption* (título que aparecerá na barra de título da janela). Vamos alterar a propriedade *Name* para *frmJanela* e a propriedade *Caption* para *Primeira Aplicação*. A Figura 19 apresenta a tela já com as modificações realizadas nas propriedades, incluindo a modificação do nome do formulário na janela do código-fonte.

Figura 19: Alterando as propriedades dos controles

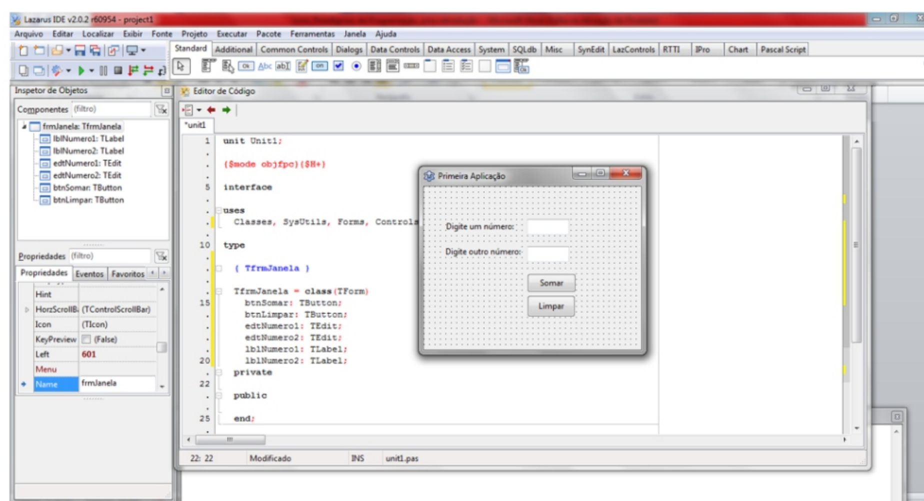


Fonte: os autores

Vamos alterar, também, as seguintes propriedades dos demais controles constantes do formulário. O resultado destas alterações é apresentado na Figura 20:

- Primeiro *TLabel*: alterar a propriedade *Name* para *lblNumero1* e a propriedade *Caption* para *Digite um número;*
- Segundo *TLabel*: alterar a propriedade *Name* para *lblNumero2* e a propriedade *Caption* para *Digite outro número*;
- Primeiro *TEdit*: alterar a propriedade *Name* para *edtNumero1* e deixar a propriedade T*ext em branco*;
- Segundo *TEdit*: alterar a propriedade *Name* para *edtNumero2* e deixar a propriedade *Text* em branco;
- Primeiro *TButton*: alterar a propriedade *Name* para *btnSomar* e a propriedade *Caption* para *Somar*;
- Segundo *TButton*: alterar a propriedade *Name* para *btnLimpar* e a propriedade *Caption* para *Limpar*.

Figura 20: Propriedades atualizadas



Fonte: os autores

Agora vamos implementar o código-fonte dos botões *Somar* e *Limpar*. Para tanto vamos utilizar o evento *OnClick*, ou seja, quando o usuário clicar nos botões o código implementado deverá ser executado. Vamos, então, dar um duplo clique no botão *btnSomar* e será aberta a janela de código do método *btnSomarClick*, que corresponde ao evento clique com o *mouse* sobre o botão, como mostra a Figura 21.

Figura 21: Janela do código-fonte

Fonte: os autores

Exercícios do Capítulo 4

1. Marque V para Verdadeiro e F para Falso, considerando que as afirmações indicam eventos de acordo com o paradigma estudado neste capítulo:

( ) Clicar com o botão esquerdo do *mouse*

( ) Acessar uma tabela no banco de dados

( ) Pressionar uma tecla sobre um controle

( ) Mostrar um componente (controle) na tela

( ) Entrar ou sair de um formulário (janela)

2. Complete as lacunas:

Os controles, também podem ser chamados de componentes ou objetos. Os controles possuem _________________ que são as suas características e ______________ que permitem a inserção de código-fonte associado à ocorrência de ______________________________.

Capitulo 5

PROGRAMAÇÃO CONCORRENTE EM JAVA

Neste capítulo vamos estudar o paradigma de programação concorrente, utilizando *threads* em *Java*. Os *threads* são fluxos de execução concorrente. Os exemplos práticos serão desenvolvidos utilizando a linguagem de programação *Java* e a IDE *NetBeans*.

5.1 Conceitos de Concorrência em Java

A programação concorrente, em *Java*, é realizada por meio de *threads*, que são fluxos de execução concorrentes. Este é o conceito de *multithreading,* onde o programador especifica quais aplicativos contêm fluxos de execução (*threads*), cada fluxo designando uma parte de um programa que pode ser executado concorrentemente com outros *threads*. Analisando o fluxo de um programa tradicional, temos *entrada*, *processamento* e *saíd*a. O processamento depende dos dados de entrada e, a saída, depende dos resultados do processamento. Sendo assim, não é possível realizar nenhuma dessas atividades concorrentemente (simultaneamente). Isto significa que o programador tem que entender o que pode ou não ser executado de forma concorrente em seu programa (DEITEL; DEITEL, 2017).

Um exemplo de concorrência em Java é o coletor de lixo, que libera a memória ocupada por objetos que não estão sendo mais utilizados. O coletor de lixo do Java *(garbage collector)* executa como um *thread* de baixa prioridade, ou seja, o coletor de lixo executa quando o tempo do processador está disponível e quando não há *threads* executáveis de alta prioridade (DEITEL; DEITEL, 2017).

Para utilizarmos a programação *multithread* em Java, precisamos da classe *Thread*, que funciona como um modelo de fluxos de execução concorrentes. A classe *Thread* possui vários construtores. O construtor *Public Thread(String*

threadName) constrói um objeto *Thread* cujo nome é *ThreadName.* O construtor *public Thread()* constrói um objeto cujo nome é "*Thread-*" concatenado com um número (*Thread-1*, *Thread-2*, etc.). Vamos utilizar estes construtores em nossos exemplos práticos.

5.1.1 Métodos de um Thread

O código que faz o trabalho "de verdade" de um *thread* é colocado no seu método *run*. Um programa dispara a execução de um *thread* chamando o método *start* do *thread* que, por sua vez, chama o método *run* (DEITEL; DEITEL, 2017).

Depois de *start* carregar o *thread,* a execução retorna para o método chamador imediatamente. O método chamador então executa concorrentemente com o *thread* carregado. Por exemplo, o método chamador pode ser a rotina *main* do programa. Além do método *start*, existem outros métodos que podemos utilizar para controlar a programação *multithread* (DEITEL; DEITEL, 2017):

- O método *sleep* é chamado com um argumento que especifica quanto tempo o *thread* atualmente em execução deve dormir (em milissegundos). Enquanto um *thread* "dorme" ele não disputa o processador, então outros *threads* podem executar. Isso pode dar oportunidade para *threads* de prioridade mais baixa serem executados, como é o caso do *Garbage Collector*;
- o método *getName* retorna o nome do *Thread*;
- o método *toString* retorna uma *string* contendo o nome do *thread*, prioridade e seu *ThreadGroup*;
- o método *currentThread* retorna uma referência ao *Thread* atualmente em execução.

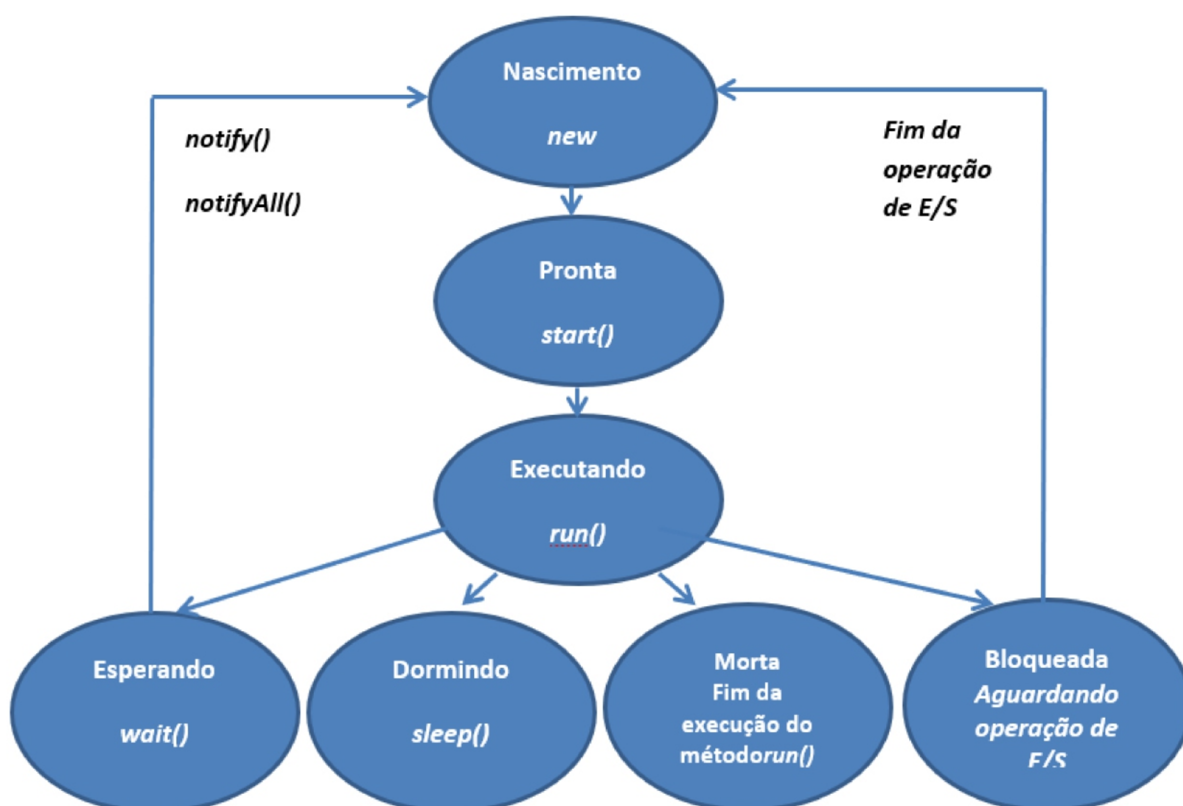
5.1.2 Estados de um Thread

Os estados de um *Thread* são: nascimento, pronto, morto, executando, bloqueado, adormecido e em espera, apresentados graficamente na Figura 22 (DEITEL; DEITEL, 2017):

- *nascimento*: *thread* que acaba de ser criado e permanece neste estado até o método *start* ser chamado;
- *pronto*: ao chamar o método *start* o *thread* passa para o estado pronto (executável). O *thread* pronto de prioridade mais alta entra no estado executando quando o sistema aloca um processador para o *thread*;

- *morto*: um *thread* entra no estado morto quando seu método *run* completa ou termina por alguma razão;
- *bloqueado*: um *thread* pode entrar no estado bloqueado quando precisa realizar alguma operação de entrada ou saída (E/S). Nesse caso um *thread* bloqueado torna-se pronto quando a E/S pela qual está esperando é concluída. Um *thread* bloqueado não pode utilizar um processador, mesmo se algum estiver disponível;
- *adormecido*: Quando um método *sleep* é chamado em um *thread* executando, esse *thread* entra no estado adormecido. Um *thread* adormecido torna-se pronto depois que o tempo designado para dormir expira; um *thread* adormecido não pode utilizar um processador, mesmo se algum estiver disponível,

Figura 22: Estados de um *Thread*



Fonte: Adaptado de DEITEL; DEITEL (2017)

5.1.3 Prioridades de um thread

Todo *thread* Java tem uma prioridade definida no intervalo de *Thread.MIN_PRIORITY (1)* e *Thread.MAX_PRIORITY(10)*. Por *default* cada *thread* recebe prioridade *Thread.NORM_PRIORITY (5).* Cada novo *thread* herda a prioridade do *thread* que o cria (DEITEL; DEITEL, 2017).

Algumas plataformas Java suportam um conceito denominado fracionamento de tempo *(time slicing)* e algumas não. Sem fracionamento de tempo, cada *thread* em um conjunto de *threads* de igual prioridade executa até sua conclusão (a menos que o *thread* deixe o estado de execução e entre no estado de espera, adormecido ou bloqueado). Com o fracionamento de tempo, cada *thread* recebe um breve período de tempo do processador (chamado de *quantum*) durante o qual esse *thread* pode ser executado. Na conclusão do *quantum,* mesmo se esse thread não tiver terminado a execução, o processador é tirado desse *thread* e alocado ao próximo *thread* de igual prioridade (DEITEL; DEITEL, 2017).

5.1.3 Exemplo Prático

Vamos construir um exemplo que demonstra técnicas de *threading* básicas:

- Criação de uma classe derivada de *Thread*; construção de uma *thread* e utilização do método *sleep*;
- Cada *thread* exibe seu nome depois de dormir por uma quantidade aleatória de tempo entre 0 e 15 segundos;
- O programa consiste de duas classes: *TesteThread* e *ImprimirThread*;
- A classe *ImprimirThread* (que herda de *Thread* para que cada objeto da classe possa executar em paralelo) possui uma variável *sleepTime*, um construtor e um método *run*;
- Cada objeto *ImprimirThread* dorme pela quantidade de tempo especificada por *sleepTime* e, então, envia seu nome para a tela.

Analisando o exemplo tem-se que: quando o método *start* de um *ImprimirThread* é invocado, o objeto *ImprimirThread* entra no estado pronto. Quando o sistema aloca um processador ao objeto *ImprimirThread* ele entra no estado executando e seu método *run* inicia a execução. O código-fonte completo é apresentado na Figura 23 na próxima página.

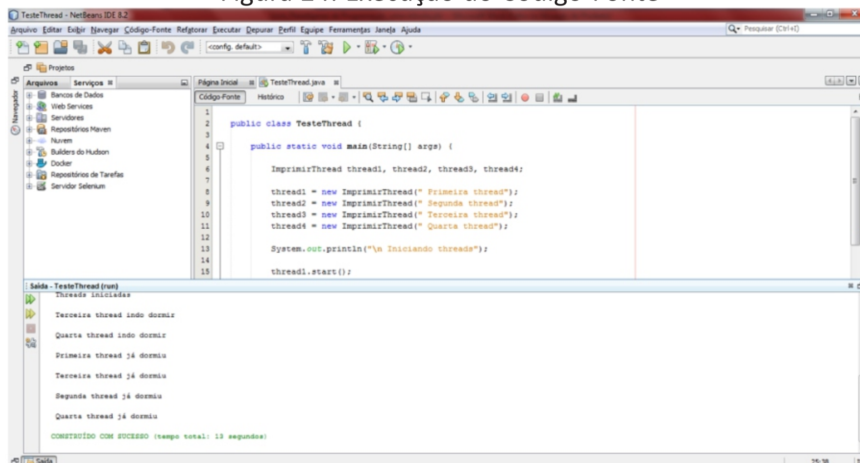
Figura 23 – Exemplo de Código-Fonte em Java

```
public class TesteThread {

    public static void main(String[] args) {

        ImprimirThread thread1, thread2, thread3, thread4;
          thread1 = new ImprimirThread(" Primeira thread");
          thread2 = new ImprimirThread(" Segunda thread");
          thread3 = new ImprimirThread(" Terceira thread");
          thread4 = new ImprimirThread(" Quarta thread");

        System.out.println("\n Iniciando threads");
          thread1.start();
          thread2.start();
          thread3.start();
          thread4.start();
        System.out.println("\n Threads iniciadas \n");
    }
}
class ImprimirThread extends Thread {
    private int tempoEspera;
    //construtor ImprimirThread atribui um nome ao thread
    public ImprimirThread(String name)
    {
        super(name);
        //dorme entre 0 e 15 segundos
        tempoEspera=(int) (Math.random() * 15000);
        System.out.println("\n Nome da thread:" + getName() + " dorme:" +
        tempoEspera);
    }
    //executa o thread
    public void run()
    {
        //coloca thread para dormir por um intervalo aleatório
        try
        {
            System.out.print(getName() + " indo dormir \n");
            Thread.sleep(tempoEspera);
        }   catch (InterruptedException exception) {
            System.out.println(exception.toString());
        }
        //Imprime o nome da thread
        System.out.println(getName() + " já domriu \n");
        }
}
```

Fonte: Adaptado de DEITEL; DEITEL (2017)

A Figura 24 mostra o resultado da execução deste exemplo.

Figura 24: Execução do Código-Fonte



Fonte: Elaborado pelos autores

5.1.4 Segundo Exemplo Prático

Vamos modificar o código da classe *ImprimirThread* do exemplo anterior, incluindo laços *for* para escrever valores na tela após o *sleep*; incluir um segundo *sleep* e um segundo laço *for*. Destacamos as partes que foram inseridas no código-fonte em negrito mostrado na Figura 25.

Figura 25: Código-Fonte Alterado

```
//executa o thread
    public void run()
    {
        //coloca thread para dormir por um intervalo aleatório
        try
        {
            System.out.print(getName() + " indo dormir \n");
            Thread.sleep(tempoEspera);
            for (int i=0;i<=5;i++) {
                System.out.print(getName() + " valor de I:" + i);
            }
            Thread.sleep(tempoEspera);
            for (int j=0;j<=10;j++) {
                System.out.print(getName() + " valor de J:" + j);
            }
        }
        catch (InterruptedException exception) {
            System.out.println(exception.toString());
        }
        //Imprime o nome da thread
        System.out.println(getName() + " já dormiu \n");
    }
```

Fonte: Os autores (adaptado de DEITEL; DEITEL (2017))

Execute o código-fonte com esta modificação e veja o resultado. Cada *thread* irá "contar" até 5 e, posteriormente até 10, de acordo com o seu processamento, definido pelo tempo que cada uma irá dormir.

5.2 Sincronização de Threads

Nesta seção vamos detalhar um exemplo prático baseado em Deitel; Deitel (2017), compreendendo a sincronização de *threads*. Dados compartilhados podem se tornar corrompidos se não sincronizarmos o acesso entre múltiplos *threads*. Em um relacionamento produtor/consumidor, um *thread* produtor chamando um método *produce* pode ver o que o *thread* consumidor não leu na última mensagem de uma região compartilhada da memória denominada *buffer*, então o *thread* produtor chamará *wait*. Quando um *thread* consumidor ler a mensagem, ele chamará *notify* para permitir que um produtor em espera prossiga. Quando um thread consumidor entra no monitor e descobre o *buffer* vazio, ele chama *wait* (DEITEL; DEITEL, 2017).

Um produtor que encontra o *buffer* vazio, grava no *buffer* e chama *notify*, de modo que um consumidor em espera possa prosseguir. O exemplo a seguir mostra a utilização de um dado compartilhado (variável *int sharedInt*). Os métodos produtor e consumidor acessam esta área de memória compartilhada sem nenhuma sincronização. Uma vez que os *threads* não são sincronizados, os dados podem ser perdidos se o produtor colocar novos dados no *buffer* antes de o consumidor consumir os dados anteriores e os dados podem ser "duplicados", se o consumidor consumir os dados novamente antes que o produtor produza o próximo item (DEITEL; DEITEL, 2017).

O exemplo prático compreende a criação de um produtor de números intreiros (*ProduceInteger*) e um consumidor destes números (*ConsumeInteger*). Os métodos utilizam uma área de memória compartilhada (*sharedInt*) sem sincronização. O código-fonte deste exemplo é apresentado na Figura 26.

Figura 26: *Threads* sem sincronização

```
public class SharedCell {
public static void main(String[] args) {
   HoldIntegerUnsynchronized h = new
HoldIntegerUnsynchronized();
   ProduceInteger p = new ProduceInteger(h);
   ConsumeInteger c = new ConsumeInteger(h);
   p.start();
   c.start();
      }
```

Continua

Continuação

```
}
public class ProduceInteger extends Thread {
    private HoldIntegerUnsynchronized pHold;
public ProduceInteger(HoldIntegerUnsynchronized h)
{
    super("ProduceInteger");
    pHold=h;
}
public void run()
{
    for (int count=1;count<=10;count++) {
        try {
            Thread.sleep( (int) (Math.random() * 3000));
        }
        catch (InterruptedException e) {
            System.out.println(e.toString());
    }
    pHold.setSharedInt(count);
}
System.out.println(getName()+ " finalizou a produção de valores" +
"\n terminando " + getName());
}
}
public class ConsumeInteger extends Thread {
    private HoldIntegerUnsynchronized cHold;
    public ConsumeInteger(HoldIntegerUnsynchronized h)
    {
        super("ConsumeInteger");
        cHold = h;
    }
    public void run() {
        int val, sum = 0;
        do {
            try {
                Thread.sleep( (int) (Math.random() * 3000));
            }
            catch (InterruptedException e) {
                System.out.println(e.toString());
            }
            val=cHold.getSharedInt();
            sum+=val;
        } while (val !=10);
        System.out.println(getName() + " recuperando valores totais:" + sum +
        "\n Terminando " + getName());
        }
}
```

Continua

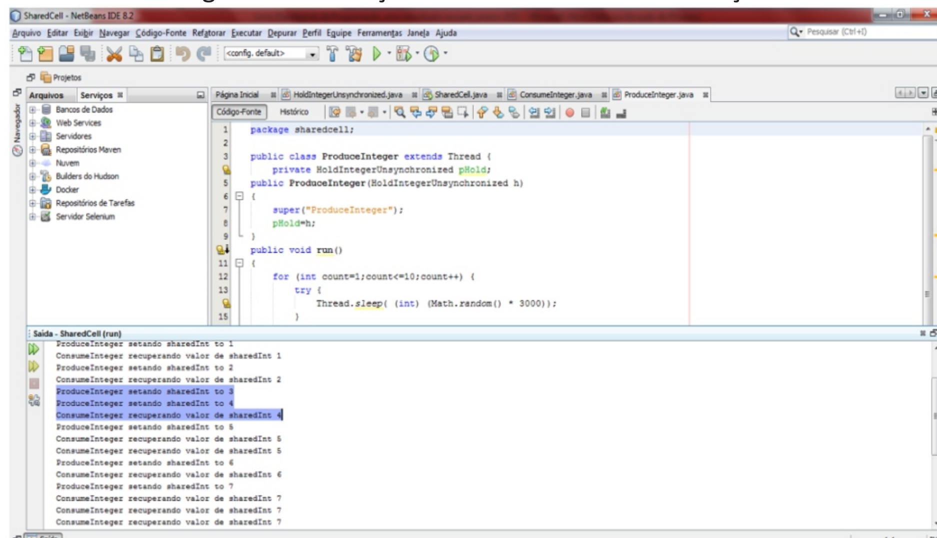
Continuação

```
public class HoldIntegerUnsynchronized {
    private int sharedInt = -1;
    public void setSharedInt(int val) {
        System.out.println(Thread.currentThread().getName() +
        " setando sharedInt para " + val);
        sharedInt = val;
    }
    public int getSharedInt()
    {
        System.out.println(Thread.currentThread().getName() +
        " recuperando valor de sharedInt " + sharedInt);
        return sharedInt;
    }
}
```

Fonte: DEITEL; DEITEL (2017)

A Figura 27 mostra o resultado de uma das execuções deste código-fonte. Verifica-se nesta execução que um dos valores produzidos pelo método *ProduceInteger* (o número 3) não foi consumido pois o método produtor gerou dois valores (3 e 4) até que o método *ConsumeInteger* acessasse a área compartilhada. Isto se deve à falta de sincronização.

Figura 27: Execução de *Thread* sem Sincronização



Fonte: Elaborado pelos autores

O próximo exemplo demonstra um produtor e um consumidor acessando uma célula de memória compartilhada com sincronização, de modo que o consumidor consome somente depois do produtor produzir um valor.

A variável *writeable* (variável de condição do monitor) é utilizada pelo método *setSharedInt* para determinar se o *thread* que o chama pode gravar na posição

de memória compartilhada e é utilizada pelo método *getSharedInt* para determinar se o *thread* que o chama pode ler da posição da memória compartilhada. O exemplo do código-fonte apresentado na Figura 28 é adaptado do exemplo anterior de Deitel & Deitel (2017).

Figura 28: *Threads* com sincronização

```
public class ThreadSincronizada {
    public static void main(String[] args) {
        NumeroSincronizado h = new NumeroSincronizado();

        ProduzirInteiro p = new ProduzirInteiro(h);
        ConsumirInteiro c = new ConsumirInteiro(h);

        p.start();
        c.start();
    }
}
public class ConsumirInteiro extends Thread {
    private NumeroSincronizado cHold;

    public ConsumirInteiro(NumeroSincronizado h) {
        super("ConsumirInteiro");
        cHold=h;
    }

    public void run() {
        int val, sum=0;
        do {
            //dorme por um intervalo aleatório
            try {
                Thread.sleep( (int) (Math.random() * 3000));
            }
            catch (InterruptedException e) {
                System.err.println(e.toString());
            }
            val=cHold.getSharedInt();
            sum+=val;
        } while (val !=10);

        System.err.println(getName() + " recuperados valors totalizando: " + sum);
    }
}
public class ProduzirInteiro extends Thread {
    private NumeroSincronizado pHold;

    public ProduzirInteiro(NumeroSincronizado h) {
        super("ProduzirInteiro");
        pHold=h;
    }
```

Continua

Continuação

```
        public void run() {
            for (int count=1; count<=10; count++) {
                try {
                     Thread.sleep( (int) (Math.random() * 3000));
                }
                catch (InterruptedException e) {
                    System.err.println(e.toString());
                }
                pHold.setSharedInt(count);
            }
            System.err.println(getName() + " finalizou a produção de valores ");
        }
    }
    public class NumeroSincronizado {
        private int sharedInt = -1;
        private boolean writeable = true;
        public synchronized void setSharedInt(int val) {
            while (!writeable) {
                try {
                    wait();
                }
                catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        System.err.println(Thread.currentThread().getName()  +  "  setando
valor compartilhado para"
+ val);
        sharedInt=val;
        writeable=false;
        notify(); //diz para um thread em espera tornar-se pronto
        }
        public synchronized int getSharedInt() {
             while (writeable) {
                try {
                    wait();
                }
                catch (InterruptedException e) {
                    e.printStackTrace();
            }
        }
        writeable=true;
        notify();
        System.err.println(Thread.currentThread().getName() + " recuperando
    valor compartilhado "
    + sharedInt);
            return sharedInt;
    }
```

Fonte: Adaptado de DEITEL; DEITEL (2017)

A linguagem de programação Java utiliza monitores para executar sincronizações (todo objeto com métodos *synchronized* é um monitor). O monitor permite que um *thread* de cada vez execute um método *synchronized* sobre o objeto; isso é realizado quando o método *synchronized* é invocado (também conhecido como obter o bloqueio). Se existirem vários métodos *synchronized*, somente um método *synchronized* pode estar atuando em um objeto de cada vez; todos os outros métodos tentando invocar métodos *synchronized* devem esperar (DEITEL; DEITEL, 2017).

Quando um método *synchronized* termina de executar, o bloqueio sobre o objeto é liberado e o monitor permite que o *thread* de prioridade mais alta tente invocar um método *synchronized* para prosseguir. Um *thread* sendo executado em um método *synchronized* pode determinar que não pode prosseguir chamando o método *wait*. Isto tira o *thread* da disputa pelo processador e da disputa pelo objeto monitor. O *thread* entra, então, no estado de espera enquanto os outros *threads* tentam entrar no objeto monitor (DEITEL; DEITEL, 2017).

Quando um *thread* que executa um método *synchronized* se completa, o *thread* pode notificar (*notify*) um *thread* em espera para ele se tornar novamente pronto, de modo que ele possa tentar obter o bloqueio sobre o objeto monitor novamente e ser executado. O *notify* atua como um sinal para o *thread* em espera de que a condição que o *thread* estava esperando agora está satisfeita. Os métodos *wait* e *notify* são herdados por todas as classes da classe *Objec*t, o que significa que qualquer objeto tem o potencial de ser um monitor. Os threads no estado de espera por um objeto monitor devem, em algum momento, ser acordados explicitamente com um *notify* ou *interrupt*. Caso contrário o *thread* esperará eternamente, o que pode causar *deadlock* (DEITEL; DEITEL, 2017).

Deve-se, para cada chamada *wait* ter uma chamada *notify* (ou *notifyAll*). Os objetos monitores mantêm uma lista de todos os *threads* esperando para entrar no objeto monitor para executar métodos *synchronized* (um *thread* é inserido na lista e espera pelo objeto se chamar um método *synchronized* enquanto um outro *thread* já está sendo executado). Um *thread* também é inserido na lista se chamar *wait* enquanto está operando dentro do objeto (DEITEL; DEITEL, 2017).

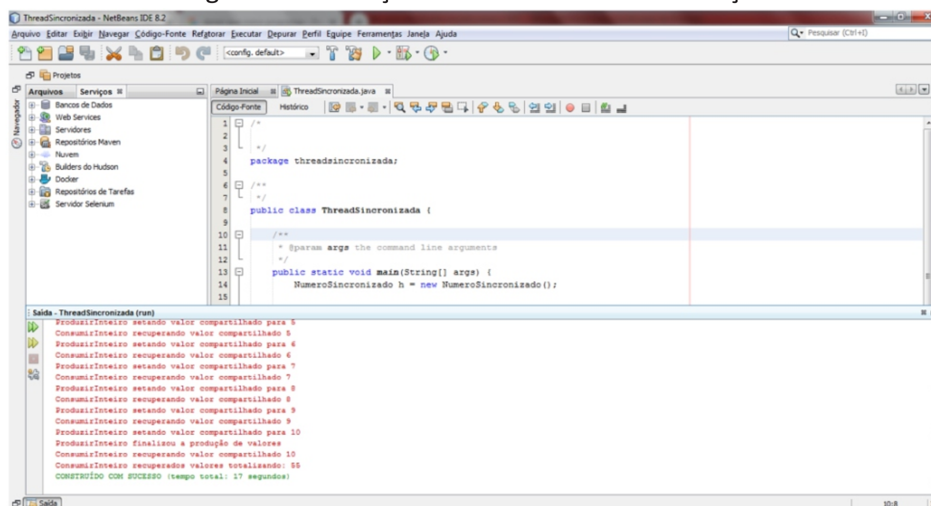
É importante distinguir *threads* de espera que bloquearam porque o monitor estava ocupado versus threads que explicitamente chamaram *wait* dentro do monitor (DEITEL; DEITEL, 2017).

Na conclusão de um método *synchronized*, os *threads* externos que bloquearam porque o monitor estava ocupado podem prosseguir para entrar no objeto. Os *threads* que explicitamente invocaram *wait* somente podem prosseguir quando

notificados via uma chamada feita por outro *thread* (*notify* ou *notifyAll*) (DEITEL; DEITEL, 2017).

A Figura 29 mostra uma das execuções do código-fonte da Figura 28. Nessa versão, utilizando a sincronização, é possível verificar que nenhum valor produzido é perdido pelo consumidor, ou seja, o produtor espera até que o consumidor consuma o valor para produzir um novo.

Figura 29: Execução de *thread* com sincronização



Fonte: Elaborado pelos autores

Exercícios do Capítulo 5

1) Responda V para verdadeiro e F para falso, para cada uma das afirmações abaixo:
   - (   ) O método *run* contém o código que controla a execução de um *thread*
   - (   ) O coletor de lixo da linguagem Java executa como um *thread* de alta prioridade
   - (   ) Um programa dispara a execução de um *thread* chamando o método *run* do *thread*
   - (   ) Um *thread* que acabou de ser criado está no estado executando
   - (   ) O método *getName* define o nome do *thread*
   - (   ) Qualquer objeto Java tem chance de ser um monitor na sincronização de *threads*
   - (   ) Para esperar um número designado de milissegundos e então retornar a execução, um *thread* deve chamar o método *wait*
   - (   ) Um *thread* que executa sobre um método *synchronized* não pode determinar que não pode prosseguir
   - (   ) Enquanto um *thread* dorme ele continua disputando o processador
   - (   ) O método *notify* move um *thread* no estado de espera do objeto para o estado pronto
   - (   ) Um *thread* permanece no estado de nascimento até que seu método start seja chamado
   - (   ) Um *thread* adormecido entra no estado pronto após a expiração do tempo designado de adormecimento
   - (   ) Cada objeto que tem métodos *synchronized* é um monitor
   - (   ) Todas as plataformas Java suportam o conceito de fracionamento de tempo
   - (   ) Um *thread* entra no estado morto somente quando completa sua execução
   - (   ) Um *thread* entra no estado de espera quando chama o método *notify*
   - (   ) Todo *thread* Java tem uma prioridade entre 1 e 100
   - (   ) Quando um *thread* executa um método *synchronized* sobre um objeto ele obtêm o bloqueio deste objeto
   - (   ) O trabalho do *scheduler* do Java é manter o *thread* de prioridade mais alta executando
   - (   ) Um *thread* que chamou *wait* é acordado por um *thread* que chama *notify*
   - (   ) O *multithreading* de Java é dependente de plataforma. Portanto, um aplicativo *multithreaded* pode comportar-se de maneira diferente em diferentes implementações de Java
   - (   ) Depois do método *start* carregar o *thread*, volta para a execução do método chamador

Programação Concorrente – *Threads* em Java

Os exercícios são adaptados dos materiais disponíveis em Rocha (2021) e Tavares & Ferraz (2021):

1) Escreva um programa em Java que realize o cálculo das somas dos valores de uma matriz 4x4 de números inteiros e imprima o resultado destas somas na tela. Faça com que o cálculo do somatório de cada linha seja realizado em paralelo, por *threads;*

2) Crie duas *threads* em Java, uma que fica enviando notícias a cada 5 segundos (textos quaisquer, no total de 10), enquanto a outra *thread* fica enviando a hora, a cada 10 segundos. A hora deverá ser informada no mínimo 5 vezes para que o programa se encerre;

3) Elabore um programa Java que cria 3 *threads*. A primeira escreve na tela a letra "A", a segunda escreve a letra "B" e a terceira a letra "C". Faça com que seja escrito na tela sempre "ABC"

4) Escreva uma classe em Java que permita paralelizar uma pesquisa em um vetor de número inteiros. Isso deve ser feito com o seguinte método: *public buscaParalela(int x, int[] A, int numThreads)*. Este método cria tantas *threads* quanto especificadas em *numThreads*, divide o *array A* em muitas partes e dá a cada *thread* parte do *array* para procurar sequencialmente pelo valor X. Se uma *thread* encontrar o valor X, deve mostrar uma mensagem indicando a posição onde o valor foi encontrado

5) Cinco lebres disputarão uma corrida. Cada lebre pode dar um salto que varia de 1 a 3 metros de distância. A distância percorrida é de 20 metros. Na corrida, cada lebre dará um salto. Informar quantos metros ela pulou a cada salto realizado. Em seguida, a lebre pára para descansar por 2 segundos (*sleep*). Escreva um programa, utilizando *threads* (uma para cada lebre), que informe a lebre vencedora e a colocação de cada uma delas ao final da corrida. Informar, também, quantos pulos cada uma delas deu.

6) Analise o código-fonte abaixo, escrito em Java, e responda às perguntas:

```
1. public class Soma extends Thread {
2. private int mat[][];
3. int i,j, soma=0;
4. public Soma(String name, int matriz[][]) {
5.       this.mat=matriz;
6. }
```

```
7. public void run() {
8. for (i=0;i<=3;i++) {
9.       for (j=0;j<=3;j++) {
10.                 System.out.println(mat[i][j]);
11.                 soma=soma+mat[i][j];
12.      }
13. }
14. System.out.println("Soma:" + soma);
15. }
16. }
```

a) Qual(is) instrução(ões) indicam que esta classe é uma *Thread*?

b) Em que linha inicia o código que é executado quando um objeto baseado na classe Soma é criado?

c) Em que linha inicia o código contendo o que será executado quanto a *Thread* tiver um processador alocado?

d) Qual é o resultado que esta *Thread* calcula (marque a opção correta)?

   a. Soma todos os valores da matriz recebida como parâmetro

   b. Soma os valores da diagonal principal da matriz recebida com parâmetro

   c. Soma os valores da diagonal secundária da matriz recebida como parâmetro

7) Analise o código-fonte abaixo, escrito em Java, e responda às perguntas:

```
1. public class GeraMatriz extends Thread {

2. private int mat[][];
3. int i, j;

4. public GeraMatriz(String name, int mat[][]) {
5.         this.mat=mat;
6. }

7. public void run() {
8. GeraValoresMatriz();
9. }

10. private int[][] GeraValoresMatriz() {
11. for (i=0;i<=3;i++) {
12.       for (j=0;j<=3;j++) {
13.             mat[i][j]=i+j;
14.       }
15. }
16. return mat;
17. }
18. }
```

Responda:

a) A classe *GeraMatriz* é uma *Thread* (Sim ou Não)? Por quê?

b) Por quê o método *Run* chama o método *GeraValoresMatriz* (por quê o código do método *GeraValoresMatriz* não está dentro do método *run*)?

c) Para que serve a instrução *this* (linha 5)?

d) Qual é o resultado calculado retornado pelo método *GeraValoresMatriz*? Marque a resposta certa:

i. Cada posição da matriz recebe o valor da linha atual
ii. Cada posição da matriz recebe o valor da coluna atual
iii. Cada posição da matriz recebe o valor da linha mais coluna atuais

8) Analise o código-fonte abaixo, escrito em Java, e responda às perguntas:

```
1. public class ExemploAula {
2. public static void main(String[] args) {
3.     int matriz[][]= new int[4][4];
4.    GeraMatriz thread1;
5.    Soma thread2;
6.   thread1 = new GeraMatriz("Primeira Thread", matriz);
7.   thread2 = new Soma("Segunda Thread", matriz);
8.   thread2.setPriority(10);
9.   thread1.setPriority(1);
10. thread1.start();
11. thread2.start();
12. }
13. }
```

Responda:

a) Em que linhas do código as *threads* são criadas (nascimento das *threads*)?

b) Em que linhas do código as *threads* ficam prontas para serem executadas?

c) Quantas *threads* são criadas neste código?

d) Quando uma das *threads* inicia a execução, a rotina principal (*main*) executa de forma concorrente com a *thread* (sim ou não)?

e) A linha estabelece a prioridade 10 para a 2ª *thread*. O que isso significa, ou seja, o que está sendo indicado para o escalonador do Java que irá colocar as *threads* em execução (alocar o processador)?

9) Analise o código-fonte abaixo, escrito em Java, e responda às perguntas:

```
1.  public class NumeroSincronizado {
2.  private int sharedInt = -1;
3.  private boolean writeable = true;
4.  public synchronized void setSharedInt(int val) {
5.  while (!writeable) {
6.  try {
7.      wait();
8.  }
9.  catch (InterruptedException e) {
10.         e.printStackTrace();
11. }
12. }
13.
14. System.err.println(Thread.currentThread().getName() + " setando valor
    compartilhado para" + val);
15. sharedInt=val;
16. writeable=false;
17. notify();
18. }
19. public synchronized int getSharedInt() {
20. while (writeable) {
21. try {
22.     wait();
23.
24. }
25. catch (InterruptedException e) {
26.         e.printStackTrace();
27. }
28. }
29. writeable=true;
30. notify();
31. System.err.println(Thread.currentThread().getName() + " recuperando
    valor compartilhado " + sharedInt);
32. return sharedInt;
33. }
34. }
```

Responda:

a) Em que linha(s) do código está a instrução que informa que os *threads* em espera podem ser executados (entrar na fila para execução)?

b) Qual instrução indica que esta classe possui métodos do tipo monitor?

c) Em que linha(s) do código o *thread* é colocado em estado de espera?

d) Qual é a variável de condição do monitor utilizada neste código?

e) Em que linha(s) do código existe o tratamento de exceções?

REFERÊNCIAS BIBLIOGRÁFICAS

BERTAGNOLLI, S. C. Fundamentos de Programação Orientada a Objetos com Java 1.6. Porto Alegre: UniRitter, 2009.

CHARÃO, A. S. Programação Lógica. DELC/CT/UFSM, 2013.

DEITEL, H. M.; DEITEL, P. J. Java: como programar. 10. ed. Pearson, 2017.

GNU.ORG. Kawa. Disponível em: https://www.gnu.org/software/kawa. Acesso em 7 nov. 2019.

LAZARUS-IDE.ORG. Lazarus. Disponível em: https://www.lazarus-ide.org/. Acesso em 7 nov. 2019.

LIMA, E. S. INF 771 Inteligência Artificial. Disponível em: http://edirlei.3dgb.com.br/aulas/ia_2015_1/ListaExercicios03.pdf. Acesso em 7 nov. 2019.

LORENZI, F.; SILVEIRA, S. R. Desenvolvimento de Sistemas de Informação Inteligentes. Porto Alegre: UniRitter, 2011.

PELLEGRINI, J. C. Programação Funcional e Concorrente com Scheme. Santo André, SP: UFABC – Universidade Federal do ABC, 2013.

PEREIRA, S. L. Introdução à Linguagem Prolog. Disponível em: ime.usp.br/~slago/IA-Prolog.pdf. Acesso em 7. Nov. 2019.

ROCHA, E. Threads. Disponível em: https://sites.google.com/site/edemberg/Home/disciplinas/pod/exercicios/threads. Acesso em 21 jan. 2021.

SEBESTA, R. W. Conceitos de Linguagens de Programação. 11. ed. Porto Alegre: Bookman, 2018

SILVEIRA, S. R. Programação com Visual Basic.NET 2005. Porto Alegre: UniRitter, 2006.

SWI-PROLOG.ORG. SWI-PROLOG. Disponível em: https://www.swi-prolog.org/download/stable. Acesso em 7 nov. 2019.

TAVARES, E. et al. Lista de Exercícios de Threads. Disponível em: https://www.cin.ufpe.br/~dfop/Arquivos/Monitoria%20Infra-Software/Lista%20deThreads.pdf. Acesso em 21 jan. 2021.

Tecnologia da Informação

PARADIGMAS DE PROGRAMAÇÃO: UMA INTRODUÇÃO

Autores:
Sidnei Renato Silveira
Antônio Rodrigo Delepiane de Vit
Cristiano Bertolini
Fábio José Parreira
Guilherme Bernardino da Cunha
Nara Martini Bigolin

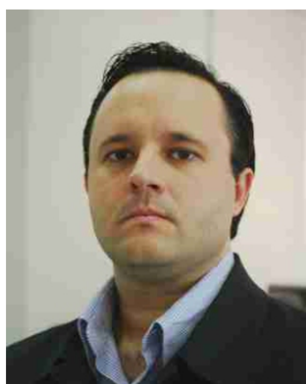
Compartilhando conhecimento

Sobre os Autores

Sidnei Renato Silveira

Professor Adjunto do Departamento de Tecnologia da Informação da UFSM (Universidade Federal de Santa Maria) – Campus Frederico Westphalen/RS. Mestre e Doutor em Ciência da Computação pela UFRGS (Universidade Federal do Rio Grande do Sul). Especialista em Gestão Educacional pelo SENAC. Especialista em Administração e Planejamento para Docentes e Bacharel em Informática pela ULBRA. Técnico em Processamento de Dados pelo Centro Politécnico Cristo Redentor. Coordenador do Curso de Licenciatura em Computação EaD da UAB (Universidade Aberta do Brasil)/UFSM.

Antônio Rodrigo Delepiane de Vit



Professor Adjunto do Departamento de Tecnologia da Informação da UFSM (Universidade Federal de Santa Maria) – Campus Frederico Westphalen/RS. Doutor em Ciência da Computação pela PUC-RS (Pontifícia Universidade Católica do Rio Grande do Sul). Mestre em Ciência da Computação pela UFRGS (Universidade Federal do Rio Grande do Sul) e Bacharel em Informática pela UNIJUÍ.

Cristiano Bertolini

Professor Adjunto do Departamento de Tecnologia da Informação da UFSM (Universidade Federal de Santa Maria) – Campus Frederico Westphalen/RS. Doutor em Ciência da Computação pela UFPE (Universidade Federal de Pernambuco), Mestre em Ciência da Computação pela PUC-RS (Pontifícia Universidade Católica do Rio Grande do Sul). Bacharel em Ciência da Computação pela UPF (Universidade de Passo Fundo).

SOBRE OS AUTORES

Fábio José Parreira

Professor Associado do Departamento de Tecnologia da Informação da UFSM (Universidade Federal de Santa Maria) – Campus Frederico Westphalen/RS. Mestre e Doutor em Engenharia Elétrica pela UFU (Universidade Federal de Uberlândia). Bacharel em Ciência da Computação pelo UNITRI (Centro Universitário do Triângulo).

Guilherme Bernardino da Cunha

Professor Associado do Departamento de Tecnologia da Informação da UFSM (Universidade Federal de Santa Maria) – Campus Frederico Westphalen/RS. Mestre e Doutor em Engenharia Elétrica pela UFU (Universidade Federal de Uberlândia). Bacharel em Ciência da Computação pelo UNITRI (Centro Universitário do Triângulo).

Nara Martini Bigolin

Professora Associada do Departamento de Tecnologia da Informação da UFSM (Universidade Federal de Santa Maria) – Campus Frederico Westphalen/RS. Doutora em Ciência da Computação pela Université de Paris VI Pierre et Marie Curie. Mestre em Ciência da Computação pela UFRGS (Universidade Federal do Rio Grande do Sul). Bacharel em Informática pela PUC-RS (Pontifícia Universidade Católica do Rio Grande do Sul).

Tecnologia da Informação

https://www.facebook.com/Synapse-Editora-111777697257115

https://www.instagram.com/synapseeditora

https://www.linkedin.com/in/synapse-editora-compartilhando-conhecimento/

31 98264-1586

editorasynapse@gmail.com

Compartilhando conhecimento