



# Interativa

## Paradigmas de Linguagens

**Autor:** Prof. Luiz Roberto Forçan

**Colaboradores:** Profa. Vanessa Santos Lessa

Prof. Tiago Guglielmeti Correale

## Professor conteudista: Luiz Roberto Forçan

Profissional atuante no mercado de trabalho de tecnologia da informação como analista de sistemas e desenvolvedor de software desde 1994. É mestrando pela Universidade Paulista (UNIP) em Engenharia de Produção no grupo de lógica paraconsistente, possui especialização em Ensino a Distância pela mesma universidade, bem como especialização em Gestão da Tecnologia da Informação pelo Centro Universitário das Faculdades Metropolitanas Unidas, onde também graduou-se em Administração de Empresas. Exerce a função de professor nos cursos de bacharelado em Ciência da Computação e Sistemas de Informação na UNIP desde 2008. Atualmente, desenvolve soluções nas plataformas de serviços Web, Application Programming Interface (API), Microserviços, Redes, Desktop e Robotic Process Automation (RPA).

### Dados Internacionais de Catalogação na Publicação (CIP)

F697p Forçan, Luiz Roberto.

Paradigmas de Linguagens / Luiz Roberto Forçan. – São Paulo: Editora Sol, 2022.

140 p., il.

Nota: este volume está publicado nos Cadernos de Estudos e Pesquisas da UNIP, Série Didática, ISSN 1517-9230.

1. Tipos. 2. Paradigmas. 3. Programação. I. Título.

CDU 681.3

U514.79 – 22

Prof. Dr. João Carlos Di Genio  
**Reitor**

Profa. Sandra Miessa  
**Reitora em Exercício**

Profa. Dra. Marília Ancona Lopez  
**Vice-Reitora de Graduação**

Profa. Dra. Marina Ancona Lopez Soligo  
**Vice-Reitora de Pós-Graduação e Pesquisa**

Profa. Dra. Claudia Meucci Andreatini  
**Vice-Reitora de Administração**

Prof. Dr. Paschoal Laercio Armonia  
**Vice-Reitor de Extensão**

Prof. Fábio Romeu de Carvalho  
**Vice-Reitor de Planejamento e Finanças**

Profa. Melânia Dalla Torre  
**Vice-Reitora de Unidades do Interior**

### **Unip Interativa**

Profa. Elisabete Brihy  
Prof. Marcelo Vannini  
Prof. Dr. Luiz Felipe Scabar  
Prof. Ivan Daliberto Frugoli

### **Material Didático**

Comissão editorial:

Profa. Dra. Christiane Mazur Doi  
Profa. Dra. Angélica L. Carlini  
Profa. Dra. Ronilda Ribeiro

Apoio:

Profa. Cláudia Regina Baptista  
Profa. Deise Alcantara Carreiro

Projeto gráfico:

Prof. Alexandre Ponzetto

Revisão:

Kleber Souza  
Auriana Malaquias



# Sumário

## Paradigmas de Linguagens

APRESENTAÇÃO .....	7
INTRODUÇÃO .....	8

### Unidade I

1 CONCEITOS BÁSICOS DAS LPS .....	14
1.1 Conceitos básicos .....	15
1.1.1 Algoritmos e programas .....	15
1.1.2 Linguagens de programação (LPs) .....	18
1.1.3 Processo de desenvolvimento de programas .....	35
1.1.4 Estilo e qualidade de programas .....	40
1.2 Tipos de dados .....	43
1.2.1 Tipos de dados primitivos .....	44
1.2.2 Tipo string de caracteres .....	47
1.2.3 Tipos ordinais definidos pelo usuário .....	47
1.2.4 Tipos array .....	48
1.2.5 Tipos registro .....	51
1.2.6 Tipos ponteiro .....	52
1.3 Expressões e instruções de atribuição .....	54
1.3.1 Expressões aritméticas .....	54
1.3.2 Conversões de tipo .....	59
1.3.3 Expressões relacionais e booleanas .....	60
1.3.4 Instruções de atribuição .....	61
2 ESTRUTURAS DE CONTROLE .....	63
2.1 Estruturas condicionais .....	64
2.2 Estruturas de repetição .....	73
3 SUBPROGRAMAS .....	78
3.1 Fundamentos .....	78
3.2 Métodos de passagem de parâmetros .....	83
3.3 Sobrecarga de subprogramas .....	85
4 TIPOS ABSTRATOS DE DADOS .....	86
4.1 Fundamentos de abstração .....	87
4.2 Encapsulamento .....	88
4.3 Métodos de acesso a dados (public, private e protected) .....	88
4.4 Exemplos de abstração de dados em Java .....	88

## Unidade II

5 PARADIGMAS DAS LPS .....	96
5.1 Programação estruturada .....	96
5.1.1 Caracterização.....	96
5.2 Linguagens imperativas (Basic, Pascal, C) .....	99
6 PRÁTICAS DE PROGRAMAÇÃO UTILIZANDO PROGRAMAÇÃO ESTRUTURADA: EXEMPLOS DE PROGRAMAS E IMPLEMENTAÇÕES PRÁTICAS (BASIC, PASCAL, C).....	100
7 PROGRAMAÇÃO ORIENTADA A EVENTOS .....	103
7.1 Conceitos fundamentais .....	103
7.2 Caracterização .....	108
7.3 Linguagens orientadas a eventos (Delphi, Visual Basic).....	110
7.4 Práticas de programação utilizando programação orientada a eventos (Delphi, Visual Basic).....	111
8 PROGRAMAÇÃO ORIENTADA A OBJETOS.....	115
8.1 Conceitos fundamentais .....	115
8.2 Caracterização e comparação .....	129
8.3 Linguagens orientadas a objetos (Smalltalk, C++, Java, C#) .....	130

## APRESENTAÇÃO

Prezado aluno, este livro-texto apresenta uma introdução aos principais paradigmas de programação e sua importância na solução de problemas. Realiza-se um estudo comparativo das várias linguagens de programação (LPs) atuais e as razões para aprender e ampliar os conhecimentos sobre elas no desenvolvimento de software e programas. Pretende-se demonstrar as propriedades desejáveis e os fatores que influenciam o projeto de uma LP, a arquitetura do computador e as técnicas básicas de programação.

Para adquirir conhecimento dos paradigmas de LPs, demanda-se muito estudo e dedicação. As LPs oferecem mecanismos que, quando utilizados corretamente, trazem muitos benefícios aos programadores, evitando perda de tempo de processamento, fato este capaz de comprometer a produtividade do profissional.

O entendimento dos diferentes paradigmas é adquirido através do estudo dos conceitos e princípios que orientam a construção das LPs, isso ajuda o aluno a definir qual dos paradigmas utilizar para determinada solução de um problema particular da área computacional.

A disciplina *Paradigmas de Linguagens* pretende desenvolver o saber sobre os vários paradigmas de programação existentes, assim como as diversas linguagens de computador onde eles se aplicam. Algumas das questões mais comuns a emergir são: quais são os principais paradigmas? qual é a intenção e ideologia de cada um deles? qual a origem de cada paradigma? quais são as LPs dominantes? qual a finalidade de cada LP? qual o princípio de cada LP? qual a relevância de tudo isso no mercado de trabalho? Aqui, focaremos nos paradigmas e LP mais utilizados pelo mercado de trabalho.

Esta disciplina tem o propósito de incentivar o aluno a estudar as regras sintáticas e semânticas, os componentes das linguagens, as suas expressões, as atribuições, os tipos de dados, a utilização de subprogramas e estruturas de controle que contribuem com o desenvolvimento de software.

Consequentemente, apresenta-se a evolução das LPs, que vem progredindo constantemente para se adaptar às novas demandas de mercado, originando-se assim vários paradigmas de programação e linguagens de computadores.

Bons estudos!

## INTRODUÇÃO

Este livro-texto tem como propósito transmitir conhecimentos sobre os principais paradigmas de programação aplicados no desenvolvimento de software. Estudaremos a evolução das LPs e a aplicação dos algoritmos utilizando a estrutura dessas linguagens.

Na unidade I, demonstraremos uma discussão sobre a importância do estudo dos diversos paradigmas de linguagem e dos motivos para aprendermos os conceitos básicos das LPs.

Descreveremos os conceitos básicos das LPs como algoritmos e programas e os tipos de dados utilizados em um programa, como aqueles primitivos, strings de caracteres, ordinais definidos pelo usuário, array, registro e ponteiro.

Ainda, explicaremos a utilização de expressões e instruções ou sentenças de atribuição em LPs e sua combinação de expressões aritméticas, relacionais e booleanas, além de apresentar as conversões entre tipos e os operadores de atribuição.

Posteriormente, descreveremos as estruturas de controle, nas quais serão tratadas as diversas estruturas condicionais e as estruturas de repetição muito utilizadas para implementar as condições lógicas dos algoritmos.

Discutiremos acerca de subprogramas e suas implementações no desenvolvimento de software, apresentando tópicos como métodos de passagem de parâmetros por valor, por resultado e por referência.

Por fim, demonstraremos os fundamentos dos tipos abstratos de dados e métodos de acesso a dados com exemplos de implementação na LP Java.

Já na unidade II, estudaremos alguns dos principais paradigmas de linguagens existentes:

- Programação estruturada.
- Programação orientada a eventos.
- Programação orientada a objetos.

Essas seções sobre os paradigmas serão independentes entre si, podendo os alunos estudá-las pela ordem que considerarem mais interessante. Em cada uma delas, serão apresentadas as particularidades fundamentais de cada paradigma e as LPs que fornecem suporte para este paradigma.

Exibiremos as particularidades do paradigma de linguagem estruturado e exemplos de aplicação nas LPs C, Basic e Pascal.

Na sequência, exporemos os conceitos fundamentais do paradigma de programação orientada a eventos como exemplo de aplicação nas linguagens Visual Basic e Delphi. Esse paradigma encontra-se



em evidência atualmente nas mais variadas aplicações de programação, por exemplo, em mobile e aplicativos de interface gráfica com usuários (GUI) em que os alunos precisam estudar tópicos como troca de mensagens e manipulação de eventos.

Por último, trataremos do paradigma de programação orientada a objetos (POO). Serão abordados os conceitos da OO através das LPs Java, Visual C# e C++, devido a sua dominância no mercado de desenvolvimento de softwares comerciais e embarcados. O estudo desse paradigma é muito importante, pois, atualmente, é muito exigido o seu conhecimento pela área de TI. Falaremos sobre os seus principais conceitos como herança, polimorfismo, encapsulamento, abstração, entre outros.



# Unidade I

Iniciaremos os estudos abordando os conceitos e componentes fundamentais que fazem parte de uma LP. Esse conhecimento é fundamental para reflexão acerca do que são e quais LPs influenciam diretamente no desenvolvimento de software na atualidade e que detêm relevância, diante do mercado, na criação de soluções modernas e práticas.

O entendimento das LPs exhibe perspectivas de realizações nos mais diversos segmentos do mercado. Traremos as informações que são importantes por fazerem parte dos fundamentos da área de tecnologia da informação e desenvolvimento de software.

É importante estudarmos a evolução das LPs por meio da relação entre estas e os algoritmos. Apresentaremos os principais elementos das LPs, por exemplo:

- Conhecer os conceitos básicos de uma LP e sua importância no desenvolvimento de software.
- Aprender os conceitos sobre algoritmos e programas.
- Entender os conceitos sobre as estruturas computacionais como tipos de dados, expressões aritméticas, relacionais, lógicas e literais.
- Compreender as estruturas de controle, condicionais e de repetição, bem como as instruções de atribuições e o conceito de subprogramas.

**Quais as razões para estudar os conceitos de paradigmas de linguagens ou paradigmas de programação?**

O papel do desenvolvedor de software ou programador tornou-se essencial na sociedade, que está ficando cada vez mais digital. Então, por que o ensino sobre Ciência da Computação e Sistemas de Informação ou Tecnologia da Informação (TI) está se intensificando agora? O que é diferente do passado?

A diferença é que o avanço tecnológico tem sido de tal ordem que requer um número muito maior de desenvolvedores de software altamente treinados e capacitados. A economia, os serviços e métodos de produção também se aprimoraram, de modo que atualmente existe a necessidade de digitalização dos processos e informação. Sendo assim, a ênfase em TI surge muito mais como consequência da rapidez com que acontecem as mudanças tecnológicas e novas demandas no mercado de trabalho, não apenas como um modismo. A competição acirrada entre empresas concorrentes e por ganhos na economia também força o mercado a adotar tecnologias diferentes para atender às necessidades dos empreendedores.

O objetivo principal de um paradigma de linguagem é determinar o entendimento ou a maneira como o desenvolvedor de software ou programador deve conduzir-se para estruturar e executar um programa de computador, criado em determinada LP (SEBESTA, 2011).

A palavra paradigma quer dizer modelo, ou seja, adota-se um padrão a ser seguido. Então, o paradigma de linguagem para um desenvolvedor de software é o modo como ele, de maneira organizada, estrutura os comandos de determinada linguagem de computador, com o objetivo de gerar um software capaz de executar uma tarefa ou atividade para resolver o problema de alguém.



### Lembrete

Paradigma é um substantivo masculino, trata-se de um exemplo ou padrão a ser seguido. Deriva do grego "parádeigma, atos", sendo uma palavra utilizada para representar uma maneira de pensar ou agir em determinada época ou em um contexto específico. Ou seja, é um modelo ou padrão a ser utilizado pela maioria das pessoas. O termo também pode fazer referência a uma abordagem de pensamento filosófico ou científico.

O desenvolvedor ou programador pode adotar um paradigma de linguagem como uma metodologia para desenvolver um programa de computador. Portanto, a decisão sobre qual paradigma deve ser utilizado é feita do ponto de vista da realidade ou da maneira como o profissional atua sobre ela, estabelecendo uma maneira particular de tratar os problemas e criar as respectivas soluções.

Para potencializar as análises e soluções na resolução de um problema, o desenvolvedor pode combinar em uma LP vários paradigmas de linguagem, encontrando o mais adequado para aquele inconveniente computacional. Por exemplo: qual é o paradigma ou LP mais apropriado para desenvolver um sistema de comércio eletrônico? um sistema de vendas? um sistema qualquer em mobile?



### Observação

O paradigma de linguagem determina a forma ou visão com que o desenvolvedor de software deve seguir para estruturar e executar um programa de computador.

Uma LP pode dar suporte a regras ou estruturas de um ou mais paradigmas. Temos como exemplos as linguagens Python e C++ que oferecem suporte ao paradigma estruturado e orientado a objetos, enquanto a linguagem experimental Leda é projetada para fazê-lo aos paradigmas de linguagem orientados a objetos, funcional, imperativa e lógica (TUCKER; NOONAN, 2009). Estudaremos com mais detalhes esses paradigmas posteriormente.

Nas aulas iniciais de programação de computadores, em muitos cursos de computação e nos cursos de graduação tradicionais, ainda predominam as LPs que dão suporte ao paradigma estruturado, pois é mais fácil de entender e apresenta uma curva de aprendizado menor. Entretanto, não existe uma padronização nas linguagens de programação utilizadas nos cursos e podem surgir casos em que o aluno tenha que aprender a programar em um ou mais paradigmas.

Um paradigma de programação é um padrão de solução de problemas relacionados a determinada categoria de programas e LPs (TUCKER; NOONAN, 2009). Quatro paradigmas de programação diferentes e essenciais evoluíram nas últimas três décadas. São eles:

- Programação funcional.
- Programação lógica.
- Programação imperativa.
- Programação orientada a objetos.

### Exemplo de aplicação

---

Por qual paradigma de programação você começaria a estudar? Faça uma pesquisa na internet sobre os cinco paradigmas de linguagens mais utilizados hoje em dia e reflita a respeito.

---

### Motivos para aprender os conceitos de LPs

Por que os desenvolvedores de software profissionais e estudantes de Ciência da Computação e Sistemas de Informação devem estudar conceitos gerais sobre as LPs?

Para responder a essa pergunta, discutiremos rapidamente os principais ramos da programação, os recursos da linguagem, a sua estrutura e os critérios que podem servir de motivos para nortear os estudantes sobre os benefícios de investir seu tempo e recursos aprendendo sobre conceitos de LPs. Podemos listar os seguintes itens como vantagens:

- Aperfeiçoar a capacidade para expressar ideias e transformar soluções computacionais em programas de computador. O maior entendimento na aplicação dos conceitos de uma LP pode capacitar o desenvolvedor a pensar e estruturar a solução para um problema de negócio ou computacional.
- Melhorar a habilidade e o potencial de um desenvolvedor para utilizar uma linguagem já conhecida. Não é comum um desenvolvedor utilizar e conhecer todos os recursos da LP que ele utiliza, pois as LPs contemporâneas são extensas e complexas. Quanto maior o conhecimento a respeito das funcionalidades de partes antes desconhecidas e não utilizadas das linguagens e implementação de uma LP, maior a possibilidade de o desenvolvedor construir softwares mais eficientes e melhores.

- Aprimorar o embasamento do desenvolvedor com o objetivo de selecionar uma LP mais apropriada para determinado projeto de software por estar familiarizado com os recursos e opções fornecidos por essa linguagem e implementar tais recursos em ambiente de produção. Por exemplo, saber que a linguagem C não realiza checagem dinâmica dos índices de acessos a posições de vetores pode ser determinante para escolhê-la em uma aplicação de tempo real que faz uso frequente de acessos vetoriais (VAREJÃO, 2004). Na prática, o desenvolvedor no momento de escolher a LP para um projeto novo acaba utilizando aquela com a qual está mais familiarizado. Quanto mais LPs o desenvolvedor conhecer, mais opções ele terá para definir a melhor arquitetura de software capaz de resolver os problemas de negócio em projetos de desenvolvimento de software (SEBESTA, 2011).
- Aumentar a aptidão e facilidade de aprendizado das novas. O desenvolvedor que domina os conceitos de determinado paradigma tem mais habilidade para aprender linguagens que dão suporte àquele paradigma. Por exemplo, quem tem domínio sobre o paradigma orientado a objetos, possui mais facilidade em aprender linguagens como Java, C# e C++. O desenvolvimento de software ainda é uma área de conhecimento relativamente nova. As metodologias de gerenciamento de projetos, modelagem de software e programação estão em constante evolução. Isso torna a profissão de desenvolvedor desafiante, pois exige aprendizado contínuo e muita dedicação.
- Conhecer previamente o paradigma de uma LP pode diminuir a sua curva de aprendizado. Por isso é importante aprender os conceitos fundamentais de uma LP. O mesmo acontece com as linguagens naturais. Quanto mais conhecimento se tem sobre a gramática de seu idioma nativo, mais facilidade terá para aprender uma segunda língua (SEBESTA, 2011).
- Possuir um aprendizado mais amplo dos recursos da LP torna o desenvolvedor menos limitado na programação de computadores. Quanto maior a habilidade e o entendimento na implementação e nas funcionalidades de uma LP, mais possibilidades o profissional tem para desenvolver softwares mais eficientes e melhores. Devemos, por exemplo, entender o motivo de durante a implementação de uma LP os algoritmos iterativos serem mais eficientes que os recursivos (VAREJÃO, 2004).
- Dar mais possibilidades e conhecimento aos desenvolvedores para criação das LPs Embora projetar uma nova LP seja uma oportunidade rara na carreira de um profissional de tecnologia da informação, é comum surgirem situações em que ele necessite fazê-lo. Por exemplo, em uma linguagem de comunicação entre os usuários de sistemas e o programa de computador.

### 1 CONCEITOS BÁSICOS DAS LPs

Neste capítulo, apresentaremos os conceitos básicos para aprendizagem e ampliação dos conhecimentos sobre LPs. Mostrando inclusive a importância de entender o papel de cada LP no processo de desenvolvimento de software.

## 1.1 Conceitos básicos

Com objetivo de obter produtividade e consequentemente ganho de tempo, é importante conhecer as propriedades de uma LP, os detalhes que influenciam o projeto de uma LP, a arquitetura do computador e as técnicas de programação.

### 1.1.1 Algoritmos e programas

A vida de um desenvolvedor de software é solucionar, por meio de um programa de computador ou através da utilização de códigos, problemas computacionais do dia a dia das pessoas ou organizações. Assim sendo, devemos inicialmente estudar como pensa um programador, e disciplinar sua mente para transformar sua ideia de solução em códigos de computador. O primeiro passo para essa transformação é entender o que são algoritmos. Eles podem ser definidos como um conjunto de procedimentos, tarefas ou ações, que utilizam uma sequência lógica, para realizar determinada atividade (DAURICIO, 2015).

Um exemplo de algoritmo pode ser a sequência de passos para a instalação de um programa de computador. Essa atividade requer que executemos determinada sequência de ações a fim de instalarmos o programa e fazermos sua configuração, para depois podermos utilizá-lo.



#### **Lembrete**

Algoritmo é uma sequência de procedimentos ordenados de maneira lógica para a resolução de um problema definido ou tarefa (DAURICIO, 2015).

A primeira etapa para aprender programação de software não compreende somente o uso do computador, mas disciplinar a mente a encontrar o melhor caminho a ser percorrido para executar uma tarefa com eficiência e eficácia.



#### **Observação**

O conceito de algoritmo é a essência da Ciência da Computação e Programação, então desenvolver softwares é basicamente construir algoritmos.

No nosso dia a dia, lidamos com várias situações para resolver problemas ou chegar a um objetivo específico. Para tanto, utilizamos uma sequência determinada de passos ou procedimentos. Através do raciocínio sobre qual seria a melhor sequência de etapas, reproduzimos, mesmo que seja através de uma linguagem natural, os passos para solucionar algo, buscando antecipar o entendimento de como realizar tal elucidação computacionalmente a partir daquela solução encontrada.

Não existe um padrão para a solução. Por exemplo, considere um programa cujo objetivo seja fazer a soma de 2 números e exibir o resultado do cálculo. Vamos implementar o exemplo desse algoritmo em linguagem natural:

- Informar o primeiro número.
- Informar o segundo número.
- Realizar a soma dos dois números informados.
- Exibir o resultado da soma.

Repare que, ao ler a descrição narrativa anterior, qualquer pessoa pode entender qual é o problema a ser resolvido. Então, uma vez definido que esses passos são os mais lógicos para a resolução, a seguir, devemos convertê-los em comandos de programação.

Posteriormente à definição dos passos, podemos representar o algoritmo por meio de fluxograma, por exemplo, conforme a figura a seguir.

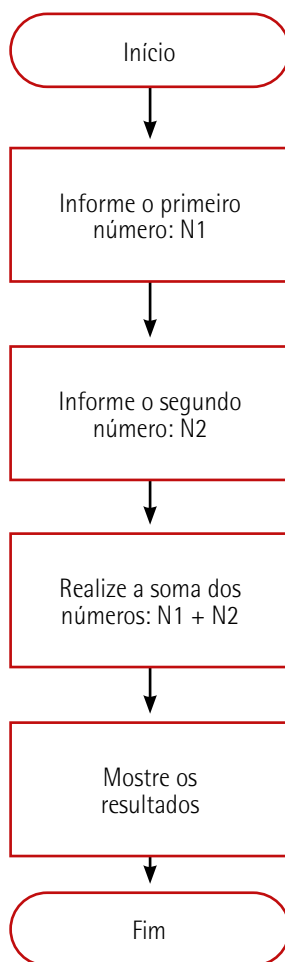


Figura 1 – Algoritmo em fluxograma



Um modo de representação de algoritmos é chamado de Portugol ou pseudocódigo. Nele são encontradas formas de representar algoritmos, com instruções, seguindo determinado padrão ou estrutura muito similar ao utilizado em uma LP. Posteriormente facilitando a transferência do código para qualquer LP.

Consta na sequência um exemplo do algoritmo em Portugol:

Algoritmo soma de números positivos

Declaração de variáveis

numero1, numero2, soma: Inteiro

Início

leia(numero1)

leia(numero2)

soma <- numero1+ numero2

escreva(soma)

Fim



### Observação

Antes de começarmos a fazer qualquer programa de computador, devemos analisar o problema e elaborar a solução na forma de algoritmo. Depois disso, podemos executar o algoritmo em qualquer LP.

Um programa também é um software, ou seja, um programa de computador que possibilita que novos deles sejam escritos, facilitando para que o desenvolvedor de software elabore os seus próprios programas. Um ambiente de desenvolvimento integrado (IDE) reúne ferramentas essenciais para desenvolvimento e teste de software em um único ambiente gráfico.

O IDE utiliza um compilador que também é um programa que converte o código-fonte gerado em uma LP de alto nível para um código de máquina de baixo nível. Ele vem evoluindo muito com o passar dos anos, tornando-se mais que um simples editor de código-fonte, com boa usabilidade, pois auxilia na criação de código de software por meio de diversas facilidades, por exemplo, correção da sintaxe com indicações visuais, complemento automático de código e verificação automática de erros. O IDE permite diferentes níveis de acesso aos hardwares disponíveis e diversos sistemas operacionais (multiplataforma).

Esses ambientes facilitam a vida do desenvolvedor e aumentam consideravelmente a sua produtividade. Entretanto, apesar das facilidades oferecidas por tais plataformas, o desenvolvedor precisa conhecer a fundo os conceitos da linguagem de computador, como suas instruções, propriedades, além de respeitar a sintaxe da linguagem, tipos de dados e regras para a construção dos comandos da linguagem. Também é preciso que ele saiba implementar um algoritmo como uma sequência lógica determinada na construção do programa de computador.



### Observação

Para desenvolver um software, é necessário dominar os comandos, a sintaxe e a semântica da LP e estudar seus tipos de dados, que variam de uma linguagem para outra.

### 1.1.2 Linguagens de programação (LPs)

As LPs são utilizadas pelo desenvolvedor para descrever os comandos ou instruções provenientes dos algoritmos para um computador. Por essência, também são um programa que permite aos programadores escreverem seus próprios programas.

Com o objetivo de aumentar sua produtividade, é importante que o desenvolvedor amplie seus conhecimentos na LP que for utilizar em relação à sua sintaxe, semântica, instruções, propriedades, tipos de dados, símbolos e regras para a formação de comandos ou instruções da linguagem (SEBESTA, 2011).

Cada LP possui certos tipos de dados que oferecem mais recursos que outras para determinada situação. Portanto, a fim de dominar a escrita de um programa de computador é importante que o profissional conheça a sintaxe e a semântica das LPs para ter uma visão completa de suas vantagens e limitações.



### Lembrete

Como cada instrução de uma LP termina por estabelecer uma ação que o hardware deve executar, é importante conhecer o sistema operacional mais apropriado para ela e em quais hardwares a sua execução será mais adequada.

É normal muitos desenvolvedores iniciantes se preocuparem apenas com a LP, porém, ao trabalharmos com desenvolvimento de software, precisamos entender que, além da LP, temos de nos atentar tanto ao sistema operacional (SO), quanto ao hardware de execução.

O ambiente de desenvolvimento de uma LP precisa possibilitar acesso a rotinas específicas do SO e do hardware. Uma LP pode rodar em apenas um SO, por exemplo, Windows, ou em Windows e Linux de maneiras diferentes. Também existem algumas linguagens interpretadas como Java e Python, que são mais independentes do SO.



### Observação

Antes de implementar uma LP, devemos pesquisar em qual ambiente operacional (Windows, Linux, Android, Mac ou Apple IOS) ela é mais eficiente com relação a utilização de memória, tempo de processamento, tamanho do código-fonte etc.

É muito comum um usuário queixar-se para o desenvolvedor que o sistema está lento, porém um sistema engloba mais que apenas um LP. Ele envolve aspectos de hardware, por exemplo, discos, memória, SO e processador. Muitas vezes, a lentidão pode ser causada também por hardware e não por um algoritmo mal elaborado ou um programa malfeito.



### Lembrete

Quando pensamos em sistema de software devemos estabelecer a correta relação entre a LP, a arquitetura do computador, o SO e as instruções de hardware.

Cada LP pode dar suporte a um ou mais paradigmas de programação. Por exemplo, a linguagem Python suporta tanto paradigma estruturado como paradigma orientado a objetos. A LP segue padrões de codificação binária, com semânticas e sintaxe próprias. O desenvolvedor de software se utiliza dessas bibliotecas de conjuntos de funções, códigos e recursos padrões e nativos para criar sistemas que resolvem problemas das organizações e pessoas.



### Observação

O Eniac (Electrical Numerical Integrator and Computer) é considerado o primeiro computador do mundo, tendo sido desenvolvido na época da Segunda Guerra Mundial. Ele era enorme, ocupava um espaço de 270 m<sup>2</sup> e pesava 30 t.

A LP é definida por suas regras sintáticas e semânticas, que criam os programas compostos de instruções enviadas para a CPU. Após concluído o código-fonte e serem as regras verificadas, o compilador traduz o programa que foi escrito para o computador em binário, para que seja executado.



### Saiba mais

Para converter textos ou números em binário, da mesma maneira que o compilador, utilize a ferramenta disponibilizada a seguir e amplie seus conhecimentos:

UNIT-CONVERSION.INFO. *Convert text to binary*. [s.d.]. Disponível em: <https://bit.ly/3FjeaUq>. Acesso em: 11 jan. 2022.

Em relação à forma de tradução do código-fonte para o código de máquina ou binário, as LPs podem ser classificadas em interpretadas ou compiladas. Também existem abordagens híbridas, como Java, que é compilada, mas requer uma máquina virtual para a execução do bytecode (SEBESTA, 2011).



### Lembrete

Uma LP atua como uma camada intermediária, que estabelece a comunicação entre o ser humano, que se expressa através de um idioma, e um computador, que entende linguagem de máquina.

No início, a maioria dos programas comerciais das grandes corporações era desenvolvida para os computadores de grande porte, conhecidos como mainframes. A LP Cobol reinava na época e é utilizada até hoje, inclusive com a denominação "Cobol for Windows", já com um visual mais bonito e técnicas de arrastar e soltar componentes e interfaces.

Outras linguagens também tiveram sua época de domínio. Na década de 1950, havia a predominância da linguagem Assembly, porém surgiu o Fortran, uma linguagem imperativa que foi muito utilizada pelos cientistas para realizar cálculos, além do Cobol e Lisp, que foram consideradas pioneiras das LPs atuais. Elas facilitavam o trabalho dos programadores e buscavam a eficiência computacional por meio do uso racional do processador e da memória.

Nos anos de 1960, surgiram Pascal e Basic, voltadas para o paradigma estruturado. Entre os anos de 1970 e 1980, foi criada a linguagem Ada, já voltada para computadores de pequeno porte (microcomputação). Nessa época, começou a programação em larga escala para atender à crescente demanda de mercado. Ainda, na mesma época, apareceu o paradigma orientado a objetos com a Smalltalk, bem como as LPs Eiffel, Perl e C++.

A partir de 1990, com a expansão da internet, surgiram linguagens para trabalhar nesse ambiente, como o Java, Python, PHP, Ruby, C# e Haskell.

Entre os anos de 1960 e 1970, os computadores eram mainframes, ou seja, computadores de grande porte, com o processamento centralizado que executava um grande volume de informações. As LPs

que davam suporte a esse tipo de processamento eram imperativas e estruturadas. A arquitetura dos projetos era do tipo top-down e assim começaram a ser construídos grandes sistemas, por exemplo, sistemas de vendas para produtos comerciais e industriais.

Nos anos de 1980, surgiram os computadores interligados por grandes redes de telecomunicação, e as LPs rodavam no servidor, que era uma máquina centralizada, semelhante ao mainframe. As metodologias de desenvolvimento começaram a ser orientadas a dados e não mais a processos. Também começaram a ser desenvolvidos projetos orientados a objetos utilizando os conceitos de abstração de dados e encapsulamento.







Durante a década de 1990, a internet passou a crescer e se popularizar, marcando o início de uma era de grande evolução tecnológica na qual foi muito utilizada a arquitetura cliente-servidor, que deu início às LPs que rodavam do lado do servidor e as LPs que rodavam do lado do cliente, geralmente nos browsers como Netscape e Internet Explorer, entre outros.





A partir dos anos 2000, começaram a aparecer os grandes datacenters com processamento em nuvens. As empresas passaram a ter a opção de descentralizar seus datacenters internos, optando por obter uma infraestrutura em forma de serviços em nuvens, e, até hoje, as LPs se adaptam para dar suporte a este ambiente de desenvolvimento.

Atualmente, as LPs não são mais tão limitadas, uma vez que dão suporte a vários paradigmas e possuem muitos recursos disponíveis. Por exemplo, para desenvolver um programa com o POO, dependendo da aplicação e do contexto em que será desenvolvido, o programador tem um leque de linguagens disponíveis a ser empregado, por exemplo, Python, Java, C# e C++.

Quais são as LPs mais utilizadas pelo mercado na atualidade?

**Tabela 1 – Ranking das 10 LPs atuais**

Jan 2022	Jan 2021	Change	Programming Language		Ratings	Change
1	3	▲		Python	13.58%	+1.86%
2	1	▼		C	12.44%	-4.94%
3	2	▼		Java	10.66%	-1.30%
4	4			C++	8.29%	+0.73%
5	5			C#	5.68%	+1.73%
6	6			Visual Basic	4.74%	+0.90%

Jan 2022	Jan 2021	Change	Programming Language	Ratings	Change
7	7		 JavaScript	2.09%	-0.11%
8	11	▲	 Assembly language	1.85%	+0.21%
9	12	▲	 SQL	1.80%	+0.19%
10	13	▲	 Swift	1.41%	-0.02%

Fonte: Tiobe (2022).

A figura a seguir mostra a evolução das LPs.

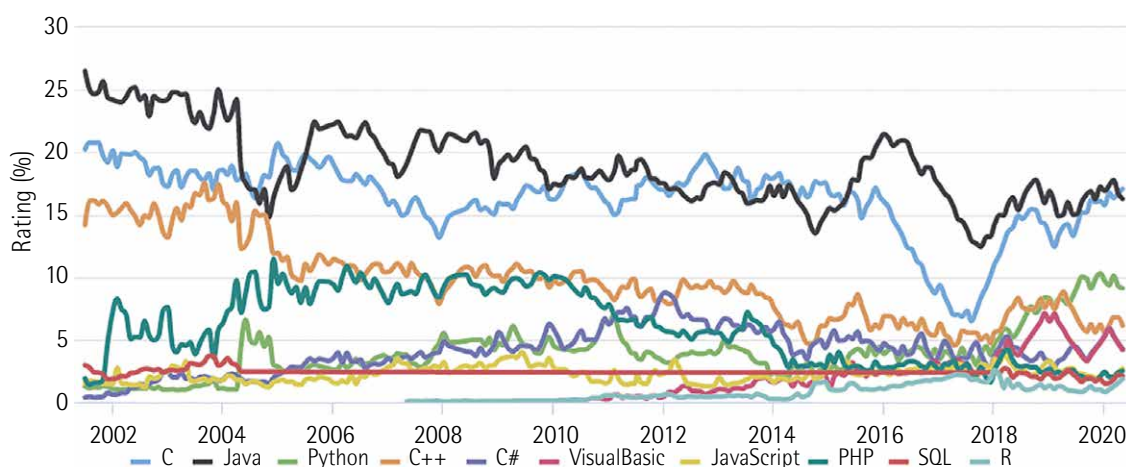


Figura 2 – Evolução das LPs através dos anos

Fonte: <https://bit.ly/346Rxpl>. Disponível em: 21 jan. 2022.

Quanto mais profundo for o conhecimento de um paradigma pelo desenvolvedor, mais fácil se tornará para ele o aprendizado de novos paradigmas existentes com o objetivo de aplicação no desenvolvimento de software. Na prática, um paradigma é um modelo a ser seguido.



## Saiba mais

Com o objetivo de conhecer quais são as 20 LPs mais utilizadas atualmente, acesse o site a seguir.

TIOBE. *Tiobe index for January 2022*. Tiobe, 2022. Disponível em: <https://bit.ly/3KQmxLq>. Acesso em: 21 jan. 2022.

## O paradigma imperativo

Trata-se do paradigma mais antigo, baseado no modelo computacional de John von Neumann, conforme a figura 3. Nessa arquitetura, tanto os dados quanto os programas são armazenados na mesma memória. A unidade central de processamento (CPU), que executa instruções, é separada da memória. Então, as instruções e dados devem ser transmitidos da memória para a CPU e os resultados das operações na CPU devem ser retornados para a memória.

O programa contém uma série de comandos para obter entradas, produzir saídas, executar cálculos ou atribuir valores a variáveis que correspondem à abstração do endereçamento das posições de memória. Esse paradigma é a base para outros com as mesmas propriedades: estruturado, concorrente e orientado a objetos. Linguagens: Fortran, Cobol, C, Ada, Perl (SEBESTA, 2011).

No seguinte exemplo de trecho de código do paradigma imperativo, se o valor da variável `tipoDocumento` for igual a "CPF", o programa realizará a impressão:

```
String tipoDocumento = "CPF";  
if(tipoDocumento == 'CPF') {  
    print("O documento selecionado foi o CPF.");  
}
```

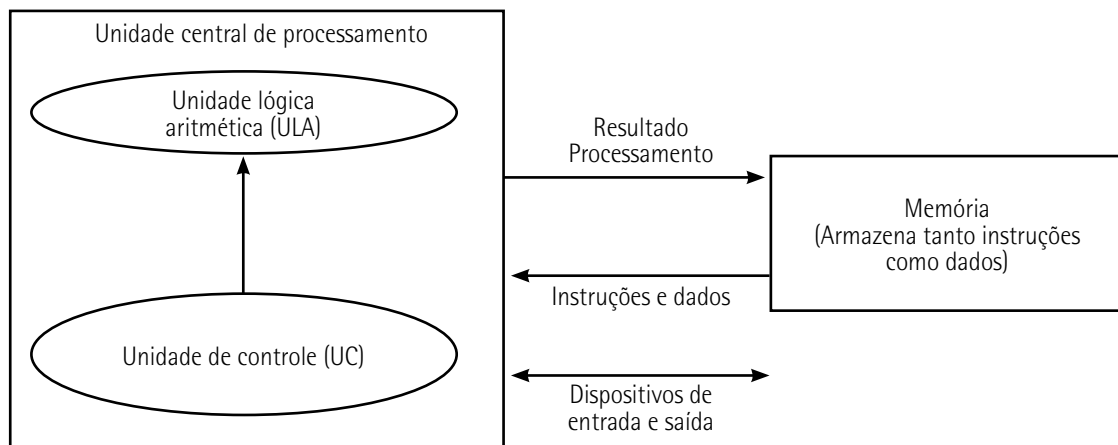


Figura 3 – A arquitetura de Von Neumann

Fonte: Sebesta (2011, p. 39).

## Paradigma orientado a objetos

Paradigma que fornece um modelo no qual um programa é uma coleção de objetos que trocam mensagens entre si, transformando o seu estado. Temos como exemplos: Smalltalk, Java, Python, C++, e C#.

### Paradigma estruturado

Paradigma que dá suporte às principais LPs das décadas de 1970 e 1980, como C e Pascal. Ele tem como base a metodologia top-down, em que o desenvolvimento era realizado do módulo mais geral para os mais especializados, conforme passava-se a conhecer melhor os problemas computacionais.

### Paradigma funcional

Esse paradigma modela o problema computacional como uma coleção de funções matemáticas, que recebem um conjunto de valores e retornam um valor. O cerne aqui está na avaliação e manipulação transparente de funções. Por exemplo, a função matemática  $f(y) = y + 1$ . Se o valor de entrada  $y$  for 1, o resultado será 2.

Uma função pode fazer chamada a outras, uma vez que utiliza o resultado de uma função como argumentos de entrada para outra função (conceito chamado de recursão). Essas LPs dão suporte a sistemas especialistas que utilizam o conceito de inteligência artificial (IA) por meio das seguintes linguagens: F#, Haskell, Lisp e ML.

Utilizaremos como exemplo o código em Lisp, que calcula recursivamente o fatorial do número positivo. O número informado será 3; para chamar a função, utiliza-se (fatorial 3). Como o cálculo do fatorial é realizado pela multiplicação dos seus antecessores, o programa calculará  $3 \times 2 \times 1$ . Quando o resultado do cálculo do fatorial for zero, o programa retornará 1. Observe que existe uma chamada da própria função em seu corpo, caracterizando, portanto, uma função recursiva.

```
(defun fatorial (n)
  (if (= n 0)
      1
      (* n (fatorial (1- n)))
  )
)
```

### Paradigma lógico

O paradigma lógico fornece suporte para sistemas especialistas baseados em inteligência artificial a partir de avaliações lógico-matemáticas, como o estudo da lógica de predicados e regras de inferência para produzir resultados. Sua arquitetura permite modelar um problema de maneira declarativa, baseando-se em um conjunto de regras e restrições do problema, em vez de procedural, expressando-se apenas as especificações dos resultados desejados e não de uma sequência imperativa de instruções para solução do problema.

A LP Prolog é a mais conhecida desse paradigma e utiliza uma máquina de inferência ou mecanismo de dedução automática que é similar à maneira de pensar do ser humano na solução de problemas. Vejamos um exemplo a seguir.



- Se fizer sol hoje, eu vou à praia surfar.
- Está fazendo sol hoje.
- Então, eu vou à praia surfar.

## Paradigma declarativo

O paradigma declarativo apenas declara o que o programa deve fazer, ou seja, qual a tarefa, ao contrário dos paradigmas imperativos, que focam no que deve ser feito. A LP Haskell do paradigma funcional e a LP Prolog do paradigma lógico também se encaixam no paradigma declarativo. Assim, o paradigma declarativo pode envolver vários tipos de linguagens. Nele, o programador declara de maneira abstrata a tarefa a ser feita.

Como modelos, podemos citar as chamadas linguagens de marcação: HTML, XAML, XML e XSLT.

Por exemplo no HTML:

```
<body>  
  
    <h1>Exemplo de Paradigma declarativo</h1>  
  
</body>
```

Outro exemplo a ser mencionado é a linguagem SQL:

```
SELECT nomedacidade FROM cidade WHERE idCidade = 13
```

Consta a seguir um esquema com os diversos paradigmas de linguagem:

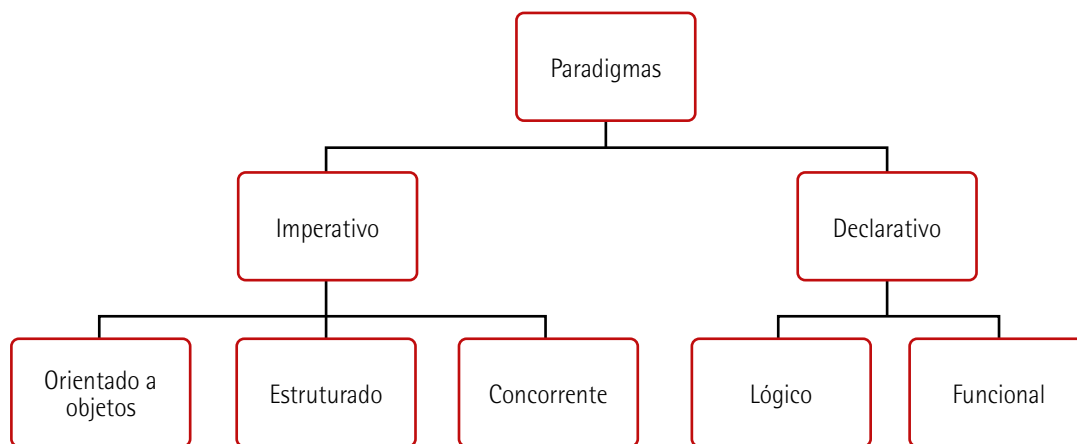


Figura 4 – Paradigmas de linguagem

### Tipos de LPs: alto nível e baixo nível

As LPs têm sido classificadas em dois grupos: linguagens de alto nível e linguagens de baixo nível. Em linhas gerais, podemos dizer que aquelas de alto nível, como Python, podem ser mais fáceis de programar do que as de baixo nível, que estão mais próximas da linguagem de máquina.

As linguagens de baixo nível utilizam linguagem de máquina como instruções a serem executadas pelo processador. A CPU executa nativamente a instrução em linguagem de máquina (ou binária), que consiste em uma sequência de bytes. Sendo que cada byte significa algo para o processador.

Ainda que a linguagem citada esteja representada de forma binária, nem toda sequência binária corresponde à linguagem de máquina. Além disso, seria interessante diferenciar binário de linguagem de máquina, já que a linguagem de máquina é composta das instruções da CPU, e nem toda sequência binária é uma delas.

Por exemplo, a frase "estudo para ter mais conhecimento" está representada a seguir, de acordo com a tabela ASCII, e em binário.

```
01000101 01110011 01110100 01110101 01100100 01101111 00100000 01110000 01100001 01110010  
01100001 00100000 01110100 01100101 01110010 00100000 01101101 01100001 01101001 01110011  
00100000 01100011 01101111 01101110 01101000 01100101 01100011 01101001 01101101 01100101  
01101110 01110100 01101111
```



#### Saiba mais

Com a finalidade de converter frases para código binário, acesse o seguinte site.

INVERTEXO. *Código binário*. [s.d.]. Disponível em: <https://bit.ly/3s20A38>. Acesso em: 21 jan. 2022.

Com a segunda geração das linguagens de baixo nível surgiu o Assembly, que é projetada para um tipo específico de processador, com uma notação já mais legível para humanos, mas que também utiliza instruções de código de máquina que uma arquitetura de computador como microprocessadores e microcontroladores compreende.

Cada instrução utilizada na linguagem Assembly é convertida para o código equivalente em linguagem de máquina, então ao invés de usar uma instrução ilegível para humanos, por exemplo, 01000101, no Assembly empregamos instruções mais fáceis de serem entendidas, como and, or, not, add e div. A instrução "div" faz com que o processador efetue a divisão de duas variáveis. Um exemplo ilustrativo seria "div z, y", que significa dividir os valores de z por y.

Demonstraremos na sequência um exemplo de como fica o código Assembly em determinada arquitetura, já que ele está atrelado a uma arquitetura específica.

mov AX, 10; O registrador AX contém o valor 10

mov CX, 2; O registrador CX contém o valor 2

div CX; A instrução div Divide AX por CX (10/2)

As linguagens de alto nível estão mais próximas da linguagem humana e mais distantes da linguagem de máquina. Ao serem mais bem entendidas pelo programador, se tornam uma linguagem mais abstrata e não estão mais relacionadas diretamente com arquitetura do computador. São exemplos: Python, Java, C#, Ruby, PHP e JavaScript.

A seguir, mostraremos o mesmo código feito em Assembly e convertido para a linguagem C, que é de alto nível.

```
int AX, CX, Resultado;    // Declara as variáveis AX, CX e Resultado como do tipo inteiro;

int main()                // Inicia o método main;
{
    // Início do corpo do programa;
    AX = 10;               // Atribui o valor 10 à variável AX;
    CX = 2;                // Atribui o valor 2 à variável CX;

    Resultado = 10/2;      /* Divide o conteúdo das variáveis AX e CX e armazena o resultado na
variável Resultado */

}                          //Término do corpo do programa;
```

### Classificação das LPs por gerações

Outra forma de classificar as LPs é por gerações, sempre que surgem novas melhorias e recursos são atribuídos.

As LPs de primeira geração manipulam diretamente a arquitetura do computador, uma vez que as instruções são executadas em linguagem de máquina em processadores específicos, ou seja, são programas criados para funcionar em máquinas ou CPU específicas. O nível de abstração dessas primeiras linguagens é inexistente, tornando o seu aprendizado extremamente complexo.

As LPs de segunda geração ainda são consideradas de baixo nível, apesar de estarem mais próximas da linguagem humana, já que usam de instruções mnemônicas, melhorando o nível de abstração. Essas instruções, chamadas de Assembly, são convertidas para linguagem de máquina por um programa que faz a montagem. Ele é denominado Assembler, não confundir com Assembly. O Assembler ou montador contém tabelas que armazenam as instruções do programa que está sendo montado.

Para o desenvolvedor, a curva de aprendizado ainda é enorme, pois além, de precisar decorar os mnemônicos, é necessário que ele possua conhecimento sobre a arquitetura dos computadores.

Já as LPs de terceira geração são consideradas de alto nível, pois suas instruções são mais fáceis de serem entendidas pelo ser humano. Em geral, elas dão suporte a tipos de dados simples, por exemplo, inteiro, real, lógico e caractere, assim como permitem o uso de estrutura de dados, como registros, vetores de matrizes.

Além de suportar instruções repetitivas e condicionais e o uso de operadores, essas LPs permitem o uso de procedimentos e funções em uma programação modular. Essa geração deu início ao uso de interpretadores, compiladores e tradutores para transformar o código em linguagem de máquina. São exemplos: Cobol, Basic, Fortran, Pascal e C.

As linguagens de quarta geração são consideradas de alto nível, pois já permitem que o desenvolvedor especifique o que deseja que o programa faça e não precisa focar em como deve ser feito. Podemos citar como exemplo a linguagem SQL (structured query language), que é utilizada para manipular informações em banco de dados.

As linguagens de quinta geração são aquelas que especificam a solução de problemas de maneira declarativa e não por meio de algoritmo. Por exemplo, a linguagem Prolog, que é utilizada na área de inteligência artificial em sistemas especialistas e também de reconhecimento por voz.

A figura a seguir esquematiza e resume a classificação das LPs por gerações.

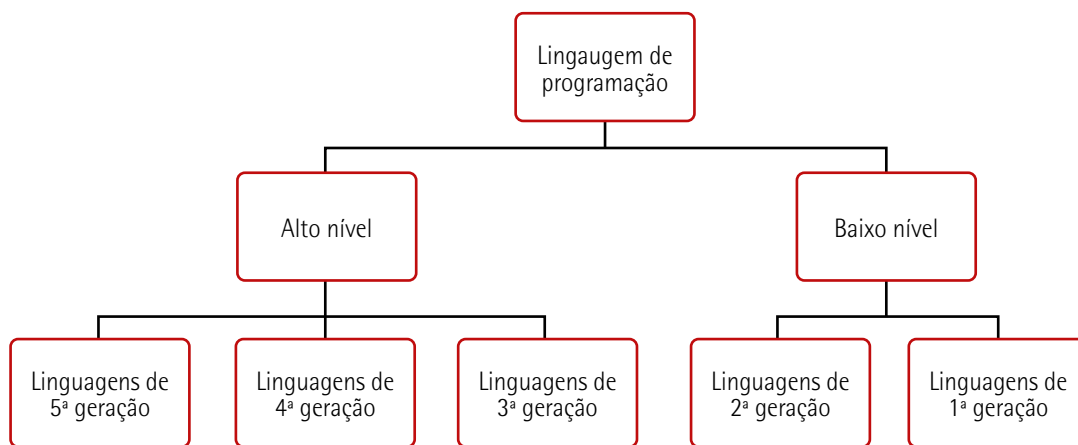


Figura 5 – Classificação das LPs por gerações

### Identificadores em LPs

As LPs fazem o uso de identificadores e símbolos. Identificadores em LPs identificam as diversas entidades em um programa, por exemplo, subprogramas, variáveis, procedimentos, funções e seus parâmetros. Já os símbolos podem ser de operação, operadores e palavras reservadas da LP. Seguem exemplos na sequência:

- **Operação:** soma, subtração, multiplicação e divisão.
- **Operador:** pode ser relacional =, <=, >= e <>; ou lógico (or e not, and).
- **Palavras reservadas da linguagem e seus tipos de dados:** if, else, int, boolean, char.

Via de regra, para definir um nome ideal a um identificador, deve-se utilizar a linguagem ubíqua. Ela usa termos que têm um significado diferente para cada área de negócio ou segmento empresarial. Qualquer área de negócio em uma empresa possui a sua própria terminologia. É importante que elas sejam extraídas da realidade do campo de atuação e utilizadas tanto no código quanto na documentação de software. Isso significa empregar o jargão da área para nomear variáveis, procedimentos, funções e quaisquer outras entidades do programa.

Podemos citar como exemplo do uso da linguagem ubíqua, um procedimento cujo objetivo do negócio seja fazer impressão de boletos bancários, poderíamos nomear o identificador como:

```
public void ImprimirBoletosBancarios():
```

Ou de maneira mais abstrata:

```
public void GerarBoletosBancarios();
```

Dependendo da LP, os identificadores podem ser implementados de várias formas. Algumas delas, como o Java e o C#, são case sensitive, isto é, fazem distinção entre letras maiúsculas e minúsculas. O desenvolvedor precisa prestar atenção nesse detalhe para não prejudicar a legibilidade do programa. Por exemplo, para essas linguagens, Pessoa e pessoa são identificadores distintos que apontam para entidades diferentes. Por outro lado, as linguagens que não implementam o case sensitive tratam o identificador do nosso modelo da seguinte maneira: Pessoa ou pessoa como pertencentes às mesmas entidades, o que também pode gerar problemas de entendimento e legibilidade do código. São exemplos de linguagens que não utilizam case sensitive: Delphi, Pascal e Ada.

É comum nas equipes de desenvolvimento de software a preocupação com a padronização dos identificadores para aumentar a legibilidade e a compreensão do código. Esse trabalho deve ser realizado no código-fonte, na documentação de software e no banco de dados.

Além dos identificadores que são nomeados pelos programadores (variáveis, métodos, parâmetros), existem os identificadores especiais próprios de cada linguagem, que são palavras reservadas que não podem ser nomeadas pelo programador ou utilizadas para outros fins. Temos na linguagem Java os exemplos de for, switch e if; ou Goto e Const em outras LPs.



### Observação

Toda LP possui uma documentação na qual consta o item "Palavras reservadas da linguagem". Elas somente podem ser utilizadas para a finalidade para a qual foram criadas.

### Variáveis em LP

Variável é algo que pode variar; que se muda, altera, é mutável. Em matemática é o termo indeterminado que, em uma função ou relação, pode ser substituído por termos determinados ou valores diversos.

No campo de tecnologia da informação (TI), a variável é uma área de memória, associada a um nome, que pode armazenar valores de determinado tipo de dado.

A importância das variáveis é que elas são responsáveis por armazenar e controlar as informações que trafegam em um programa de computador ou os dados manipulados por um algoritmo. Esses dados são armazenados em um endereço que representa um espaço na memória do computador, de onde posteriormente são acessados ou modificados durante a execução do programa de computador.

Em geral, porque isso pode variar entre as linguagens, uma variável precisa ter um nome ou identificador, um tipo de dado associado a ela e a informação que ela armazena. Algumas linguagens permitem que uma variável possa armazenar diferentes tipos de dados, como é o caso da Python. Outras delas, como C, requerem que o programador declare o tipo da variável. Adicionalmente, uma variável pode ser declarada e não ter nenhum valor associado (o que pode gerar problemas em um programa).

Uma variável, em certas linguagens, pode então ser constituída pelos seguintes atributos:

- nome ou identificador;
- endereço ou posição na memória;
- tipo de dados (texto, decimal, inteiro);
- escopo;
- visibilidade pública, protegida ou privada;
- valor (recebe atribuição de valores ou do resultado de expressões utilizadas em algoritmos).

Sem muitos detalhes técnicos é realizada a amarração do nome ou identificador da variável, com a ajuda do sistema operacional, a seu endereço de memória quando o programa for carregado na memória para a sua execução. Uma variável pode ser global ou local, dependendo do escopo.

As variáveis globais geralmente são definidas no programa principal, porque a partir daí todos os procedimentos podem ter acesso a elas, desde que não seja implementada alguma regra de visibilidade.

Geralmente uma variável local é criada em um procedimento ou subprograma. Nesse caso, ela somente pode ser acessada no próprio procedimento. O endereço de uma variável, seja global ou seja local, pode variar entre as execuções (fato que depende do sistema operacional). Contudo, no caso de alocação estática e de linguagens compiladas, a quantidade de memória alocada é definida em tempo de compilação. No caso da alocação dinâmica, a quantidade de memória pode ser definida em tempo de execução. O endereço dessa variável é criado quando o programa for executado, ou seja, em tempo de execução.

O escopo define as partes do programa nas quais uma variável pode ser acessada, manipulada e ter o seu tempo de vida.

Na LP Visual Basic, temos definida a variável "strGlobalInformeSeuNome". O prefixo "str" é para informar que a variável é do tipo String, o termo Global é apenas para deixar registrado que se trata de uma variável Global. O texto InformeSeuNome identifica o conteúdo contido na variável. Perceba que a variável foi declarada no escopo mais externo do arquivo, e no método Command1\_Click foi atribuído à variável global o valor informado pelo usuário em uma caixa de mensagem do tipo input box e, em seguida, o programa exibe para o usuário o nome que foi indicado.

Veja o exemplo de variável global a seguir:

```
Public GlobalInformeSeuNome As String
Private Sub Command1_Click ()
GlobalInformeSeuNome = InputBox("Informe o seu nome ?")
MsgBox "O nome informado é " & GlobalInformeSeuNome
End Sub
```

### Definição do escopo de visibilidade das variáveis

Escopo é a característica que determina um local onde uma variável pode ser utilizada ou referenciada, como um identificador em um programa. Uma variável declarada em um procedimento é normalmente o local; o contexto define o escopo.

De maneira geral, um escopo pode ter visibilidade estática, feita antes da execução do programa e não muda, ou dinâmica, que troca durante a execução do programa.

No escopo estático, as variáveis são amarradas em tempo de compilação do programa. O termo amarrar pode ser substituído por ligar ou *binding*, trata-se de uma associação entre entidades ou objetos de programação. Por exemplo: uma variável e o valor atribuído a ela, ou um identificador e o seu tipo de dado.

A maioria das LPs imperativas utiliza o escopo estático por causa dos subprogramas, fazendo com que eles possam ser aninhados em outros subprogramas, criando assim uma hierarquia de escopo.

Na programação existe o conceito de blocos de códigos, que é um conjunto de instruções executados em sequência, delimitado por marcadores dependendo da LP, conforme veremos a seguir. Também, em razão do local onde foi declarada a variável, podem ser função, procedimento, classe, bloco, módulo ou estrutura.

### Quadro 1 – Exemplos de delimitadores de blocos

Linguagem	Delimitador de bloco
Ada, Algol e Pascal	Begin/End
Java e C#	{ } (Chaves)
Python	Indentação e espaços em branco

Observe a seguir, nas linhas 1 e 6, as chaves que definem o escopo da classe. Nas linhas 2 e 5, elas estabelecem o escopo do método principal, o main.

Consta a seguir um exemplo de delimitadores de bloco em Java:

```
L01 public class DelimitadoresDeBlocos {  
L02     public static void main(String args[]) {  
L03         System.out.println("As chaves são delimitadores de blocos na LP Java  
L04 e definem o escopo");  
L05     }  
L06 }
```



#### Observação

A linguagem Python foi desenvolvida para proporcionar fácil leitura e visual agradável, evitando pontuações como em outras linguagens.

O Python utiliza espaços em branco e indentação para separar os blocos de código, ao contrário de outras que usam delimitadores visuais, como chaves (Java, C# e C) ou instruções (Pascal, Basic e Fortran). Porém, por não utilizar os delimitadores visuais de blocos, a indentação é obrigatória, uma vez que determina o início e o fim de um novo bloco.



Os blocos de códigos podem ser estruturados de 3 formas: monolítico, blocos aninhados e blocos não aninhados, conforme figura a seguir.

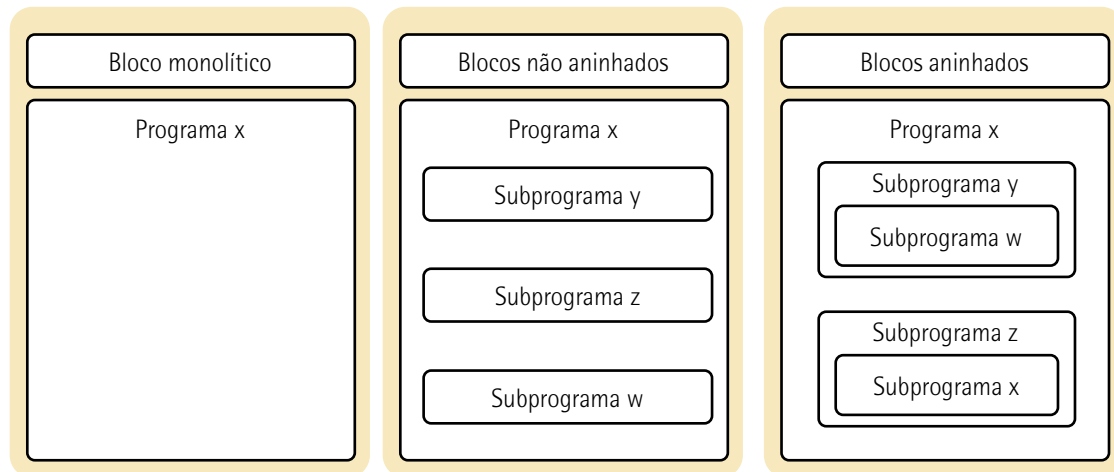


Figura 6 – Tipos de estrutura de blocos

Fonte: Varejão (2004, p. 32).

- **Bloco monolítico:** comum nas versões antigas do Basic e Cobol, o programa todo é codificado em um único bloco. O escopo de visibilidade das amarrações é para o programa todo. Essa estrutura torna-se complicada quando o programa começa a ficar muito grande, pois ela não tem a modularização que divide o código em vários pedaços interligados.
- **Blocos não aninhados:** o programa é subdividido em subprogramas, como na figura anterior, na qual temos os subprogramas y, z e w. O escopo de visibilidade das amarrações é definido pelo bloco onde os identificadores são criados. Eles são chamados de locais, se declarados no bloco, e globais, quando fora do bloco. A LP Fortran utiliza os blocos não aninhados.
- **Blocos aninhados:** programa mais avançado, uma vez que qualquer bloco pode ser criado dentro de outro e assim sucessivamente. Nesse caso, a visibilidade é do escopo ou bloco mais interno do bloco que está sendo utilizado para aquele mais externo ou superior. Essa estrutura é usada pelas LPs Java, C#, C++, C e Pascal.

Consta a seguir um exemplo de bloco aninhado em Java:

```
public class ProgramaX {  
    String variavelProgramaX = "ProgramaX";  
    public static void main(String[] argumentos) {  
        ProgramaX programaX = new ProgramaX();  
        programaX.subProgramaY();  
    }  
}
```

```
public void subProgramaY() {  
    String variavelSubProgramaY = "SubProgramaY";  
    System.out.println(variavelSubProgramaY + " executando. De dentro do " + variavelProgramaX);  
    subProgramaW();  
}  
{  
    System.out.println(variavelProgramaX + " executando.");  
}  
public void subProgramaW() {  
    String variavelSubProgramaW = "SubProgramaW";  
    System.out.println(variavelSubProgramaW + " executando. De dentro do subProgramaY");  
}  
}  
Imprime:  
ProgramaX executando.  
SubProgramaY executando. De dentro do ProgramaX  
SubProgramaW executando. De dentro do subProgramaY
```

No escopo dinâmico, a amarração muda durante a execução do programa, ou seja, quando os procedimentos ou subprogramas vão sendo executados ou conforme o fluxo de controle do programa. São exemplos de LPs que implementam este escopo: Lisp, Perl e Snobol4. O tipo de escopo (dinâmico ou estático) depende da linguagem. Nesse caso, podem ocorrer problemas de eficiência do programa devido à realização da checagem dos tipos de dados durante a execução. Também podem haver problemas de legibilidade, isto é, o programador precisa entender a amarração que foi realizada e os problemas de confiabilidade, pois o subprograma pode acessar variáveis locais do bloco que fez a chamada.

Esse escopo é pouco utilizado pelas LPs, pois ele pode ser facilmente substituído por passagem de parâmetros durante a execução dos procedimentos e métodos.

Consta a seguir um modelo de escopo dinâmico em pseudocódigo em que a variável "informeLetra" tem seu valor substituído em tempo de execução:

```
procedimento procedimentoPrincipal() {  
    string informeLetra = "A";  
    procedimento subProcedimento1() {  
        escreva(informeLetra);  
    }  
    procedimento subProcedimento2() {  
        string informeLetra = "B";  
        subProcedimento1();  
    }  
    subProcedimento2();  
    subProcedimento1();  
}
```

## 1.1.3 Processo de desenvolvimento de programas

Todo o programa ou código-fonte, não importa em qual LP seja desenvolvido, antes de ser executado, tem que ser convertido para a linguagem de máquina por meio de três métodos: compilação ou tradução, interpretação pura e sistema híbrido (SEBESTA, 2011).

A linguagem de máquina gerada a partir do código-fonte deverá ser compatível com hardware e sistema operacional do local onde o programa será executado.

### Compilação ou tradução

Na compilação ou tradução, o programa-fonte, escrito em uma linguagem de alto nível, é traduzido para código executável em uma versão compatível com a linguagem de máquina a qual poderá ser executada diretamente no computador. As LPs Ada, C e Cobol utilizam o processo de compilação.

As fases mais importantes do processo de compilação são exibidas a seguir:

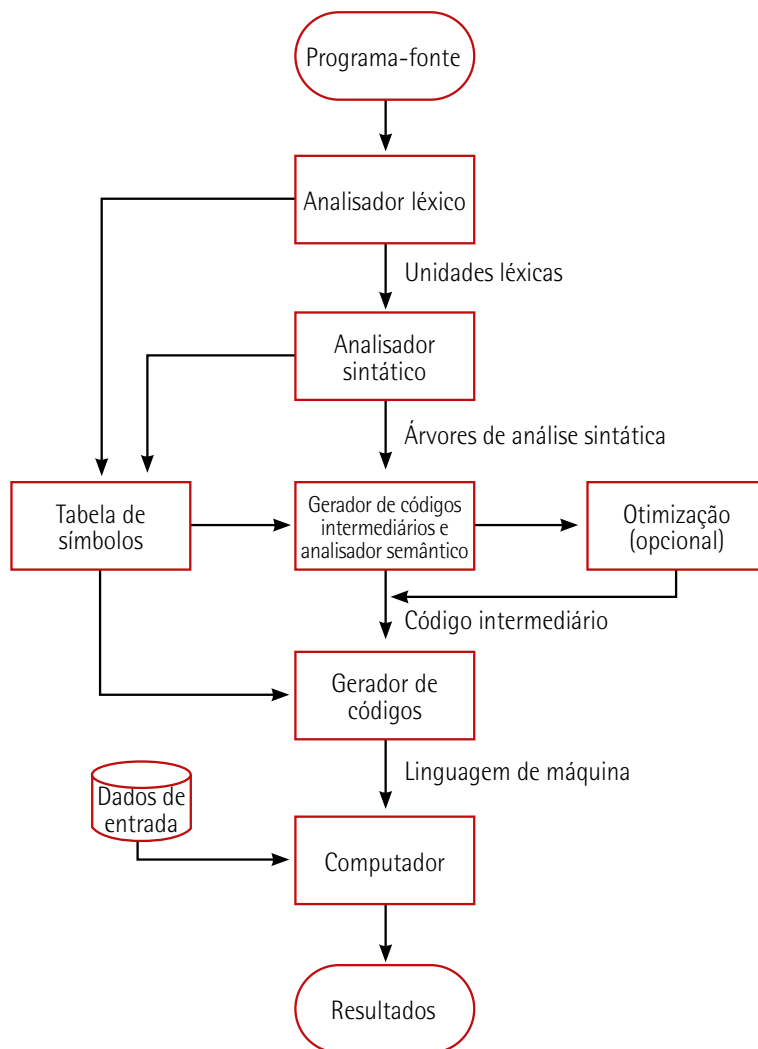


Figura 7 – As fases da compilação

A linguagem que um compilador traduz é chamada de programa-fonte. Suas fases mais importantes são:

- **Analizador léxico:** agrupa os caracteres do programa-fonte em unidades léxicas ou tokens, que são identificadores, símbolos, operadores e palavras especiais, ignorando comentários no programa-fonte.
- **Analizador sintático:** obtém as unidades léxicas do analisador léxico e as utiliza para construir estruturas hierárquicas chamadas de árvores de análise. O analisador aplica regras gramaticais da linguagem sintática e as confere a partir da sequência de tokens, caso esses componham estruturas sintáticas como comandos e expressões.
- **Gerador de código intermediário e analisador semântico:** o gerador de código intermediário cria um programa em uma linguagem diferente, em um nível intermediário entre o programa-fonte e a saída final do compilador. Muitas vezes, o código intermediário pode ficar similar a uma linguagem de montagem ou em um nível mais alto em relação a ela. O analisador semântico é parte do gerador de código intermediário, que verifica erros difíceis de serem detectados durante a análise sintática, como aqueles de tipos em comandos e expressões. Ele checa se as estruturas sintáticas fazem sentido como escopo e uso dos nomes, confere a correspondência entre declarações e uso de variáveis e se há relação entre operadores e operandos.
- **Otimizador de código:** busca aperfeiçoar os programas tornando-os mais rápidos e menores, eliminando redundâncias do código, sendo uma parte opcional da compilação.
- **Gerador de código:** traduz a versão do código intermediário já otimizado do programa em um programa similar em linguagem de máquina.

Para o processo de compilação, a tabela de símbolos serve como uma base de dados na qual estão armazenadas informações de tipo e atributos de cada um dos nomes definidos pelo usuário no programa. Essa informação é inserida na tabela pelos analisadores sintático e léxico e usada pelo analisador semântico e gerador de código.

### Interpretação

Interpretores são programas de computador que utilizam a CPU para traduzir um programa-fonte, um comando de cada vez, de uma LP interpretada, geralmente de alto nível e executada diretamente no programa. Esse processo de software fornece uma máquina virtual para a linguagem (SEBESTA, 2011).

Uma das desvantagens da tradução em relação ao tempo de execução, que é de 10 a 100 vezes mais lenta do que nos sistemas compilados, deve-se ao processo de decodificação das sentenças em linguagem de máquina. Outra desvantagem é que esse processo consome mais espaço. A tabela de símbolos também é utilizada durante a interpretação e o programa-fonte deve ser armazenado em um formato de fácil acesso e modificação. Nos anos 1960, as LPs Lips, Snobol e APL eram puramente interpretadas.

Porém, recentemente, com o desenvolvimento da web, as linguagens de scripting, como PHP e JavaScript, passaram a utilizar o processo de interpretação (SEBESTA, 2011).

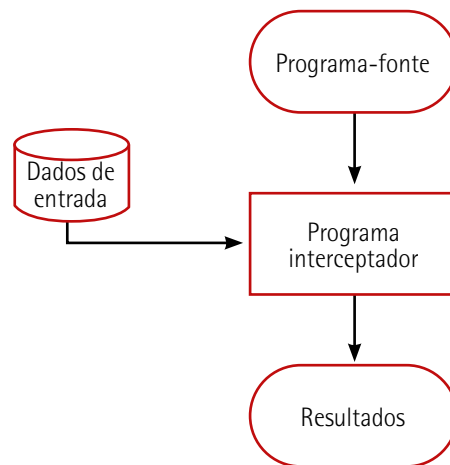


Figura 8 – Interpretação pura

Fonte: Sebesta (2011, p. 49).

O processo de interpretação é similar ao de tradução quando duas pessoas conversam idiomas diferentes, por exemplo, português e espanhol, e não têm domínio do idioma. Então eles precisam de uma terceira pessoa, que é o chamado intérprete, responsável por fazer a tradução ou conversão de um idioma para outro. Entre as principais peculiaridades do interpretador em relação aos programas compilados estão:

- **Simplicidade:** os programas são menos complexos e menores que os compiladores. O resultado da conversão é instantâneo tanto para execução do comando quanto para a exibição do erro.
- **Portabilidade:** o mesmo código pode ser aceito em qualquer plataforma que possua um interpretador.
- **Confiabilidade:** devido ao fato de terem acesso direto ao código do programa e não existirem objetos, eles podem oferecer ao desenvolvedor mensagens de erro mais assertivas.

### Interpretação híbrida

A interpretação híbrida é um meio termo entre os compiladores e os interpretadores puros, ela realiza a tradução de linguagens de alto nível para uma linguagem intermediária, projetada para facilitar a interpretação. Esses sistemas são mais rápidos do que a interpretação pura, porque as sentenças da linguagem-fonte são decodificadas apenas uma vez, e o programa interpreta o código intermediário, em vez de traduzir o código da linguagem intermediária para a linguagem de máquina, conforme a figura a seguir.

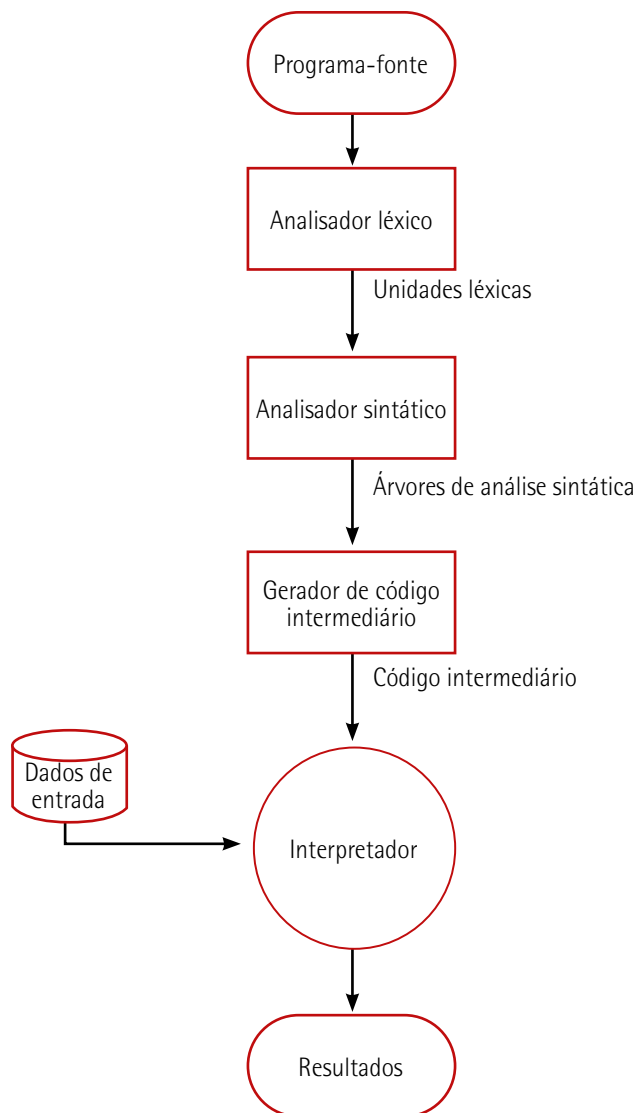


Figura 9 – Processo de interpretação híbrida

Fonte: Sebesta (2011, p. 50).

A interpretação híbrida incorpora o melhor de cada sistema: interpretação, tradução e compilação, combinando a execução rápida dos compiladores com a portabilidade dos interpretadores, gerando um código intermediário que pode ser implementado em qualquer sistema operacional: Windows, Linux, Unix e MAC. Para cada sistema operacional, existe um interpretador apropriado.

A LP Perl é implementada como um sistema híbrido, porém também é parcialmente compilada para detectar erros antes da interpretação. As implementações iniciais do Java eram híbridas, criadas em um formato intermediário chamado de bytecode, elas possuíam a vantagem de fornecer portabilidade para qualquer máquina que tivesse muitos apresentadores de bytecodes; que é um sistema de tempo de execução associado, sendo o conjunto de ambos chamado de máquina virtual Java (JVM). Atualmente, existem sistemas que traduzem diretamente os bytecodes Java para o código de máquina, tornando a execução mais rápida.

A tradução para uma linguagem intermediária é feita pelo sistema de implementação Just-in-Time (JIT), que compila os métodos da linguagem intermediária para a linguagem de máquina durante a execução. Tanto o Java quanto as linguagens .NET são implementadas com um sistema JIT (SEBESTA, 2011).



### Observação

Desenvolvida pela Sun Microsystems, a LP Java é uma linguagem híbrida, porque requer a compilação em bytecode que depois deve ser interpretada, sendo a mais conhecida na atualidade. Tornou-se famosa pelo seu uso no ambiente web. Seu código-fonte, com extensão .java, é traduzido na forma de bytecode, um código intermediário, que pode ser executado por uma máquina virtual Java (JVM) em diversos ambientes operacionais, como Windows, Linux, Unix e MAC.

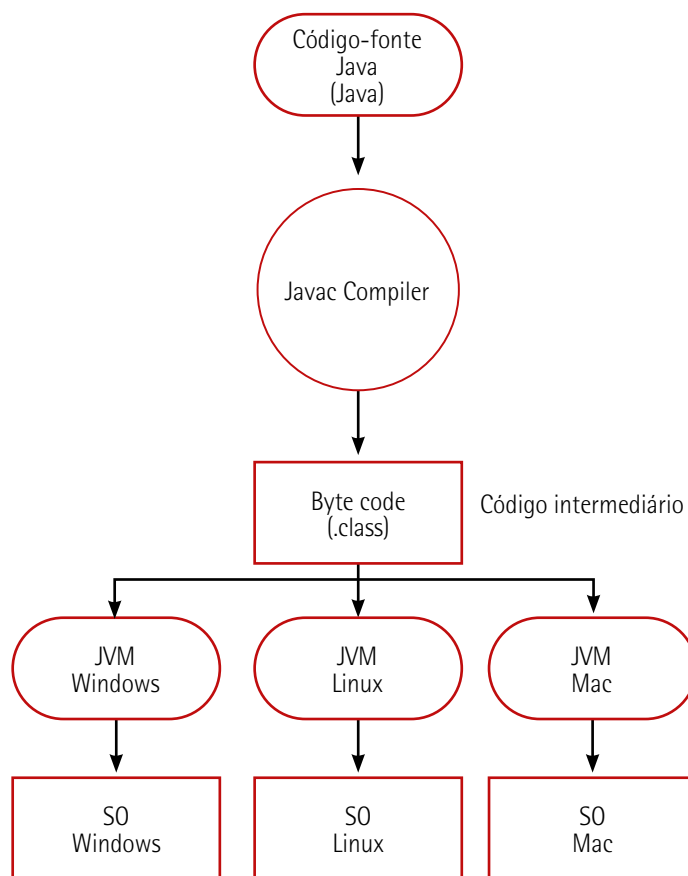


Figura 10 – Processo de tradução Java

Consta a seguir um quadro comparando a compilação ou tradução com a interpretação:

**Quadro 2 – Comparação entre compilação ou tradução e interpretação**

Tradução ou compilação	Interpretador puro	Interpretador híbrido
Gera código executável	Sem geração de código	Geração de código intermediário
Execução rápida. Tradução lenta	Execução lenta independentemente de plataforma	Execução não muito rápida. Tradução rápida
Depende da plataforma de execução	Independência da plataforma de execução	Independência da plataforma de execução

## 1.1.4 Estilo e qualidade de programas

A engenharia de software preocupa-se com o triângulo de ferro, denominado tempo, custo e escopo do projeto de software. Portanto, o tempo para o desenvolvedor é vital durante a criação de software e as LPs devem dar apoio nesse aspecto. Uma LP é confiável se ela proceder conforme a especificação da linguagem.

Algumas das propriedades esperadas para uma LP são:

- **Legibilidade:** é a facilidade com que um código-fonte pode ser lido e entendido. As LPs que são difíceis de serem lidas também são de serem escritas, prejudicando a sua confiabilidade. Quanto mais a linguagem de uma LP for natural, menos complicada e mais fácil será o seu entendimento durante a leitura. A legibilidade afeta a confiabilidade da LP tanto nas fases de desenvolvimento quanto naquelas de manutenção do ciclo de vida. Programas difíceis de ler são também difíceis de escrever e modificar (SEBESTA, 2011).

Por exemplo: se não existir a obrigatoriedade de um marcador em Java para determinar o escopo como as chaves `{ }`, que representam os blocos de inicialização, o código pode ser difícil de ser compreendido, pois fica ilegível, conforme demonstra o código a seguir. O `else` pertence ao primeiro ou ao segundo `if`?

```
if(y > 10)
if(y < 20)
y = 3;
else
y = 100;
```

- **Sintaxe:** tem muita influência na legibilidade, na maneira como ela define suas variáveis e identificadores. Por exemplo, o significado da palavra `static`, na linguagem C, se altera, dependendo do contexto em que foi declarada.



- **Simplicidade:** quando a LP possui um número muito variado de componentes, tende a ser subutilizada, o que afeta negativamente a legibilidade. Isto é, na linguagem Java, um programador pode incrementar uma variável de 4 maneiras diferentes, diminuindo a legibilidade. Por exemplo:

```
contador = contador + 1;  
contador++;  
++contador;  
contador +=1;
```

- **Redigibilidade e simplicidade:** uma LP deve propiciar ao desenvolvedor a facilidade de se concentrar nos algoritmos centrais de um programa para resolução do problema de negócio e não se preocupar com aspectos não relevantes, por exemplo, nos detalhes de implementação. Ou seja, o programador deve focar em como é que o programa vai se comportar no hardware.
- **Eficiência:** a LP deve fornecer meios adequados para que determinado tipo de aplicação possa atingir os seus objetivos, por exemplo, a linguagem Python dá suporte para aplicações de inteligência artificial e ciência de dados.
- **Facilidade de aprendizado:** uma LP deve fornecer suporte para que o profissional de TI, que trabalha desenvolvendo softwares para computadores, tenha condições de assimilar rapidamente seus conceitos e utilização. Por exemplo, o tempo da curva de aprendizado entre as LPs Assembler e Java é rápido.
- **Ortogonalidade:** é quando não existe uma relação de dependência entre componentes. Se um componente for alterado, essa alteração não vai afetar outros. A LP deve permitir a combinação de conceitos básicos sem que se produzam efeitos atípicos, o ideal é que ela tenha muita ortogonalidade. A falta de ortogonalidade gera exceções às regras da linguagem, fato que gera falta de confiança no desenvolvedor. Por exemplo, um caso clássico de falta de ortogonalidade na linguagem C pode ser observado quando há dois tipos de dados estruturados, struct e arrays. O struct pode ser retornado em uma função, enquanto o arrays não.
- **Reusabilidade:** possibilidade que um software tem de poder ser reutilizado ao todo ou em parte por outras aplicações, aumentando sua flexibilidade e portabilidade e reduzindo os custos de desenvolvimento e manutenção.
- **Modificabilidade:** permite a modificação do programa em função de novos requisitos ou a alteração do que já foi implementado. Os mecanismos de construção de subprogramas, interface e tipos de dados abstratos possibilitam uma boa modificabilidade no software.
- **Portabilidade:** é influenciada pelo grau de padronização da LP. Trata-se da capacidade de um programa de se comportar de maneira similar, independentemente da arquitetura computacional, hardware ou sistema operacional sobre a qual está sendo executado. Um exemplo seria um programa que pudesse ser executado tanto em uma máquina Linux quanto em outra Windows.

- **Tipos de dados e estruturas:** uma LP deve permitir a implementação da estrutura de dados e dos tipos de dados, aumentando a legibilidade do programa.
- **Verificação de tipos:** uma LP deve possibilitar a detecção de erros de tipos tanto durante a execução do programa quanto no momento da compilação. Quanto mais uma LP realiza a verificação de tipos de erros em tempo de compilação, mais confiável ela se torna. Por exemplo, no Java, a verificação de expressões e variáveis é feita em tempo de compilação; tal funcionalidade praticamente elimina o erro em tempo de execução. A atribuição de uma String a uma variável do tipo inteira deve resultar em erro. As chamadas linguagens fortemente tipadas aumentam a confiabilidade de uma LP, pois somente os valores do tipo declarado são atribuídos às variáveis.
- **Tratamento de exceção:** evidentemente que uma LP será mais confiável se conseguir interceptar erros em tempo de execução, promover facilmente medidas corretivas e permitir que o programa continue sem ser interrompido devido à ocorrência de erros, tornando o comportamento de um programa mais previsível. Alguns exemplos de LPs que implementam o tratamento de exceções são C++, C# e Java (SEBESTA, 2011).
- **Custo:** é medido em razão de várias de suas características quando os programadores precisam ser treinados para utilizar a LP. O custo de escrita da linguagem depende da facilidade de sua escrita. As linguagens de alto nível foram criadas para diminuir o custo da criação de software. Tanto o custo de escrever programas quanto o de treinar programadores pode ser reduzido quando um ambiente de programação é bem otimizado e padronizado.

Existe também o custo de compilar programas em uma LP uma vez que ela necessita de muitas operações de verificações de tipo, o que ocasionará uma execução menos rápida na checagem do código. Atualmente a eficiência da execução do código é considerada menos importante do que a compilação. Isto se deve ao fato de os programadores geralmente compilarem seus programas várias vezes, e o executarem poucas vezes.

O custo de implementação da linguagem influencia na sua utilização. Por exemplo, a linguagem Java teve um grande sucesso de aceitação devido aos seus sistemas de compilação e interpretação serem gratuitos e disponibilizados ao público. A confiabilidade em uma LP também é um custo, ou seja, no caso de falha de um sistema crítico, como sistemas de exame de saúde, o custo pode ser muito alto. O custo de manutenção de software depende de uma série de características da linguagem, sobretudo da legibilidade, devido ao fato de normalmente ela ser feita por desenvolvedores diferentes daqueles que criaram o sistema. De todos os fatores citados, três são os mais importantes: confiabilidade, desenvolvimento de programas e manutenção.

O quadro a seguir exibe um resumo dos critérios de avaliação mais importantes das LPs. Observe que alguns deles, como facilidade de escrita, são muito amplos, enquanto outros, como tratamento de exceções, são mais específicos (SEBESTA, 2011).

**Quadro 3 – Critérios de avaliação de LPs**

Característica	Legibilidade	Facilidade de escrita	Confiabilidade
Simplicidade	X	X	X
Ortogonalidade	X	X	X
Tipos de dados	X	X	X
Sintaxe	X	X	X
Abstração		X	X
Expressividade		X	X
Verificação de tipos			X
Tratamento de exceções			X
Apelidos			X

Fonte: Sebesta (2011, p. 26).

## 1.2 Tipos de dados

Tipos de dados são uma coleção de valores de dados e um conjunto de operações que podem ser realizadas sobre eles. Programas de computador produzem informações e conhecimento por meio da manipulação dos dados. Quantos mais tipos e estruturas de dados estiverem disponíveis, em uma LP, para representar problemas do mundo real, mais facilidade ela terá para realizar suas tarefas e cumprir com seus objetivos (SEBESTA, 2011).

Um programa de computador ao ser executado precisa guardar na memória as seguintes informações de determinado produto: nome, se ele foi descontinuado ou não, e seu preço. O nome do produto é uma sequência de caracteres, se foi descontinuado é uma informação do tipo verdadeira/falsa e o preço contém um valor numérico. Para armazená-los, o desenvolvedor precisa definir variáveis, cada uma de determinado tipo de dados, por exemplo, a informação "descontinuado" poderia ser do tipo "booleano", o preço, por ser um valor monetário, poderia ser do tipo "double" e assim sucessivamente.

Um dado é uma informação bruta que permite que se chegue a um conhecimento de algo. Para obter uma informação, é necessário recuperar e manipular um dado. Por exemplo, uma data é um dado que por si não representa nada relevante, e não conduz a nenhum entendimento. Porém, se, em um sistema essa data for utilizada, organizando-a na forma de data de aniversário de um cliente, com o objetivo de lhe enviar um e-mail parabenizando-o, nesse contexto, ela passa a transmitir um significado e se torna uma informação com alto valor agregado.

Armazenar dados temporariamente em memória é fundamental para que um sistema alcance o seu objetivo, que é permitir aos desenvolvedores criar programas que recebam dados e os transformem em informação útil para a sociedade.

As LPs devem fornecer um conjunto de tipos de dados, componentes de estrutura de dados mais complexos, propriedades, constantes e variáveis. É fundamental que o desenvolvedor conheça esses recursos, saiba quais tipos de dados podem ser aplicados, além de comandos e formas de particionar um programa em subprogramas.

A LP que fornece suporte a uma grande variedade de tipos de dados amplia as suas possibilidades de armazenamento e manipulação, além de disponibilizar as operações para estruturar e transformar esses dados em informação.

Os tipos de dados são a descrição de um valor ou de uma variável armazenada na memória do computador, sendo que é preciso definir o tamanho que será ocupado dessa memória, como esse valor será representado e quais as operações que poderão ser feitas com essa variável. Por exemplo, na linguagem Java um tipo de dados inteiro pode ser dividido em quatro tipos primitivos, com cada um deles ocupando um espaço de armazenamento diferente, conforme demonstrado na coluna "Quantidade de bits" a seguir:

**Quadro 4 – Tipos inteiros em Java**

Tipo de dado	Quantidade de bits	Faixa de valores
int	32	-2147483648 até 2147483647
long	64	-9223372036854775808 até 9223372036854775807
short	16	-32768 até 32767
byte	8	-128 até 127

### 1.2.1 Tipos de dados primitivos

Os tipos de dados primitivos são suportados diretamente pelo compilador da LP, estão presentes em praticamente todas as LPs e representam os tipos mais simples. Os valores dos tipos de dados primitivos não podem ser decompostos em outros de tipos mais simples. Eles são a estrutura de todo o sistema de tipos de dados de uma LP, porque a partir deles constroem-se os demais, que em muitas LPs são chamados de tipos construídos.

Os tipos de dados primitivos se dividem em:

- **Tipos numéricos (inclui inteiro, ponto flutuante, decimal):** notas da faculdade (9,5; 8,5; 7,5); valores monetários (R\$ 100,50); e o rendimento da poupança (0,75%, 1,21%).
- **Tipos booleanos:** false e true (0 ou 1).
- **Tipos caractere:** "este é um exemplo de tipo de dado formado por caracteres".

O tipo de dados numérico (inteiro) é o mais comum encontrado nas LPs. Em geral, elas suportam diversos tamanhos de inteiros, como na linguagem C, conforme demonstrado a seguir.

**Quadro 5 – Tipos de dados inteiros em C**

Tipo de dado	Conceito	Tamanho (em bytes)	Intervalo de valores
int	Inteiro	2 (16 bits)	de -32.768 a 32.767
		4 (32 bits)	de -2.147.483.648 a 2.147.483.647
unsigned int	Inteiro não assinado	2 (16 bits)	de 0 a 65.535
		4 (32 bits)	de 0 a 4.294.967.295
short int	Inteiro curto	2	de -32.768 a 32.767
unsigned short int	Inteiro curto não assinado	2	de 0 a 65.535
long int	Inteiro longo	4	de -2.147.483.648 a 2.147.483.647
unsigned long int	Inteiro longo não assinado	4	de 0 a 4.294.967.295

Java é considerada uma linguagem fortemente tipada, pois deve-se declarar o tipo de dados antes de utilizá-los. Por exemplo em:

- `int quantidadeDeltens;`
- `quantidadeDeltens = 5;`



## Observação

Na LP Java existem somente os seguintes dados primitivos: Char, Boolean, int, long, float, double, short e byte.

A maioria das LPs dá suporte ao tipo de dados numéricos real, que utiliza ponto flutuante (ponto decimal ou vírgula), normalmente chamado de float e double. Ou seja, utilizamos quando precisamos declarar variáveis para armazenar valores monetários por exemplo. Internamente, a LP combina as representações binárias com frações e expoentes, como se fosse uma notação científica.

**Quadro 6 – Tipos de dados flutuante em Java**

Tipo de dado	Conceito	Tamanho (em bytes)	Intervalo de valores
float	Flutuante real	4	$3.4 \times 10^{-38}$ a $3.4 \times 10^{38}$
double	Flutuante duplo	8	de $1.7 \times 10^{-308}$ a $1.7 \times 10^{308}$
long double	Flutuante duplo longo	10	$3.4 \times 10^{-4932}$ a $3.4 \times 10^{4932}$

Também temos o tipo de dados numéricos decimal, suportado por algumas LPs. A maioria dos computadores de grande porte projetados para suportar aplicações de sistemas de negócios tem suporte em hardware para esse tipo de dados. Trata-se de um tipo primitivo que armazena um número fixo de dígitos decimais com o ponto decimal em uma posição fixa no valor, a vantagem é que se sabe exatamente a quantidade de dígitos da parte decimal fixa, ou seja, a quantidade de dígitos após a virgula. Esses são os tipos de dados primários para processamento de dados de negócios e, assim, são essenciais para o Cobol. A LP C# também possui um tipo de dados decimal.

Os valores em ponto flutuante são escritos em notação científica ou notação decimal, por exemplo, os seguintes valores representam o número 3.14 (TUCKER; NOONAN, 2009):

- 0.000314e4 (notação científica)
- 3.14 (notação decimal)

O tipo de dados booleano existe praticamente em todas as LPs, com exceção da LP C, e é considerado o mais simples de todos, pois é um recurso muito utilizado como condição para executar tarefas em algoritmos. Ele costuma ser representado por dois estados: falso e verdadeiro, ou false e true, ou zero e um. Ocupa apenas um byte da memória do computador.

O tipo de dados caracteres é armazenado nos computadores como codificações numéricas. A codificação mais comum é ASCII de 8 bits, que usa os valores de 0 a 127 para codificar 128 caracteres diferentes. O ISO 8859-1 é outra codificação de 8 bits para caracteres, mas ele permite 256 caracteres diferentes (SEBESTA, 2011).

Temos na sequência um exemplo de implementação de uso dos tipos primitivos em Java. Observe que o tipo de dados char é sempre atribuído utilizando-se aspas simples. Para os tipos float, utiliza-se o caractere "f" no final do valor para sua identificação, assim como para o tipo de dados long, que usa o "L".

```
public class TiposPrimitivosJava {
    public static void main(String[] args) {
        char tipoDadoChar = 'Char';
        boolean tipoDadoBooleano = true;
        byte tipoDadoByte = 127;
        short tipoDadoShort = 32767;
        int tipoDadoInt = 2147483647;
        long tipoDadoLong = 9223372036854775807L;
        float tipoDadoFloat = 3.5f;
        double tipoDadoDouble = 4.50;

        System.out.println("Valor do tipo de dados char = " + tipoDadoChar);
        System.out.println("Valor do tipo de dados booleano = " + tipoDadoBooleano);
        System.out.println("Valor do tipo de dados byte = " + tipoDadoByte);
        System.out.println("Valor do tipo de dados short = " + tipoDadoShort);
        System.out.println("Valor do tipo de dados int = " + tipoDadoInt);
        System.out.println("Valor do tipo de dados long = " + tipoDadoLong);
        System.out.println("Valor do tipo de dados float = " + tipoDadoFloat);
        System.out.println("Valor do tipo de dados double = " + tipoDadoDouble);
    }
}
```

## 1.2.2 Tipo string de caracteres

As strings são essenciais em programação, sendo suportadas pela maioria das LPs (TUCKER; NOONAN, 2009).

Um tipo de dados de cadeia de caracteres é aquele no qual os valores consistem em sequências de caracteres. Ele é utilizado em saídas e entradas do computador para todos os tipos de dados realizados em termos de cadeias. As cadeias de caracteres são um tipo essencial para os programas que realizam manipulação de texto (SEBESTA, 2011). Por exemplo, em C:

- `char cumprimentando[] = "Bom dia!";`

Algumas LPs se referem a esse tipo de dados, como tipos primitivos, outras como tipos construídos ou novos tipos, há ainda aquelas que os classificam como uma lista, e assim por diante. Algumas operações comuns para este tipo de dados são atribuição e comparação, concatenação e seleção.

## 1.2.3 Tipos ordinais definidos pelo usuário

No tipo ordinal, uma faixa de valores possíveis pode ser facilmente associada com o conjunto dos inteiros positivos. Por exemplo, na LP Java, os tipos ordinais primitivos são `char`, `integer` e `boolean`. Existem dois tipos ordinais definidos pelo usuário e suportados pelas LPs enumerações e subfaixas (SEBESTA, 2011).

No tipo enumeração podemos utilizar todos os valores, os quais se tornam constantes simbólicas e são enumerados na definição, por exemplo, em C#:

- `enum diasDaSemana {segunda, terça, quarta, quinta, sexta, sábado, domingo};`

As constantes de enumeração são preenchidas implicitamente por atribuições de valores inteiros (0, 1, 2...6, como exibido anteriormente).

Consta na sequência outro exemplo de como declarar constantes na LP C:

```
#define Brasil 0
#define Italia 1
#define Alemanha 2
#define Franca 3
```

- `define Brasil 0`
- `define Italia 1`

- define Alemanha 2
- define Franca 3

Definindo o enum conforme a seguir:

```
enum PaisesDaCopa
{
    BRASIL,
    ITALIA,
    ALEMANHA,
    FRANCA
};
```

O tipo subfaixa é uma subsequência contígua de um tipo ordinal, por exemplo, 12..14 é uma subfaixa do tipo de dados inteiro. Os tipos subfaixa foram introduzidos na LP Pascal e incluídos em Ada. Em Ada, elas são incluídas na categoria chamada de subtipos. Esses subtipos não são novos tipos, em vez disso, são novos nomes para versões possivelmente restritas de tipos existentes (SEBESTA, 2011).

Veja a seguir um exemplo em Pascal:

```
program primeiro;
type faixaMaiuscula = 'A' .. 'Z';
var letra: faixaMaiuscula;
begin
    letra:= 4;
    writeln(letra); //exibe 'D', índice começa em 1
end.
```

As vantagens da utilização dos tipos ordinais definidos pelo usuário são a legibilidade, pois os valores nomeados são facilmente reconhecidos, enquanto os codificados não, e a confiabilidade, uma vez que nenhuma operação aritmética é permitida nem qualquer variável de enumeração pode ter um valor atribuído a ela fora da faixa definida.

### 1.2.4 Tipos array

Em programação, o tipo array é uma estrutura de dados cujos elementos podem ser acessados ou manipulados por meio de índices. Há possibilidade de ele possuir uma ou mais dimensões, que devem ser do mesmo tipo, por exemplo: int, string, integer, char ou real.

Consta a seguir um exemplo de array com 5 elementos:





Figura 11

Matrizes e vetores são a mesma coisa, a diferença é que vetor é um array de apenas uma dimensão, composto de uma linha e uma coluna de dados, enquanto matriz é um array de duas ou mais dimensões ou um vetor multidimensional, na qual temos várias linhas e colunas, além de outras dimensões. Uma matriz pode ser representada por uma tabela com linhas e colunas que armazenam valores do mesmo tipo de dados conforme figura a seguir.

int [2] [5]

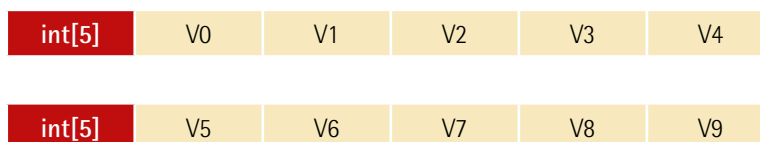


Figura 12 – Matriz com várias dimensões

Por exemplo, em C#, um array de uma dimensão:

```
string[] diasDaSemana = {"segunda", "terça", "quarta", "quinta", "sexta", "sábado", "domingo"};
```

Ou em um modelo de array multidimensional em Java. Observe a sintaxe da linguagem, que no caso desse array a maioria das LPs utiliza colchetes para definir o tamanho das dimensões, cujo resultado pode ser visto na sequência:

```
int[][] arrayMultidimensional = { { 10, 20 }, { 30, 40 }, {50,60}};
```

**Tabela 2 – Resultado do array de duas dimensões**

10	20
30	40
50	60

O array é utilizado quando precisamos que uma variável armazene muitos valores de um mesmo tipo. A fim de que ele resolva esse problema computacional, é necessário que armazene os cem números, para posteriormente realizar a impressão na ordem inversa.

Para tanto, poderiam ser criadas cem variáveis distintas e realizada a impressão delas na ordem inversa. Mas a utilização de um array possibilita um código mais legível, com menos custo de tempo. Além disso, em um cenário em que, em vez de cem, a quantidade necessária fosse mil, seria inviável solucionar o problema com simples variáveis.

A seguir o exemplo de um programa que faz a leitura de cem números e os imprime na ordem inversa:

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {

    int totalNumeros[100];
    int contador;
    printf("Informe 100 números inteiros: ");
    for (contador = 0; contador < 100; contador++) {
        scanf("%d", &totalNumeros[contador] );
    }
    printf("Imprimir números na ordem inversa:\n");
    for (contador = 99; contador >= 0; contador--) {
        printf("%d ", totalNumeros[contador]);
    }
    return 0;
}
```

A declaração de array segue a mesma sintaxe da declaração de variáveis, a diferença está no valor informado entre colchetes, que estabelece quantos elementos serão armazenados, em outras palavras, a dimensão ou o tamanho.

Na sequência exibiremos um exemplo desenvolvido na LP C com a declaração de uma array de uma dimensão com 100 elementos (do tipo int), ele possui índices de 0 a 99 e duas dimensões, ou seja, uma matriz 5 por 4 de valores do tipo int (20 valores no total). Os índices da primeira dimensão vão de 0 a 4, enquanto aqueles da segunda vão de 0 a 3.

Observe que as faixas de índice estão estaticamente vinculadas e a alocação de armazenamento declarada antes da execução tem como vantagem a eficiência, pois no caso alocação e desalocação não são necessárias. Porém, tem como desvantagem ser necessário saber o tamanho limite antes da execução do programa e o armazenamento declarado permanece fixo durante toda sua execução.

A seguir declarações de uma e de duas dimensões:

```
int arrayDeUmaDimensao [100];
int matrizDeDuasDimensoes [5][4];
```

Dependendo do tipo de dado declarado, a matriz ocupa determinado espaço na memória do computador. No exemplo, a matriz de 100 elementos do tipo int requererá em uma plataforma cujo inteiro ocupa 2 bytes,  $100 * 2$  bytes, um total de 200 bytes de memória. Se, por exemplo, o tipo de dados declarado for float, ela passa a ocupar 400 bytes de memória, ou seja,  $100 * 4$  bytes.

A sintaxe se altera em razão da LP. Veja agora o exemplo em Pascal da expressão "array of:" são palavras reservadas da linguagem, com significado predefinido.

```
var a: array[1 .. 100] of real;
```

As LPs Java, C#, C e C++ permitem a inicialização quando o armazenamento é alocado, como no exemplo:

- `int fibonacci [] = {0, 1, 1, 2, 3, 5, 8};` (LP C#).
- `char *sobrinhos [] = {"Huguinho","Zezinho","Luizinho"};` (LP C).
- `String[] contando = {"um", "dois", "três"};` (LP Java).

A maioria das LPS dá suporte a matrizes, pois elas permitem a inclusão de todos os tipos ordinais como possíveis tipos de índices e matrizes dinâmicas.

### 1.2.5 Tipos registro

Tipos registro é uma estrutura de dados composta e heterogênea que permite armazenar valores, que podem ser de diferentes tipos.

Um registro é um agregado de elementos de dados no qual os elementos individuais são denominados por nomes e acessados por meio de deslocamentos a partir do início da estrutura do dado. Normalmente, utiliza-se struct em LPs orientadas a objetos, por exemplo, C e C++.

Os registros são projetados para ajudar o programador a modelar coleções de dados que não possuem o mesmo tipo de dados. Por exemplo, informações sobre produto podem conter o nome, o código e o preço do produto, entre outras. Na coleção podemos utilizar um tipo cadeia de caracteres para o nome, um tipo inteiro para o código do produto, um ponto flutuante para o preço e assim sucessivamente (SEBESTA, 2011).

Cada elemento individual é identificado por seu nome e acessado por meio de deslocamentos a partir do início da estrutura.

A seguir um exemplo de declaração em C:

```
struct Produto{  
    string nomeProduto;  
    int codigoProduto;  
    float precoProduto;  
} reg_produto;
```

Nas LPs C#, C++ e C são suportados pelo tipo struct, e em Pascal pelo tipo record.

O tipo registro faz referência a campos, sendo que, na maioria das linguagens, usa-se um ponto na notação. Por exemplo:

- NomeRegistro.NomeCampo

Na LP Pascal, temos o seguinte exemplo para armazenar a data do descobrimento do Brasil, 07/09/1822, na variável desc:

```
desc.dia := 7; desc.mes := set; desc.ano := 1822
type Data = record
dia : 1..31;
mes : (jan, fev, mar, abr, mai, jun, jul, ago, set, out, nov, dez);
ano : integer
end;
var desc : Data;
```

As matrizes e registros são fortemente relacionados com formas estruturais. As matrizes são aplicadas quando todos os valores dos dados pertencem ao mesmo tipo de dados e são processados da mesma maneira. Quando os dados são heterogêneos, ou seja, diferentes, utiliza-se registro, pois os campos diferentes não são processados da mesma forma. Nesse caso, os campos são processados em determinada ordem particular.

### 1.2.6 Tipos ponteiro

Um tipo ponteiro é um valor que representa uma referência ou um endereço de memória. Na prática, os ponteiros são uma espécie de apontadores, ou seja, variáveis que guardam o endereço de memória de outras variáveis. Neles, as variáveis têm uma faixa de valores que consiste em endereços de memória e um valor especial, Null (Linguagem C). O valor Null não é um endereço válido e é usado para indicar que um ponteiro não pode ser usado no momento para referenciar uma célula de memória. Os ponteiros são utilizados normalmente nas LPs C, C++, Ada e Perl. A LP C fornece duas operações com ponteiros:

- **Atribuição:** é utilizada para fixar o valor de uma variável de ponteiro em um endereço útil. Ela define ou inicializa a variável com o operador Unário "Endereço de" (&); além de receber uma variável como argumento e retornar o endereço dessa variável.
- **Desreferenciamento:** aponta para o valor da célula de memória, não apenas o endereço pode ser implícito ou explícito. A LP C++ usa uma operação explícita asterisco (\*) Operador Unário de "Desreferenciação" (\*): Recebe uma referência e produz o valor dela.

Consta a seguir um exemplo de ponteiro em C:

```
#include <stdio.h>
#include <conio.h>
int main(void)
{
    //numerador é a variável que será apontada pelo ponteiro

    int numerador = 13;

    //declaração da variável do tipo ponteiro. O operador * indica que a variável é do tipo ponteiro

    int *ptr;

    //atribuindo o endereço da variável numerador ao ponteiro. O operador & indica que estamos nos
    //referindo ao endereço da variável numerador e não ao conteúdo desta.

    ptr = &numerador;

    printf("Exemplo de ponteiros\n\n");
    printf("Conteúdo da variável numerador: %d\n", numerador);
    printf("Endereço da variável numerador: %x \n", & numerador);
    printf("Conteúdo da variável ponteiro ptr: %x", ptr);
    getch();
    return(0);
}
```

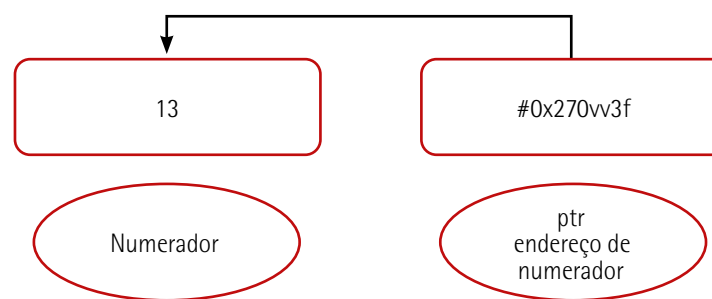


Figura 13

Ponteiros são muito úteis quando existe a necessidade de que uma variável seja acessada de diferentes lugares do programa. Também podem ser utilizados para retornar mais de um valor em uma função, manipular arrays, servir de referência para pilhas, listas, árvores e grafos, além de manipular alocação dinâmica de memória.

### 1.3 Expressões e instruções de atribuição

Conhecemos anteriormente a importância da aplicação de tipos de dados nas LPs. Agora exibiremos como construir expressões nas LPs, a aplicação de tipos de dados nas expressões, suas operações e conceitos de expressões simples e compostas.

#### 1.3.1 Expressões aritméticas

As primeiras LPs tinham como meta básica a avaliação automática de expressões aritméticas. Essas expressões precisam ser avaliadas pela LP para produzir resultado. Elas são o meio básico de especificação das instruções em uma LP. É importante que o desenvolvedor entenda que, para avaliar uma expressão, é necessário conhecer a ordem de avaliação de operadores e de operandos na respectiva LP que está estudando. O objetivo de uma expressão aritmética é especificar uma instrução executando operações aritméticas sobre os seus operandos.

##### Operadores e operandos

Os operadores formam os tipos de operações que podem ser realizados nas expressões, conforme segue:

- soma (+), Exemplo:  $2 + 2 = 4$ ;
- subtração (-), Exemplo:  $5 - 3 = 2$ ;
- multiplicação (\*), Exemplo:  $3 * 2 = 6$ ;
- divisão (/), Exemplo:  $8 / 4 = 2$ ;
- módulo (%), Exemplo:  $5 \% 4 = 1$ ;



##### Observação

O operador módulo é aquele que serve como atalho para a obtenção do resto de uma divisão.

Um operando pode ser entendido como uma das entradas de um operador.

Por exemplo, na expressão  $10 / 5$ ,

- / é o operador.
- 10 e 5 são os operandos.

## Tipos de operadores

Os operadores podem ser unários (1 operando), binários (2 operandos) ou ternários (3 operandos).

- **Operador unários:** ocorre quando há um único operando (+, ++). No exemplo, temos apenas o i.

Por exemplo: i++.

Algumas LPs como C#, Java, C, JavaScript e Perl possuem dois operadores aritméticos unários especiais que são atribuições abreviadas combinando operações de incremento (++) e decremento (--) com atribuição. Esses operadores podem ser utilizados para formar sentenças de atribuição autocontidas de um único operador ou em expressões. Eles podem ser utilizados como operadores pré-fixados (++i), uma vez que precedem os operandos, ou como operadores pós-fixados (i++), que sucedem os operandos.

O exemplo a seguir ilustra o operador unário de incremento para formar uma sentença de atribuição completa:

```
contador ++;
```

Essa operação corresponde à sentença: `contador = contador + 1;`

- **Tipos registro:** possui dois operandos (-, +, \*, /). No exemplo a seguir, temos i e j: Ex: `i + j`
- **Operador ternário:** possui três operandos (? :). No modelo seguir temos que se i for maior que j, então ele retorna um, senão ele retorna zero:

Ex.: `(condição) ? Verdadeiro : Falso`

`( i > j ) ? 1 : 0`



### Lembrete

As LPs C#, Java, C e C++ implementam o operador ternário. Observe que esse recurso é uma alternativa elegante para o uso da expressão IF, utilize-o sempre que surgir a oportunidade para ter um código mais otimizado.

## Tipos de expressões

Existem diversos tipos de expressões, que são classificadas de acordo com os seus operandos, a natureza da operação realizada e o resultado produzido. Citaremos aqui as mais comuns encontradas nas LPs.

Os tipos de expressões aritméticas são os mais conhecidos, pois são similares às expressões matemáticas que trabalham com valores numéricos: decimal, inteiro e ponto flutuante. Por exemplo, os operadores aritméticos mais simples em Java são: soma (+), subtração (-), divisão (/) e multiplicação (\*).

Como exemplo de expressão aritmética podemos usar  $5 - 2$ , que terá como resultado o número 3 (como dito é bem semelhante às expressões matemáticas).

Para calcular a área de um quadrado, onde o lado é igual a 3:

```
int areaDoQuadrado = 3 * 3;
```

Pondere o trecho de código que será utilizado na sequência, implementado em Java, e observe que temos dois resultados: um sem e outro com casas decimais.

A seguir um exemplo de operadores aritméticos em Java:

```
public class ClasseOperadorAritmetico {  
    public static void main(String args[]) {  
        double numeroPontoFlutuante;  
        int numeroInteiro = 13;  
        numeroPontoFlutuante = numeroInteiro * 5;  
        System.out.println("Sem casas decimais = " + numeroPontoFlutuante);  
        numeroPontoFlutuante = numeroInteiro * 5.2;  
        System.out.println("Com casas decimais = " + numeroPontoFlutuante);  
    }  
}
```

### Ordem de avaliação de operadores

A ordem de avaliação de operadores determina a ordem que os operadores devem ser considerados na avaliação de uma expressão. Por exemplo, em que ordem seria avaliada a seguinte expressão?

- $a / b * c + d$ ;

Qual seria a ordem correta de avaliação nas possibilidades listadas a seguir?

- $a / (b * (c + d))$ ; ou
- $(a / b * c) + d$ ; ou
- $a / (b * c + d)$ ;



A diferença na ordem correta de avaliação muda a forma de a expressão ser calculada e, portanto, o seu resultado. Na maioria das LPs, esse mecanismo funciona como na matemática tradicional, mas também depende do contexto em que está sendo utilizado. Em geral, a ordem de avaliação é feita da maior para a menor precedência ou importância em tantos níveis quanto forem precisos.

Para resolver as operações aritméticas, é necessário avaliar a ordem dos operadores, considerando:

- Regra de precedência.
- Regra de associatividade.
- Parênteses.
- Expressões condicionais.

As regras de precedência de operadores determinam a ordem na qual operadores de diferentes níveis de precedência são avaliados, conforme segue:

- operadores unários;
- **\*\*** ou **^** (se a linguagem o suporta);
- **\*** (multiplicação), **/** (divisão);
- **+** (soma), **-** (subtração);

Em uma expressão aritmética, os operadores de precedência mais alta serão resolvidos antes que aqueles de precedência mais baixa, conforme o quadro a seguir.

**Quadro 7 – Precedência de operadores**

Precedência	Java	C	Ada	Pascal
Mais alta	++, -- (pós-fixos)	++, -- (pós-fixos)	**, abs	*, /, div, mod
↕	++, -- (prefixos)	++, -- (prefixos)	*, /, mod	+, -, todos
	+, - (unário)	+, - (unário)	+, - (unário)	
	*, /, %	*, /, %	+, - (binário)	
Mais baixa	+, - (binário)	+, - (binário)		

A seguir mostraremos um exemplo de implementação na linguagem Java com a precedência dos operadores mais simples na seguinte ordem: **+** (adição), **-** (subtração), **\*** (multiplicação), **/** (divisão), **%** (resto de divisão).

```
public class PrecedenciaDeOperadores {
    public static void main(String args[]) {

        int a, b, c;
        a = 10;
        b = 5;
        c = 1;

        int y = 0;

        y = a + b;
        y = a - b;
        y = a * b;
        y = a / b;

        //módulo
        y = a % b;

        System.out.println( a * (b - c) );
        System.out.println( a * (b / a - b) );
        System.out.println( (a + b) / a - b );
    }
}
```

As regras de associatividade para avaliação de expressões determinam em qual ordem os operadores adjacentes da mesma precedência são avaliados, conforme o quadro a seguir (geralmente da esquerda para direita). Observe que os operadores soma (+) e subtração (-) têm o mesmo nível de precedência. Nesse caso, a avaliação é determinada por regras de associatividade.

**Quadro 8 – Regra de associatividade: 1 + 2 - 3 + 4**

LP	Regra de associatividade
C	Esquerda: ++ pós-fixado, -- pós-fixado, *, /, %, + binário, - binário Direita: ++ prefixado, -- prefixado, + unário, - unário
C++	Esquerda: *, /, %, + binário, - binário Direita: ++, --, + unário, - unário
Pascal	Esquerda: Todos
Fortran	Esquerda: *, /, +, - Direita: **
Ada	Esquerda: todos, exceto ** Direita: **

Os parênteses, quando colocados nas expressões, podem alterar as regras de precedência e de associatividade. As LPs que os permitem em expressões aritméticas poderiam dispensar todas as regras de precedência e simplesmente associar quaisquer operadores da esquerda para a direita ou vice-versa.

Por exemplo, na expressão a seguir, primeiramente é feita a soma dos operandos b e c, devido aos parênteses, depois o resultado é multiplicado pelo operando a.

- $a * (b + c) \rightarrow$  soma é feita primeiro

Na LP Pascal, os operadores lógicos têm maior precedência sobre os relacionais. Para alterar a ordem de precedência, deve-se usar parênteses. Em geral, as LPs utilizam a seguinte ordem de precedência: parênteses, operadores unários, exponenciação, divisão, adição e subtração e operadores relacionais.

As operações aritméticas podem ser realizadas em expressões condicionais, como o operador ternário ( $? :$ ) utilizado nas LPs Java, C e C++ da seguinte forma:

- `calculo = 10; numerador = 2;`
- `resultado = (numerador == 0) ? 0 : calculo / numerador`

Essa expressão é similar ao uso do comando `if / then / else`, conforme o seguinte pedaço de código:

```
int calculo = 10, numerador = 2;
if (numerador == 0) {
    resultado = 0;
}
else {
    resultado = calculo / numerador;
}
```

### 1.3.2 Conversões de tipo

A conversão de tipos de dados é uma operação muito comum no dia a dia do desenvolvedor de software. Em geral, componentes de software como formulários capturam os dados do usuário em formato de texto, e esses dados precisam ser convertidos para outros tipos. Dependendo da LP, ela fornece dois tipos de conversão: por estreitamento ou por alargamento.

- **Conversão por estreitamento:** transforma o valor de um tipo maior de grandeza ou precisão em um tipo menor. O valor de destino não pode armazenar todos os valores do tipo original. Por exemplo, converter `double` para `float`, uma vez que `double` possui uma faixa muito maior do que `float`.
- **Conversão por alargamento:** transforma o valor de um tipo menor de grandeza ou precisão em um tipo maior. Por exemplo, `float` para `double`, pois `float` é menor do que `double`.

## 1.3.3 Expressões relacionais e booleanas

As expressões relacionais têm como propósito comparar ou relacionar os valores de seus operandos, geralmente dois operandos e um operador relacional. O resultado dessa expressão é booleano: true (um) ou false (zero). Os operadores relacionais normalmente são sobrecarregados para uma variedade de tipos de dados e os seus símbolos variam bastante entre as LPs, conforme o quadro a seguir. Para a finalidade de comparação, os operadores, com exceção daqueles de igualdade e desigualdade, devem ser do mesmo tipo de dados.

**Quadro 9 – Expressões relacionais**

Operação	Java	Pascal	Ada
Igual	==	=	=
Diferente	!=	<>	/=
Maior que	>	>	>
Menor que	<	<	<
Maior que ou igual	>=	>=	>=
Menor que ou igual	<=	<=	<=

Os operadores relacionais verificam se o valor ou o resultado da expressão lógica à esquerda é igual ou diferente da direita, retornando um valor booleano (true/false), conforme exemplo a seguir:

```
public class ClassOperadoresRelacionais {
    public static void main(String args[]) {
        int valorEsquerda = 7;
        int valorDireita = 13;

        if(valorEsquerda == valorDireita){
            System.out.println("Retornou verdadeiro, então os valores são iguais");
        } else {
            System.out.println("Retornou falso então os valores são diferentes");
        }
    }
}
```

As expressões booleanas utilizam as operações da álgebra booleana. Os operandos são valores do tipo booleano ou semelhante e o resultado será um valor booleano ou similar, dependendo da LP, conforme veremos no quadro a seguir. Em geral, as LPs implementam os seguintes operadores lógicos:

- **Operadores binários:** E (and) para conjunção; OU (or) para disjunção.
- **Operador unário:** Não (not) ou ! para designar o não lógico.

**Quadro 10 – Expressões booleanas**

C	Ada	C#	Java	Fortran 90
&&	and	&&	&&	and
	or			or
!	not	!	!	not

A seguir um exemplo de implementação de expressão booleana:

```
public class ClassExpressaoBooleana {
    public static void main(String args[]) {
        int valorEsquerda = 1;
        int valorDireita = 2;

        if((valorEsquerda == (valorDireita - valorEsquerda)) && (valorDireita == (valorEsquerda +
valorEsquerda))) {
            System.out.println("As duas expressões comparadas são verdadeiras");
        }
    }
}
```

## 1.3.4 Instruções de atribuição

Os comandos de atribuição conferem determinado valor a uma variável que assume a forma geral (TUCKER; NOONAN, 2009):

- Alvo = <expressão>

Onde <expressão> pode ser o resultado de uma execução de uma função, o resultado de uma expressão, outra variável ou uma constante. As expressões são escritas com o uso de operadores aritméticos e lógicos encontrados em todas as LPs, assim como as chamadas eventuais a funções internas fornecidas pela linguagem.

Existe uma variedade de símbolos de operadores de atribuição, os dois formatos mais populares são os que seguem o estilo da LP Fortran (=) e aqueles que seguem o estilo da LP Algol (:=).

A semântica de uma atribuição é simples, na ausência de erros, a expressão é avaliada para um valor, que é então copiado para o destino alvo, ou seja, a maioria das linguagens imperativas usa semântica de cópia (TUCKER; NOONAN, 2009).

Em geral, a sintaxe da instrução de atribuição é:

<tipo de dado><variável que vai receber o valor> <operador de atribuição> <expressão/função/variável/constante>

Por exemplo, na linguagem C, que utiliza declaração seguida de atribuição na mesma linha e posteriormente altera essa atribuição:

- `float variavelFloatRecebeValor = 20.5;`
- `variavelFloatRecebeValor = 45.5;`

Por exemplo, nas linguagens Java, C++ e C#. Aqui está sendo feita apenas a atribuição, a declaração (com os tipos correspondentes) foi feita anteriormente:

- `variavelNumerica = 7;`
- `variavelTexto = "Luiz Roberto";`
- `variavelFaturamentoMedio = variavelFaturamentoTotal / qdeProdutosVendidos;`

Por exemplo, nas linguagens ADA, Pascal e Modula-2:

- `variavelBooleana := false;`

O símbolo do operador de atribuição pode variar em cada LP:

- Símbolo (=): Java, C++, Fortran, Basic, PL/I e C.
- Símbolo (:=): Pascal, ADA, Algol e Modula-2.

Existem diversas formas de implementar uma atribuição nas LPs. Algumas linguagens requerem a declaração anterior de uma variável (como Java) para que ela possa ser feita, enquanto outras não (como Python). Observe os exemplos a seguir (SEBESTA, 2011):

- **Atribuição simples:** é a mais tradicional: `A = 20.`
- **Atribuição múltipla:** `A=B=C=20.`
- **Atribuição como resultado de uma operação condicional:** `resultado = (var1 > var2) ? 13 : 27.`
- **Atribuição composta:** `soma += 13` (expressão equivalente a: `soma = soma + 13`).
- **Atribuição unária:** muito utilizada em contadores como incremento e decremento: `contador++;` `contador--;` (expressão equivalente a `contador = contador + 1;` ou `contador = contador - 1`).

## 2 ESTRUTURAS DE CONTROLE

Em programas de computador, as estruturas de controle permitem ao desenvolvedor controlar as condições nas quais os trechos do programa são executados. O profissional pode fazer uso dessas estruturas em qualquer tipo de programa para selecionar diferentes caminhos de execução, executar repetidamente determinadas instruções de código do programa ou fazer uma chamada de função, por meio de recursos como comandos sequenciais, condicionais e de repetição nas LPs.

As estruturas de controle são essenciais para qualquer LP, pois sem elas somente haveria uma maneira de execução do fluxo do programa: de cima para baixo, limitando muito a possibilidade de desenvolvimento de soluções computacionais.

Como vimos anteriormente, as linguagens imperativas simulam o funcionamento da máquina de Von Neumann, que é o fundamento da arquitetura dos computadores, e executam comandos de atribuição de valores a variáveis e estruturas que controlam o fluxo de execução do programa. A linguagem imperativa utiliza os seguintes comandos:

- **Atribuição:** possibilita que os valores de variáveis sejam atualizados.
- **Seleção:** permite a escolha de diferentes caminhos no fluxo de controle do programa.
- **Repetição:** proporciona que determinados trechos de código possam ser executados repetidamente.

Comandos são instruções do programa cujo objetivo é atualizar o estado das variáveis em memória e/ou controlar o fluxo de controle do programa. O suporte das LPs por meio de comandos pode ser classificado em primitivos ou compostos:

- **Primitivos:** são comandos que normalmente não podem ser subdivididos em outros. Por exemplo, um comando de atribuição para armazenar uma informação em uma variável: `int x = 5`.
- **Compostos:** são comandos que geralmente possuem um ou mais subcomandos, criando um escopo de ação. Por exemplo, um comando de repetição que contém pelo menos um subcomando e será repetido para alterar o estado da condição de parada, conforme o exemplo a seguir na LP C# (VAREJÃO, 2004).

```
int contador = 0;
while(contador < 3)
{
    Console.WriteLine("Contador: {0}", contador);
    contador += 1;
}
Console.WriteLine("FIM!");
```

### 2.1 Estruturas condicionais

A estrutura condicional utiliza uma expressão relacional para decidir qual caminho será executado pelo programa. Ela é formada pelos seguintes tipos de instruções de seleção:

- instrução de seleção unidirecional;
- instrução de seleção bidirecional;
- instrução de seleção múltipla.

#### Instrução de seleção unidirecional

Todas as LPs dão suporte à instrução de seleção unidirecional por meio da instrução que é chamada de IF, que utiliza uma expressão booleana para decidir se determinado trecho de código deverá ou não ser executado, conforme a seguinte figura.

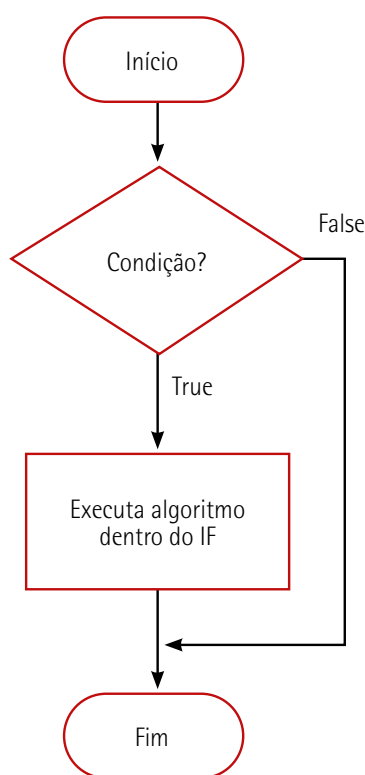


Figura 14 – Fluxograma seleção unidirecional

Esse trecho de código é executado somente quando a instrução avaliada for verdadeira, nesse caso, o fluxo seguirá um caminho ou não. Sua construção é a seguinte. O termo <expressão lógica> é uma expressão lógica que, ao ser avaliada, se o resultado for verdadeiro, executa a instrução contida em <instrução>, vide exemplo de implementação em Python:



- se <expressão lógica> então
- <instrução>
- fim se

A seguir um exemplo de implementação seleção unidirecional em Python:

```
numerolnicial=0  
resultadolnicial = 10  
if numerolnicial ==0:  
    numerolnicial = numerolnicial + 7  
    resultadolnicial =0
```

### Instrução de seleção bidirecional

A instrução de seleção bidirecional permite escolher entre dois ou mais caminhos, determinando o que será executado pelo programa quando o resultado da expressão lógica resultar em falso, conforme figura a seguir.

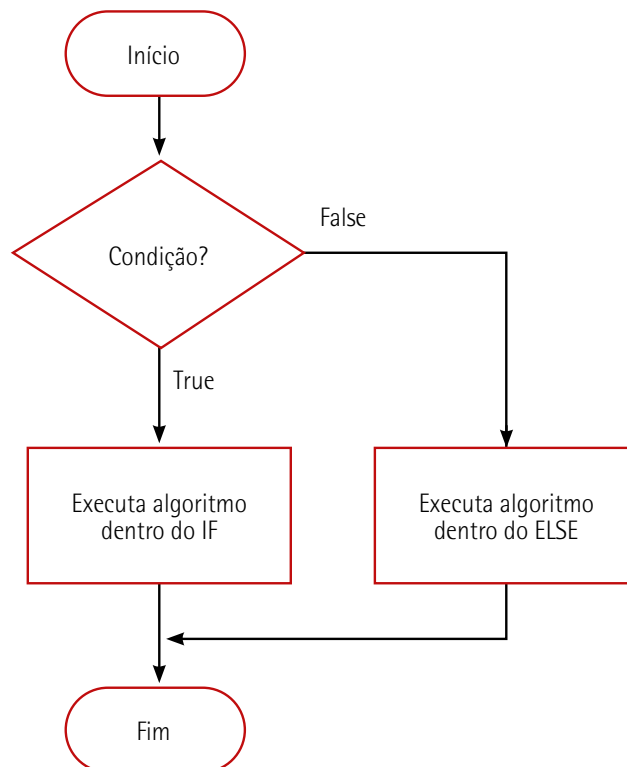


Figura 15 – Fluxograma seleção bidirecional

Ela permite ao desenvolvedor codificar o comportamento da instrução da seleção condicional tanto para o resultado verdadeiro quanto para o resultado falso. A expressão lógica se torna um seletor pelo qual escolhemos os caminhos. Essa sequência de instruções foi um grande passo na evolução da LP, inclusive podendo adicionar seletores aninhados, pois as instruções únicas acabavam por levar à dependência das instruções Goto.

A instrução bidirecional tem o seguinte formato:

- if (expressão lógica)
- <instrução>
- else
- <instrução>

Nas linguagens Java, C#, C e C++ utiliza-se a seleção bidirecional da seguinte forma:

- Avalia-se a expressão do lado do If. Caso ela seja verdadeira (true), o bloco de comando que forma o corpo do If é executado; se não, caso ela retorne Falso, o bloco de comando que pertence ao corpo do Else, caso ele exista, é que será executado.
- Observe que sempre será executado o trecho de código associado ao IF, ou o trecho de código associado ao Else, ambos nunca poderão ser rodados.

A seguir um exemplo de utilização da implementação da seleção bidirecional em Java:

```
public class SelecaoBidirecional {  
    public static void main(String args[]) {  
        int variavelA = 1;  
        int variavelB = 2;  
        if (variavelA == variavelB){  
            System.out.println("A variavelA contém um valor diferente da variavelB");  
        } else {  
            System.out.println("O valor da variavelA é igual ao valor da variavelB ");  
        }  
    }  
}
```

A instrução de seleção bidirecional foi introduzida na LP Algol 60 para executar uma única instrução ou uma instrução composta, sendo depois adotada por linguagens posteriores, conforme exemplo a seguir:

### Instrução simples

```
if (expressão lógica) then  
begin  
    <instrução 1>  
    ....  
    <instrução n>  
end
```

### Instrução composta

```
if (expressão lógica) then  
    <instrução>  
else  
    <instrução>
```

### Aninhamento de seletores

Denomina-se aninhamento de seletores quando da utilização de uma instrução de seleção em outra instrução de seleção. Por exemplo, aninhamento de seletores em C#.

Observe no exemplo a seguir a existência de um else solto, que, a princípio, não sabemos a qual if pertence. No C# temos uma regra semântica que diz que um else solto pertence ao if mais próximo, logo concluímos que ele pertence ao segundo if:

```
if (variavelA == 0)  
if (variavelB == 0)  
    variavelResultado = 0;  
else variavelResultado = 1;
```

Já no Pascal, para saber a qual if pertence um else solto, a regra semântica diz que ele pertence à instrução then mais próxima, o segundo then no exemplo.

```
if ... then  
if ... then  
...  
else ...
```

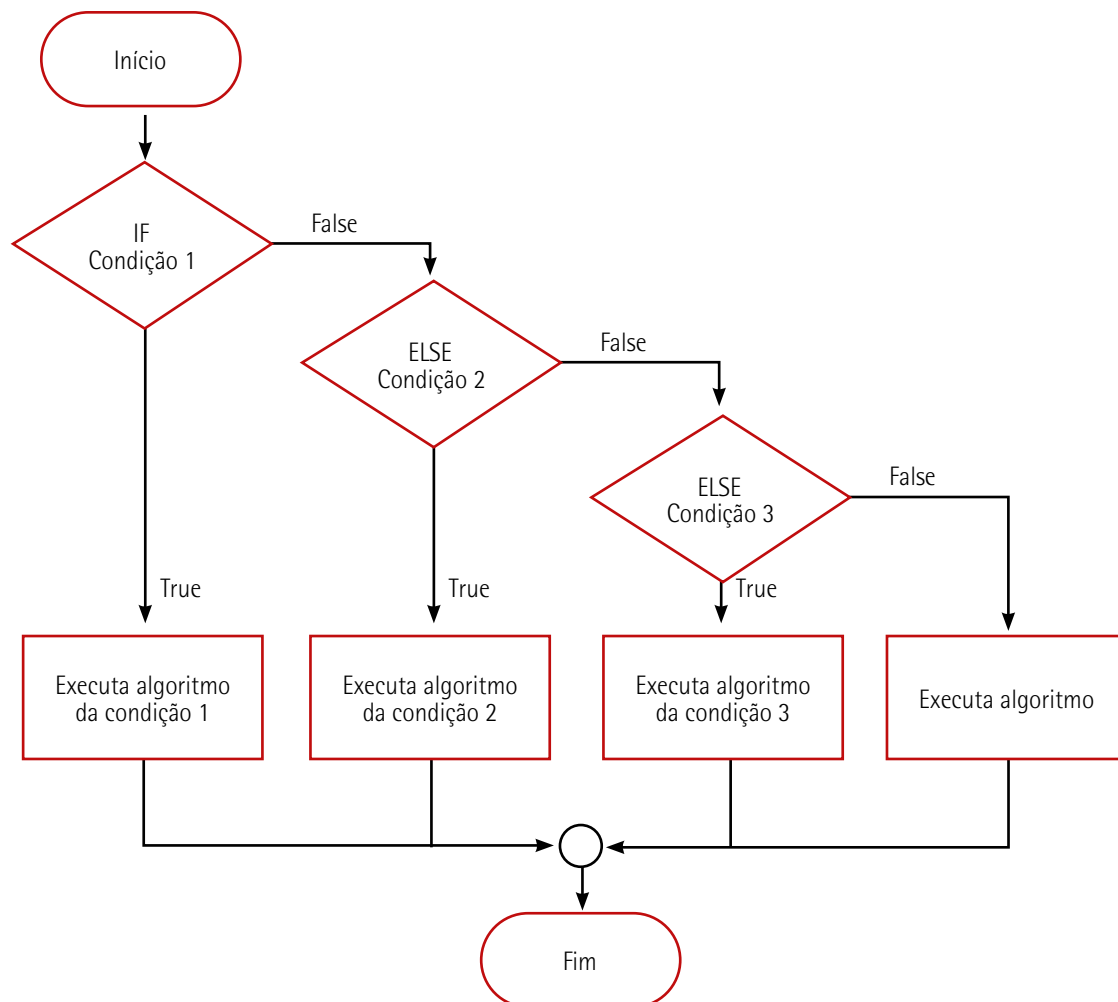


Figura 16 – Fluxograma aninhamento de seletores

Na linguagem Python o compilador sabe que um else solto pertence a um if devido à indentação ou aos espaços em branco (tabs), conforme exemplo a seguir. Se novas linhas de comandos forem adicionadas, o compilador saberá que elas pertencem ao bloco do if ou do else, e a razão de terem a mesma quantidade de espaços em branco ou indentação.

```

entradaTeclado = input("Informe a sua idade")
idade = int(entradaTeclado)
if idade <=1:
    print("Recém-nascido")
else:
    if idade < 13:
        print("Criança")
    else:
        if idade < 18:
            print("Adolescente")
        else:

```

```
if idade < 65:
    print ("Adulto")
else:
    if idade < 100:
        print ("Idoso")
    else:
        print ("Highlander")
```

Algumas linguagens oferecem uma semântica alternativa para criar instruções compostas por meio de uma forma sintática que utiliza chaves ({ }).

Observe o seguinte exemplo feito em Java, porém também serve para as LPs C#, C++, C e Perl, onde temos o código sem a utilização de chaves e depois o mesmo código com a utilização delas. Ambas as instruções funcionam e geram semelhante resultado, contudo o segundo código, usando chaves, é mais legível, uma vez que permite ao programador facilmente, por causa das chaves, identificar onde começa e termina a instrução.

Na sequência, um exemplo em Java sem a utilização de chaves ({ }):

```
if (variavelA == 0)
if (variavelB == 0)
    variavelResultado = 0;
else variavelResultado = 1;
```

Agora exibiremos o mesmo código utilizado anteriormente em Java, porém com a utilização de chaves ({ }):

```
if (variavelA == 0)
{
    if (variavelB == 0)
    {
        variavelResultado = 0;
    }
}
else
{
    variavelResultado = 1;
}
```

### Instrução de seleção múltipla

A instrução de seleção múltipla cuja origem pertence à LP Fortran permite, a partir de um conjunto de condições, determinar qual trecho de código, ou conjunto de instruções, será executado. Nesse tipo de seleção, apenas uma das condições será selecionada, isto é, produz uma escolha mutuamente exclusiva.

Na LP fortran temos o exemplo dos seletores múltiplos antigos, como o IF aritmético, por meio de um seletor tridirecional, ou seja, ele escolhe entre três caminhos de desvio baseando-se no valor de uma expressão aritmética, que pode ser:  $> 0$ ,  $= 0$ ,  $< 0$ .

Por exemplo, observe o código a seguir em que, de acordo com o número escolhido, o programa passa por três fluxos diferentes.

```
IF (expressão) 1, 2, 3
1 ...
...
GO TO 4
2 ...
...
GO TO 5
3 ...
...
4 ...
```

Apesar de ser uma alternativa interessante, note que essa implementação requer a utilização de vários comandos Goto, cujas labels de destino podem estar em qualquer lugar do código, dificultando a sua legibilidade.

### Comando de seleção múltipla Switch

O comando condicional é utilizado para selecionar caminhos alternativos durante a execução de um programa. Na maioria das linguagens, ele apresenta duas variações: os comandos if básico e case (ou switch) (TUCKER; NOONAN, 2009).

O switch é o construtor de seleção múltipla e um projeto relativamente primitivo e, em geral, apresenta a seguinte estrutura (SEBESTA, 2011):

```
switch (expressão)
{
    case expressão_constante_1: sentença_1;
    ...
    case constante_n: sentença_n;
    [default: sentença_n+1]
}
```

Onde a expressão de controle e as expressões constantes incluem tipos inteiros, como caracteres e tipos enumeração. As sentenças selecionáveis podem ser sequências de sentenças, sentenças compostas ou blocos de código. O segmento opcional default é usado para valores não representados da expressão de controle. Se o valor da expressão de controle não é representado e nenhum segmento padrão está presente, a construção não faz nada (SEBESTA, 2011).

Em muitas linguagens, como a LP C, é um requisito que cada case termine com uma declaração break explícita para evitar que o case caia accidentalmente no próximo, conforme pode ser observado na figura a seguir. Isso eventualmente pode provocar erros, porém até as LPs mais novas como Java e C# seguem esse padrão (TUCKER; NOONAN, 2009).

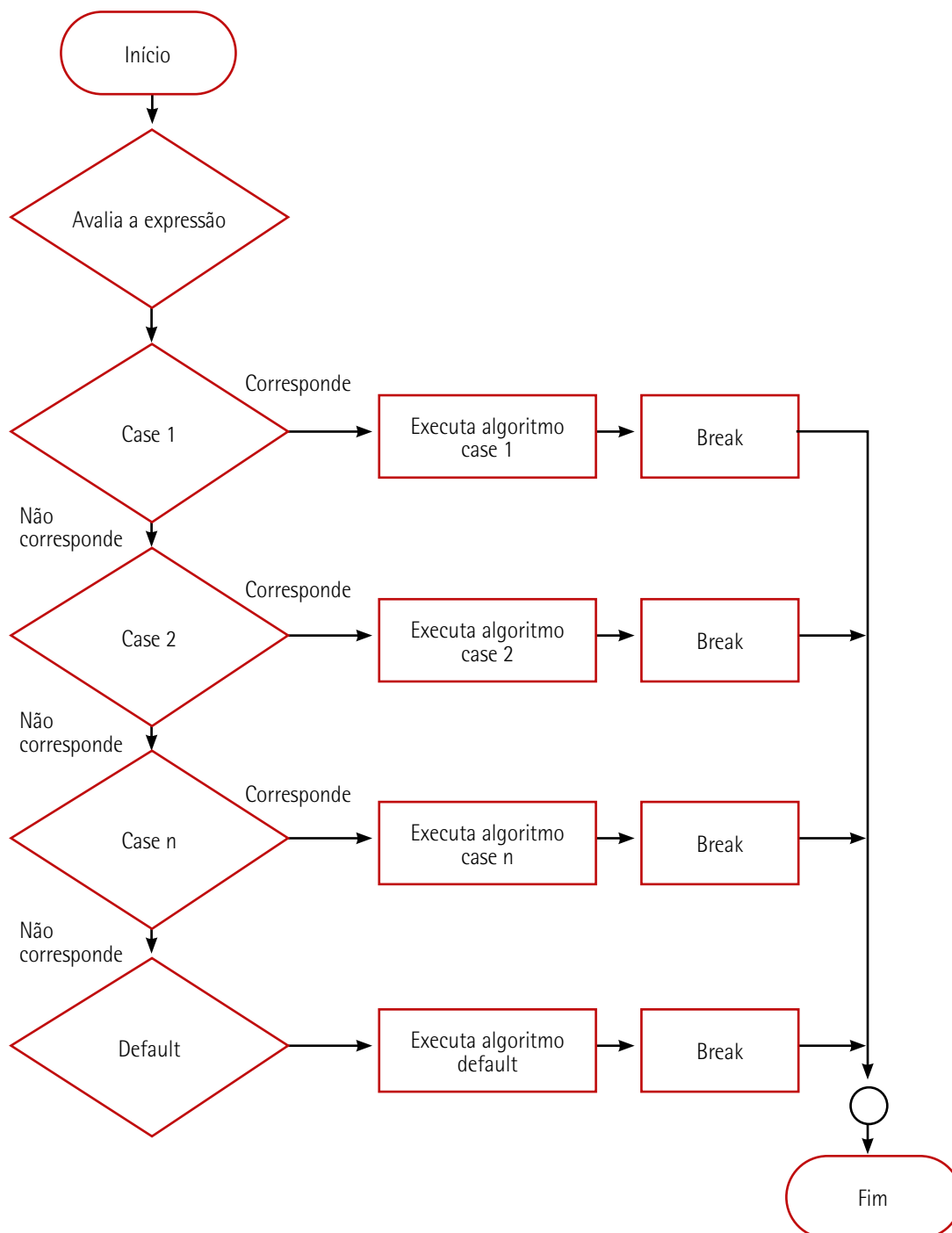


Figura 17 – Fluxograma de instrução switch

O comando de seleção múltipla switch é uma alternativa moderna que testa sucessivamente o valor de uma expressão em uma lista de constantes inteiras ou de caracteres, conforme pode ser visto na figura a seguir.

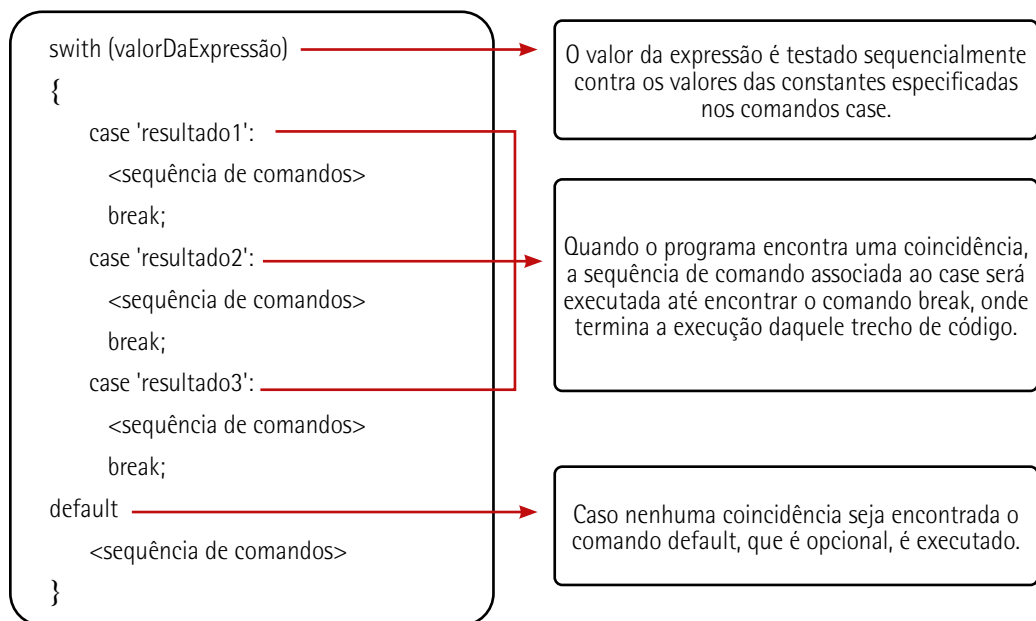


Figura 18 – Comando de seleção múltipla switch

O switch da linguagem C também é suportado pelas LPs C#, C++, Java e JavaScript.

Consta a seguir um exemplo de implementação do switch em C:

```
#include <stdio.h>
#include <conio.h>
int main (void )
{
    int numero;

    printf ("Digite um número de 1 a 3: ");
    scanf ("%d", &numero);

    switch ( numero )
    {
        case 1 :
            printf ("\n Você digitou o número 1");
            break;

        case 2 :
            printf ("\n Você digitou o número 2");
            break;
```



```
case 3 :  
    printf ("\n Você digitou o número 3");  
    break;  
  
    default :  
        printf ("\n Você digitou um número que não foi solicitado");  
    }  
  
    getch();  
    return 0;  
}
```

### 2.2 Estruturas de repetição

As estruturas de repetição, também conhecidas como estruturas iterativas ou de recursão, permitem aos desenvolvedores construir ciclos ou laços de repetição no fluxo de controle do programa, possibilitando que um bloco de instruções ou uma instrução seja executada zero, uma ou mais vezes mediante à satisfação de alguma condição lógica.

Em geral, as LPs implementam três tipos de instruções iterativas:

- laços controlados por contador;
- laços controlados logicamente;
- laços controlados pelo usuário.

#### Laços controlados por contador

Em uma estrutura de repetição que possui laço controlado por contador, os comandos são repetidos um número predeterminado de vezes.

A forma geral da sentença for é:

```
for (expressão_1; expressão_2; expressão_3)  
    corpo do laço/algoritmo/sentença
```

O corpo do laço pode ser formado por uma única sentença, uma sentença composta ou uma sentença nula. As expressões em uma sentença for são geralmente de atribuição. A expressão um é para inicialização, sendo avaliada uma única vez quando a execução da sentença for é inicializada.

A expressão dois é o controle do laço, que utiliza uma variável especial de laço, e contém uma expressão condicional para determinar quantas vezes o laço será executado e faz a avaliação antes da execução do corpo do laço. Por exemplo, na linguagem C um valor igual a zero significa falso e todos os

valores diferentes dele significam verdadeiro. Dessa forma, se o valor da expressão dois for igual a zero, a instrução for é terminada; caso contrário, as sentenças do corpo do laço são executadas conforme a figura a seguir.

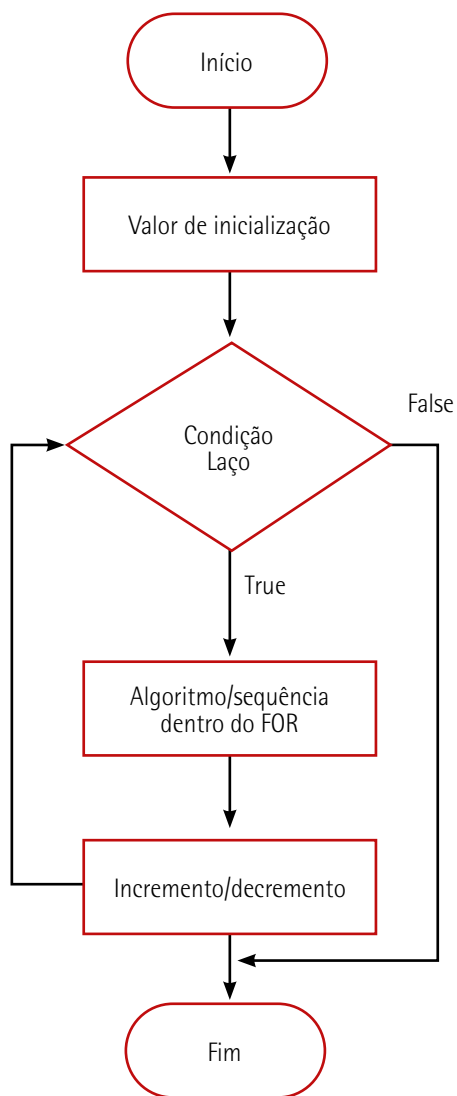


Figura 19 – Fluxograma de instrução FOR

O comando FOR, que faz parte de diversas LPs, é a principal estrutura de programação para repetir um conjunto de instruções mediante determinada condição. Ele possui meios de especificar os valores inicial e final da variável, e o tamanho do passo ou número de repetições, conforme veremos na sequência. Observe o seguinte trecho de código feito em Pascal:

- for <variavel> := <valor-inicial> to / downto <valor-final> do
- <instrução>

A seguir um exemplo de implementação da estrutura de repetição:

```
program exemploEstruturaRepeticao;  
uses crt;  
var i,resultado,valorA,valorB:integer;  
begin  
  for i:= 1 to 10 do  
  begin  
    writeln('Informe o primeiro número inteiro:');  
    readln(valorA);  
    writeln(' Informe o segundo número inteiro:');  
    readln(valorB);  
    resultado:=valorA+valorB;  
    writeln('O resultado da soma e:',resultado);  
  end;  
  readkey;  
end.
```

Em Java, C e C++, no comando FOR todas as variáveis envolvidas podem ser alteradas no corpo da instrução, a variável de inicialização é avaliada apenas uma vez, enquanto as variáveis condição e incremento são avaliadas em cada iteração, conforme figura a seguir.

- for(inicialização; condição; incremento)
- {
- <comandos;>
- }

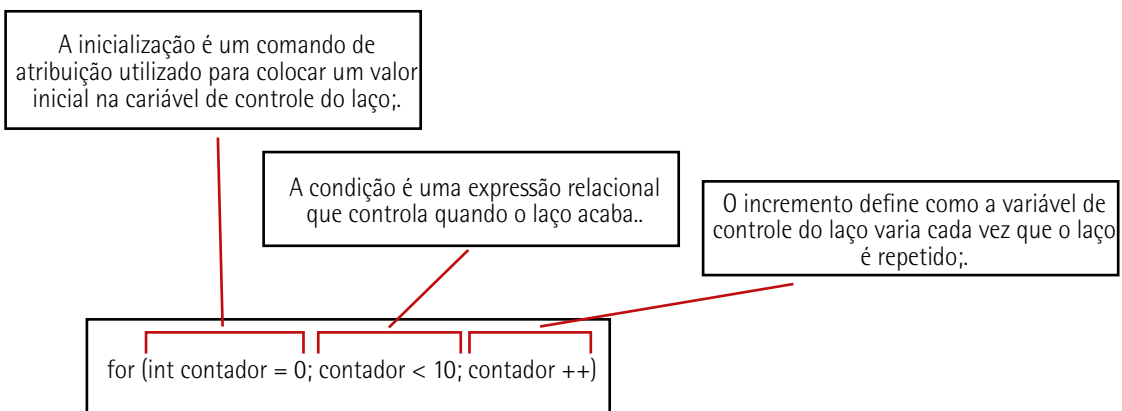


Figura 20 – Uso do comando FOR

### Laço controlado logicamente ou por condição lógica

Um laço controlado logicamente ou por condição lógica é utilizado quando não se sabe exatamente quantas vezes determinado trecho do código poderá ou não ser executado novamente, então o controle da repetição é feito por uma expressão booleana ou teste lógico e não por uma variável contador.

A quantidade de laços ou ciclos de iteração depende da avaliação de uma condição, que pode ocorrer no início, sendo também conhecida como pré-teste, ou ao final do laço ou ciclo de repetição, denominada de pós-teste.

Os laços lógicos como pré-teste e pós-teste têm as seguintes formas:

```
while (expressão_de_controle)
<corpo do laço ou expressão>
e
do
<corpo do laço ou expressão>
while (expressão_de_controle);
```

Os testes de condição realizados no pré-teste, conforme figura a seguir, avaliam a condição antes de a sequência ser executada e garantem que ela somente seja executada caso a condição seja verdadeira, isto é, se a condição for falsa, os comandos deixarão de ser executados; caso a condição nunca venha a ser falsa, ocorrerá o laço infinito.

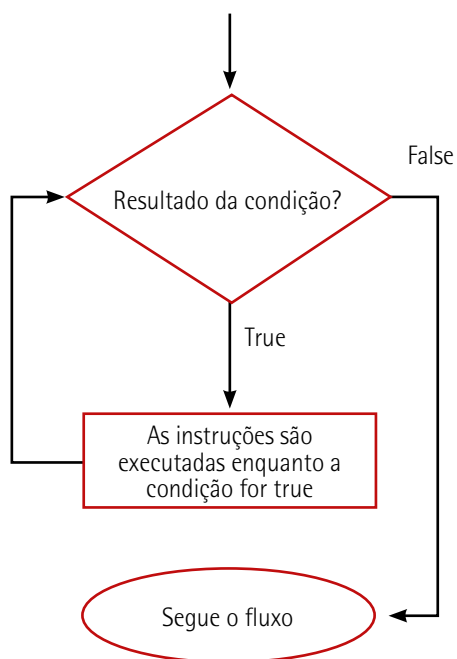


Figura 21 – Fluxograma laço controlado logicamente – pré-teste

Os testes de condição realizados no pós-teste (do), conforme figura a seguir, avaliam uma condição ao final do laço, garantindo que a sequência seja repetida pelo menos uma vez. Enquanto a condição for verdadeira, a sequência é repetida, ou seja, o corpo do laço será executado até a expressão ser avaliada como falsa.

Os laços pós-teste podem ser perigosos se utilizados com frequência, porque os programadores talvez esqueçam que ele é sempre executado ao menos uma vez, causando efeitos indesejados no fluxo do código (SEBESTA, 2011).

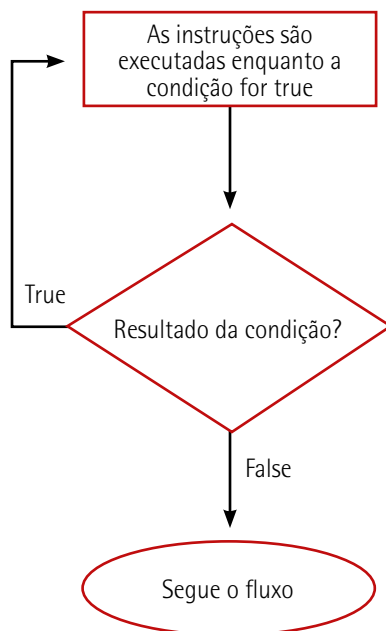


Figura 22 – Fluxograma do laço controlado logicamente – pós-teste



### Observação

O uso da avaliação da condição ao final do laço, conhecida como pós-teste, é recomendada quando se deseja que determinado trecho de código seja executado pelo menos uma vez.

Observe o seguinte exemplo em Pascal, que no caso dessa LP, possui instruções diferentes para o pré-teste e o pós-teste:

#### Avaliação no início ou pré-teste:

```
while (condição = true) do  
begin  
<instrução>  
end
```

**Avaliação ao final do laço ou pós-teste:**

```
repeat  
<instrução>  
until (condição = false)
```

### Exemplo de aplicação

---

Procure na internet quais LPs implementam o comando FOR e analise a diferença ou semelhança de sintaxe entre elas. Pesquise a utilização do comando FOR na LP Python, e reflita a respeito da flexibilidade desse comando nesta nova linguagem.

---

## 3 SUBPROGRAMAS

Subprograma é uma forma que as LPs utilizam para implementar o conceito de dividir para conquistar, com a intenção de facilitar a resolução de problemas computacionais, melhorar a compreensão do programa e viabilizar o reúso de código.

Essa técnica permite dividir o sistema em módulos nos quais os dados e procedimentos são encapsulados. Para realizar essa tarefa, utiliza-se o conceito de abstração, em que separa-se o que de fato importa no contexto do problema.

Os subprogramas são representados nas LPs por abstrações de procedimentos, eles realizam as ações necessárias quando chamados a partir do programa principal. Um subprograma fornece sua interface e identifica a sua definição por meio de seu cabeçalho, que é o seu protocolo (SEBESTA, 2011).

O cenário ideal para um subprograma é que ele contenha um código independente que possa ser compilado e testado separadamente.

### 3.1 Fundamentos

As LPs possuem um recurso importante conhecido como programação modular, que tem como função dividir um programa em submódulos ou subprogramas, conforme a figura a seguir. Subprogramas são agrupamentos ou blocos de instruções em uma pequena unidade de programa, que pode ser reutilizada, e está entre os recursos mais fundamentais em um projeto de LPs (SEBESTA, 2011).

Subprograma também pode ser definido como uma coleção de sentenças ou instruções para realizar algum algoritmo, tornando-se independente em relação ao restante do programa. Trata-se de uma técnica de projeto utilizada no desenvolvimento de software para separar o código em módulos, reduzindo a dependência de outros módulos.

Ele é um elemento fundamental em toda LP por se tratar de recursos essenciais para implementar o conceito de abstração em programação. Em algumas linguagens, os subprogramas são conhecidos como funções, procedimentos, sub-rotinas, ou métodos, e possuem várias características em comum, assim como apresentam diferenças importantes nas mais variadas linguagens (TUCKER; NOONAN, 2009).

O gerenciamento dinâmico de memória é altamente relacionado com o comportamento dos subprogramas, e caracteriza uma estrutura de memória fundamental, como a pilha, que nos ajuda a compreender melhor como a memória é organizada para implementar funções, especialmente as recursivas.

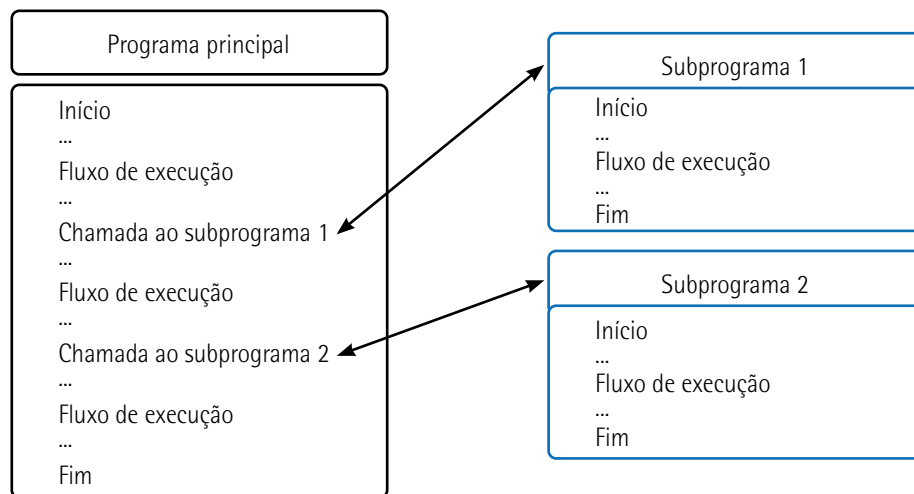


Figura 23 – Programação modular

Esse procedimento para resolução de problemas é conhecido na área de tecnologia da informação como dividir para conquistar, ou método dos refinamentos sucessivos, conforme a figura a seguir. Quando encontramos um problema computacional, podemos subdividi-lo em problemas menores, facilitando o seu entendimento, e quando esses problemas menores também não têm solução, até chegarmos no caso base, podemos dividi-los novamente na forma de módulos ou subprogramas.

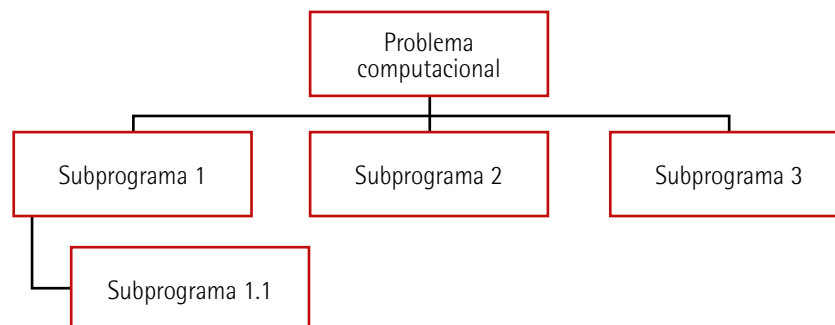


Figura 24 – Método dos refinamentos sucessivos

Em cada LP os subprogramas ou módulos são implementados de maneira diferentes, por exemplo, nas linguagens orientadas a objetos C#, C++ e Java os chamados módulos, procedimentos ou funções são

criados por meio das classes e dos métodos pertencentes a elas. Nas linguagens Cobol, Fortran, Algol 68 e Pascal, a modularização é implementada por meio dos procedimentos e das funções. Procedimentos executam um conjunto de instruções e não retornam valores para o programa principal, conforme exemplo de procedimento LP C a seguir:

```
#include <stdio.h>
int main()
{
    float altura, largura, areaTotal;
    printf("\nInforme a altura:");
    scanf("%f",&altura);
    printf("\nInforme a largura:");
    scanf("%f",&largura);
    areaTotal = altura * largura;
    printf("A área desse retângulo é: %.2f.\n", areaTotal);
    return 0;
}
```

As funções, conforme exibidas na figura a seguir, também são constituídas por uma sequência de instruções para executar uma tarefa específica, porém, no final, retornam um valor ou resultado ao programa principal. Por exemplo, a função `pow`, que calcula o valor de um número elevado a uma potência, recebe dois parâmetros e retorna um valor com o tipo de dado `double`.

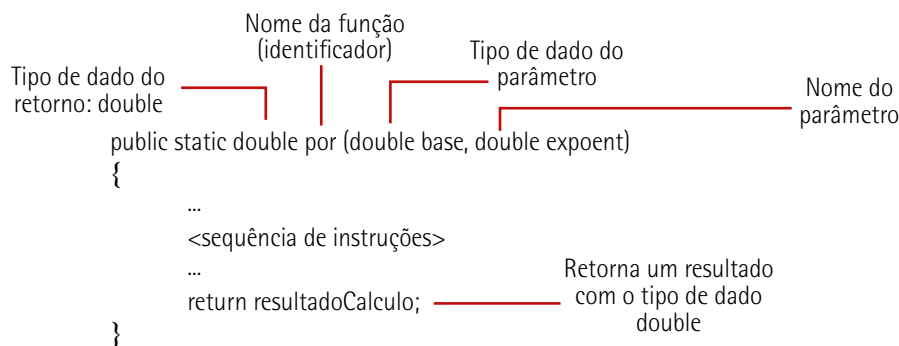


Figura 25 – Especificação de uma função

Com relação à função, podemos criar nossas próprias funções ou utilizar aquelas preexistentes, como funções matemáticas. Por exemplo, a biblioteca `Math` do Java possui diversas funções prontas para uso, conforme veremos a seguir:

- `Math.sqrt(8)` → Calcula a raiz quadrada de um número.
- `Math.pow(3, 2)` → Calcula a potência de um número: `pow(base, expoente)`, onde `base` e `expoente` são chamados de argumentos da função.



A seguir um exemplo de funções prontas em Java:

```
public class ExemploRetornoFuncao {  
    public static void main(String args[]) {  
        System.out.println("Calcula a raiz quadrada de um número: " + Math.sqrt(8));  
        System.out.println("Calcula a potência de um número: " + Math.pow(3, 2));  
    }  
}
```

A utilização de módulos ou subprogramas permite a melhor organização do código-fonte, pois cada módulo é responsável por resolver determinada função no software, facilitando a sustentação ou manutenção do código-fonte, o que permite encontrar mais rapidamente os erros do programa.



### Observação

A Máquina Analítica de Babbage, inventada nos anos 1840, que foi considerada o primeiro computador programável, tinha a capacidade de reutilizar coleções de cartões de instruções em diferentes lugares de um programa (SEBESTA, 2011).

Para facilitar o processo de desenvolvimento de software, utilizam-se os subprogramas para estruturar a solução dos problemas computacionais em pequenas partes, facilitando o entendimento e a implementação da solução.

Essa abordagem busca resolver problemas no desenvolvimento de software, aumentando a produtividade dos desenvolvedores para entregar os projetos dentro de prazos e custos aceitáveis. Ela também contribui para elevar a qualidade dos códigos implementados, facilitando a manutenção do código-fonte em programas de grande porte.

Conforme a construção de um programa se torna mais complexa, o recurso de subprogramas pode ser utilizado para dividir o programa em partes menores, que são mais fáceis de gerenciar. Todo software possui em geral um programa principal e um ou mais subprogramas, que também pode ser chamado de módulo, procedimento ou funções. Os benefícios da utilização de subprograma, em geral, são:

- Simplicidade para o desenvolvedor depurar e testar o programa.
- Descomplicação para projetar o programa, uma vez que cada subprograma executa uma tarefa específica.
- Criação de blocos de código ou módulos de programa independentes.

- Redução na quantidade de códigos de programa, pois um subprograma pode ser reutilizado em diferentes lugares do software.
- Facilitação da legibilidade do programa.
- Aumento da produtividade e qualidade no desenvolvimento de software.



### Lembrete

Para alcançar a qualidade no desenvolvimento de software criando programas de manutenção simples, seguros, com código legível e escalonáveis, é necessário que o desenvolvedor implemente arquiteturas utilizando o conceito de subprogramas, módulos, procedimentos, camadas e funções.

Por exemplo, considere que você foi contratado para desenvolver um site de compras. Existe a possibilidade de dividir esse programa em várias partes, cada uma com sua responsabilidade ou função específica. Seguindo uma certa ordem de como o processo acontece desde o início até a sua conclusão, teríamos os seguintes subprogramas ou módulos: cliente, categoria, produto, carrinho de compras, pedido, pagamento e entrega. A figura a seguir apresenta um extrato dessa arquitetura ou design de software na qual o subprograma ou módulo pagamento foi dividido em três. A ideia da separação é criar uma abstração em que cada subprograma agrupe atividades que tenham funções similares e nenhuma ou pouca dependência de outros subprogramas, organizando a forma como as tarefas são executadas, diminuindo a complexidade do software.

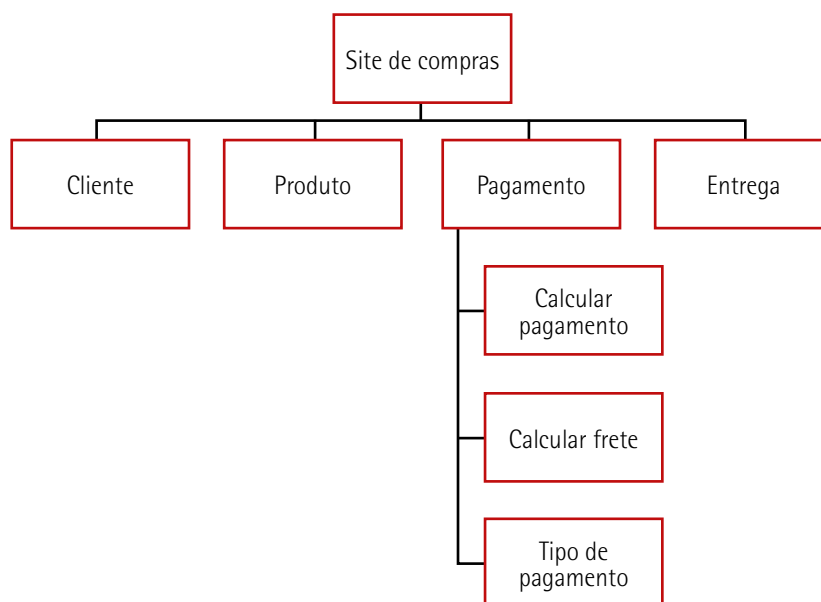


Figura 26 – Arquitetura em subprogramas



### Lembrete

Dentro do conceito de subprograma, uma pequena particularidade distingue uma função de um procedimento: a função recebe como argumento pelo menos um parâmetro e retorna um resultado, enquanto o procedimento não retorna valor.

### 3.2 Métodos de passagem de parâmetros

A passagem de parâmetros para os subprogramas é uma característica da programação modular. Essa técnica permite que os valores das variáveis do programa principal sejam enviados para os subprogramas. Nas LPs orientadas a objetos, como C# e Java, ela pode ser utilizada por meio dos métodos do programa. Nas LPs estruturadas, como Algol 68 e Pascal, essa técnica é realizada através da utilização de procedimento e funções. Como vimos anteriormente, é possível passar parâmetros ou informações para um subprograma. Para cada parâmetro, deve-se informar o tipo de dados e o seu nome ou identificador.

Existem três modos de utilização do recurso de passagem de parâmetros:

- **Modo de entrada:** utiliza-se para passar uma informação da unidade chamadora para o subprograma.
- **Modo de saída:** utiliza-se para enviar uma informação do subprograma para a unidade chamadora.
- **Modo entrada-saída:** utiliza-se tanto para enviar uma informação da unidade chamadora para o subprograma como para passar uma informação do subprograma para a unidade chamadora.

Existem três formas de passagem de parâmetros para um subprograma: passagem por valor, passagem por referência e passagem por resultado.

A **passagem por valor** é utilizada no modo de entrada, quando, para inicializar uma variável local do subprograma, é utilizado o parâmetro declarado no método. Ela somente existe durante a execução do subprograma, e todas as alterações realizadas nela não são levadas para a unidade do programa que chamou o método, geralmente o programa principal. Então, o dado usado no parâmetro do subprograma pode ser alterado, pois essas modificações não afetarão o programa principal.

Veja a seguir um exemplo em Java, no qual o programa principal, chamado main, executa o subprograma denominado de subtração. As variáveis locais desse subprograma, chamadas respectivamente de valorA e valorB, recebem os valores 4 e 2 e executam o algoritmo, retornando o resultado para o programa principal, que faz a impressão no console do computador.

```
public static void main(String args[]) {
    int valorA=4;
    int valorB=2;
    int resultado = subtracao(valorA, valorB);

    System.out.println("A subtração é = " + resultado);
}

static int subtracao (int valorA, int valorB)
{
    return valorA - valorB;
}
```

A **passagem por referência** é utilizada no modo de entrada-saída, no qual o endereço de memória da variável que será utilizada no parâmetro é enviada ao subprograma. Por esse motivo, todas as alterações realizadas nessa variável são executadas nesse endereço, refletindo-se na unidade chamadora. Então, o valor de uma variável referenciada no programa principal é afetado pela alteração realizada no parâmetro.

Consta a seguir um exemplo desenvolvido em C++. Observe que no programa principal a variável `numerolnicial` é inicializada com o valor 2. Esse valor é passado para a função `somar`, porém a passagem por referência, que possui um `&` antes da variável `valor1` no argumento da função, faz com que a variável seja referenciada através de um ponteiro. Caso ela sofra alterações na função, seu valor original também é alterado. Como dentro da função o valor da variável `valor1` é alterado para 1, então no programa principal o valor que será impresso no console do computador será 1.

```
#include <iostream>
int main(int argc, char** argv)
{
    int numerolnicial= 2;
    int resultado = somar(numerolnicial, 2);
    std::cout << numerolnicial;
    // IMPRIME: 1
}
int somar(int &valor1, int valor2)
{
    // Altera a variável valor1 para 1
    valor1 = 1;
    return valor1+ valor2;
}
```

A **passagem por resultado** é um modelo de implementação para parâmetros do modo de saída. Quando um parâmetro é passado por resultado, nenhum valor é transmitido para o subprograma. Esse parâmetro age como uma variável local, mas logo antes de o controle ser transmitido de volta para o chamador, seu valor é transmitido de volta para o parâmetro real dele por meio de uma variável que é utilizada no modo de saída, de onde a informação de uma variável local do subprograma é enviada para a unidade chamadora antes do término da execução do subprograma. Nem toda LP dá suporte a esse tipo de passagem.

Esse método apresenta as mesmas vantagens e desvantagens da passagem por valor, porém com algumas desvantagens adicionais. Caso os valores sejam retornados por cópia, em vez de serem retornados por caminhos de acesso, como ocorre na maioria das vezes, a passagem por resultado também requer, para sua correta utilização, um armazenamento extra e as operações de cópia necessárias pela passagem por valor, dificultando a implementação da passagem por resultado por meio da transmissão de um caminho de acesso onde geralmente é necessário utilizar a cópia de dados. O problema aqui está em garantir que o valor inicial do parâmetro real não seja usado no subprograma chamado (SEBESTA, 2011).

O seguinte programa em C#, que exemplifica o método passagem por resultado, utiliza o especificador `out`. No trecho de código do exemplo (SEBESTA, 2011), ao final da execução do método `RetornaValores`, será retornado o valor da variável que receber a atribuição por último. Se a variável `primeiraAtribuicao` receber a atribuição primeiro, o resultado será 3. Se a variável `segundaAtribuicao` receber a atribuição primeiro, o resultado será 2.

```
static void Main(string[] args)
{
    int valorInicial = 1;
    RetornaValores(out valorInicial, out valorInicial);
    Console.WriteLine(valorInicial);
    Console.ReadKey();
}

static void RetornaValores(out int primeiraAtribuicao, out int segundaAtribuicao)
{
    primeiraAtribuicao = 2;
    segundaAtribuicao = 3;
}
```

### 3.3 Sobrecarga de subprogramas

A sobrecarga de subprogramas permite criar subprogramas com o mesmo nome, mas com o número de parâmetros e/ou tipos de dados diferentes. As LPs C#, Python, Ada, C++, e Ruby permitem tanto a sobrecarga de subprogramas quanto de operadores. As funções podem ser definidas para construir significados adicionais para operadores (SEBESTA, 2011).

A capacidade de usar o mesmo nome pode melhorar tanto a legibilidade como a facilidade de escrita de uma linguagem, reduzindo a quantidade de nomes para subprogramas com a mesma semântica. Isto é, em Java o comando `println("isto é um teste")` pode imprimir uma variável de qualquer tipo, que é muito mais simples do que, por exemplo, na linguagem Modula, que utiliza o comando `WriteInt( )` para imprimir variáveis do tipo inteiro e `WriteReal()` para imprimir variáveis com valores monetários, obrigando o programador a decorar um número muito maior de comandos (TUCKER; NOONAN, 2009).

A LP Java, conforme exibiremos no exemplo de implementação de sobrecarga de subprogramas de Tucker e Noonan (2009), possibilita que o termo nome seja utilizado tanto para declarar a variável quanto para o método. Isso é possível pois as referências ao método têm parênteses como sufixo, permitindo que o programa diferencie a variável do método.

```
public class Aluno {  
    private String nome;  
    ...  
    public String nome ()  
    {  
        return nome;  
    }  
}
```

### 4 TIPOS ABSTRATOS DE DADOS

Nas mais diversas situações do dia a dia no desenvolvimento de software, o programador precisa lidar com um conjunto de informações que não pode simplesmente ser representado pelos tipos básicos de dados. Essas estruturas de dados podem ser compostas de um mesmo tipo de valores (homogêneo) ou de tipos de dados diferentes (heterogêneo), que podem ser dinâmicos ou estáticos. O programador também pode querer realizar operações diferentes sobre os dados.

A essas estruturas de dados e suas diversas operações damos o nome de tipos abstratos de dados (TAD), que podem ser definidos como uma estrutura que encapsula dados e os procedimentos apropriados para manipulá-los encapsulados.

Um TAD deve ser implementado em duas partes: a descrição da sua estrutura de dados e a definição de um conjunto de procedimentos, o que o torna a base primária da orientação a objeto que é o conceito de classe. Refere-se também a uma estrutura para a implementação de objetos em memória. A utilização do TAD requer a definição de uma interface para a sua manipulação, na qual o programador não precisa, por exemplo, conhecer os detalhes dos códigos definidos nos módulos em uma biblioteca, basta entender a interface de acesso e saber utilizá-la.

A importância do TAD é permitir o desenvolvimento de uma aplicação em módulos, ajudando a criar programas de maneira eficiente e organizada.

A seguir um exemplo de implementação de TAD na linguagem Java:

```
public class Pessoa {  
    private int rg;  
    private String nome;  
  
    public Pessoa(int rg, String nome) {  
        setrg(rg);  
        setnome(nome);  
    }  
}
```

```
public int getrg() {  
    return rg;  
}  
  
public void setrg(int rg) {  
    this.rg = rg;  
}  
  
public String getnome() {  
    return nome;  
}  
  
public void setnome(String nome) {  
    this.nome = nome;  
}  
  
public String getDadosPessoa() {  
    return nome + " - " + rg;  
}  
}
```

Conforme pôde ser observado, a utilização do TAD possibilita que o código possa ser reutilizado por diferentes programas. No exemplo, poderíamos criar vários objetos: Luiz, Maria ou Isabela, utilizando o mesmo TAD que foi elaborado.

### 4.1 Fundamentos de abstração

Abstração é a habilidade de se concentrar em alguns aspectos que interessam em determinado contexto, ignorando o que não tem importância, e escondendo os detalhes desnecessários de um código. Quando utilizamos um TAD, devemos nos concentrar nas suas operações, abstraindo a sua implementação. Trata-se de um conceito importante no desenvolvimento de software. Praticamente todas as LPs fornecem suporte à abstração de procedimentos através de módulos ou subprogramas.

Por exemplo, em C#, utilizar uma estrutura de dados do tipo vetor como argumento do método, conforme mostra o seguinte trecho de código onde o método classificar recebe como argumento um vetor do tipo de dados int nomeado de vetorExemplo:

- `public void classificar(int[] vetorExemplo)`

Para que uma LP dê suporte à abstração de dados, ela deve prover:

- Determinadas operações primitivas, que precisam fazer parte do processador da LP, não somente atribuições e comparações por igualdade ou desigualdade.
- Suporte para a reutilização do TAD, que uma vez implementado e testado, pode ser usado e reutilizado por diferentes programas.

- Permissão para alteração da lógica do programa sem necessariamente ter que reconstruir as estruturas de dados.
- Independência e portabilidade de código.
- Possibilidade de alteração da implementação do TAD sem necessariamente afetar a sua utilização por um ou mais sistemas, programas ou módulos. A implementação pode mudar, mas sua funcionalidade ou operações não.

### 4.2 Encapsulamento

O encapsulamento separa a especificação da implementação de determinado TAD. Apenas as operações deste TAD podem alterar esses dados, protegendo-os.

Esse mecanismo permite o uso do TAD sem conhecer nada sobre a sua implementação. Então, o TAD pode ter mais de uma implementação, trazendo como vantagens a organização do programa e a facilidade de manutenção, pois tudo o que estiver associado com uma estrutura de dados pode ser acessado pela mesma interface, sendo que a compilação pode ser realizada de maneira separada.

O uso do TAD traz mais segurança ao código, pois ao esconder as representações dos dados nenhum outro código pode acessar diretamente sua representação interna. Esse encapsulamento permite ao TAD ter sua implementação alterada sem afetar qualquer outro código do usuário que tenha acesso a ele.

### 4.3 Métodos de acesso a dados (public, private e protected)

A maioria das LPs que dão suporte ao POO implementa três tipos de visibilidade ou restrições de acessos para atributos e métodos: (+) Public (Público), (-) Private (Privado) e (#) Protected (Protegido).

- **Na visibilidade pública (+):** não há qualquer tipo de restrição de acesso aos membros de uma classe. Dessa forma, um método ou atributo declarado como público pode ser livremente utilizado por objetos de qualquer outra classe.
- **Na visibilidade privada (-):** que é a mais restritiva, o atributo declarado como privado é visível somente à própria classe onde foi criado. Então, se for informada uma propriedade privada chamada `modeloCarro` na Classe `Carro`, somente ela terá acesso a tal propriedade.
- **Na visibilidade protegida (#):** somente as classes que fazem parte da hierarquia é que têm acesso a essa propriedade. São classes que utilizam o mecanismo de herança.

### 4.4 Exemplos de abstração de dados em Java

Faremos agora um exemplo simples de abstração de dados em Java, implementaremos um pedido de venda com algumas propriedades comuns a esse tipo de operação: data da realização da venda e o



total vendido. Também utilizaremos dois métodos `CalculaImposto` para cada item vendido e o método `AddItemVendido`, que adiciona o item vendido e calcula o imposto dele.

Apenas para elucidar a abstração de maneira mais simples, observemos o código. Toda vez que um item for vendido, o imposto será calculado. Porém, o método `CalculaImposto` é público e pode ser acessado ou utilizado por qualquer outro método do programa.

Consta a seguir um exemplo de abstração de dados em Java (parte 1):

```
public class PedidoDeVenda
{
    private DateTime dataDaVenda;
    private double TotalVendido;
    public void CalculaImpostoSobreVenda() {...}
    public void AddItemVendido(ItemVendido itemVendido)
    {
        ItemVendido.Add(itemVendido);
        CalculaImpostoSobreVenda ();
    }
    ...
}
```

Para esse cenário, não faz sentido o método ser utilizado por outra parte do programa, pois tipicamente ele é usado no momento da venda. A fim de tornar o código mais correto de acordo com a ideia de abstração, bastaria torná-lo em método privado (`private`).

A seguir um exemplo de abstração de dados em Java (parte 2):

```
public class PedidoDeVenda
{
    private DateTime dataDaVenda;
    private double TotalVendido;
    private void CalculaImpostoSobreVenda() {...}
    public void AddItemVendido(ItemVendido itemVendido)
    {
        ItemVendido.Add(itemVendido);
        CalculaImpostoSobreVenda ();
    }
    ...
}
```



### Resumo

Nesta unidade, estudamos a relação entre algoritmos e LPs. Apresentamos os principais componentes das LPs: tipos de dados, expressões, instruções de atribuições e estruturas de controle, exibindo as estruturas sequenciais, condicionais de seleção e de repetição.

Exibimos também a utilização de subprogramas no desenvolvimento de software, discutindo tópicos como passagem de parâmetros por valor, por referência e por resultado, além da sobrecarga de subprogramas.

Na sequência, vimos os componentes essenciais que fazem parte das LPs mostrando alguns dos seus principais paradigmas existentes. Tivemos o intuito de expor as características de cada paradigma, bem como de demonstrar as LPs que o utilizam.

Por fim, explicitamos os tipos abstratos de dados (TAD), os conceitos de encapsulamento, e os métodos de acesso a dados, como público, privado e protegido.



## Exercícios

**Questão 1.** Considere o exemplo de código-fonte indicado a seguir, que mostra a utilização de arrays na LP Java.

```
public class Main {  
    public static void main(String[] args) {  
        String texto = "Oi, teste de texto.";  
        int[] posicoes = new int[4];  
        int pos;  
        posicoes[0]=5;  
        posicoes[1]=1;  
        posicoes[2]=10;  
        posicoes[3]=18;  
        System.out.print("Caracteres:");  
        for(pos=0;pos<posicoes.length;pos++){  
            System.out.print(texto.charAt(posicoes[pos]));  
        }  
        System.out.println("");  
    }  
}
```

Devido à simplicidade desse programa, não é necessária a utilização de nenhuma IDE para testar a sua funcionalidade. Podemos simplesmente salvar o texto com o código-fonte em um arquivo chamado "Main.java". Depois, a compilação pode ser feita diretamente com o compilador da linguagem Java (chamado "javac"), executado em um interpretador (ou console) de comandos:

```
javac Main.java
```

Finalmente, o programa poderá ser executado com o seguinte comando (executado no mesmo diretório da compilação):

```
java Main
```

Assinale a alternativa que corresponde à saída desse programa após a sua execução em um interpretador ou em um console de comandos.

A) Caracteres: Oi, teste de texto.

B) Caracteres: eid.

C) Oi, teste de texto.

D) eid.

E) Oi.

Resposta correta: alternativa B.

### Análise das alternativas

A) Alternativa incorreta.

Justificativa: apenas os caracteres indicados pelo array "posicoes" são mostrados na tela, e não todo o conteúdo da variável "texto".

B) Alternativa correta.

Justificativa: os caracteres mostrados na tela correspondem às posições: 5, 1, 10 e 18. Devemos observar que esses valores são mostrados de acordo com a ordem no array "posicoes", e não em ordem alfabética.

C) Alternativa incorreta:

Justificativa: apenas os caracteres indicados pelo array "posicoes" são mostrados na tela, e não todo o conteúdo da variável "texto". O texto mostrado na tela deve começar com "Caracteres:", como indicado pelo comando `System.out.print("Caracteres:");`. Observe que o comando utilizado é "print", e não "println": por isso, todos os caracteres são mostrados na mesma linha.

D e E) Alternativas incorretas.

Justificativa: o texto mostrado na tela deve começar com "Caracteres:", como indicado pelo comando `System.out.print("Caracteres:");`. Observe que o comando utilizado é "print", e não "println": por isso, todos os caracteres são mostrados na mesma linha.

**Questão 2.** Suponha o seguinte código-fonte, escrito em linguagem de montagem (Assembly), para a plataforma Intel/AMD x86-64. Esse programa foi desenvolvido em uma máquina Intel x86-64 executando o sistema operacional GNU/Linux (também na versão x86-64), conforme mostrado a seguir.

```
global _start

        section .text
_start:  mov    rax, 1          ; chamada de sistema write
        mov    rdi, 1          ; stdout
        mov    rsi, texto      ; end. da cadeia a ser mostrada na tela
        mov    rdx, 10         ; numero de bytes a serem escritos
        syscall                ; instrucao para invocar a
                                ; chamada de sistema write
        mov    rax, 60         ; chamada de sistema exit
        xor    rdi, rdi        ; codigo de saida 0
        syscall                ; instrucao para invocar a
                                ; chamada de sistema exit

        section .data
texto:  db      "Oi mundo!", 10
```

O código-fonte do programa foi gravado em um arquivo chamado "oi.asm". O assembler utilizado foi o NASM, e o vinculador "GNU ld", conforme exibido a seguir.

```
nasm -felf64 oi.asm
ld oi.o
```

Como resultado da execução desses comandos, foi gerado um arquivo chamado "a.out", no formato ELF executável de 64 bits (plataforma Intel/AMD x86-64, para o sistema operacional GNU/Linux). Quando esse arquivo é executado em um terminal ou no console dessa mesma máquina, obtém-se a seguinte saída:

Oi mundo!

Adaptada de: <https://bit.ly/3KQH0j4>. Disponível em: 21 jan. 2022.

Com base nessas informações, e supondo a execução direta do programa na máquina (ou seja, sem o auxílio de máquinas virtuais, emulação ou qualquer tipo de virtualização), avalie as afirmativas.

I – O mesmo binário deste programa ("a.out") pode ser diretamente executado com sucesso em diferentes plataformas executando em microprocessadores variados. Por exemplo, é possível executar o mesmo binário em uma máquina com um microprocessador ARM sobre o GNU/Linux ou em uma máquina x86 com qualquer outro sistema operacional.

II – Desde que o microprocessador seja o mesmo, o binário deste programa ("a.out") pode ser executado em máquinas rodando diferentes sistemas operacionais, como as diversas versões do GNU/Linux, os variados sistemas operacionais \*BSD ou o sistema operacional MS-Windows.

III – O código-fonte desse programa pode ser utilizado sem alterações em outros assemblers, como o "GNU AS", desde que seja executado em uma mesma plataforma.

IV – Pode ser necessário alterar o código-fonte desse programa para que seja possível a sua execução em diferentes plataformas, mesmo que seja utilizado o mesmo *assembler* (no caso, o NASM).

É correto o que se afirma apenas em:

A) I.

B) II.

C) III e IV.

D) IV.

E) I e III.

Resposta correta: alternativa D.

### Análise das afirmativas

I – Afirmativa incorreta.

Justificativa: a linguagem assembly depende da plataforma específica, inclusive da linguagem de máquina própria de uma CPU. Dessa forma, microprocessadores diferentes podem ser totalmente incompatíveis, o que faz com que um programa que rode em uma plataforma não possa ser executado em outra.

II – Afirmativa incorreta.

Justificativa: mesmo que o microprocessador seja o mesmo, sistemas operacionais diferentes podem ter formatos de binários ou chamadas de sistema diversos. Dessa forma, esse programa é incompatível com muitos sistemas operacionais.

III – Afirmativa incorreta.

Justificativa: outros *assemblers* podem ter uma sintaxe diferente do NASM, como é o caso do GNU AS. Dessa forma, programas escritos para um dado assembler podem ser incompatíveis com outros assemblers diferentes, mesmo que executando em uma mesma plataforma.

