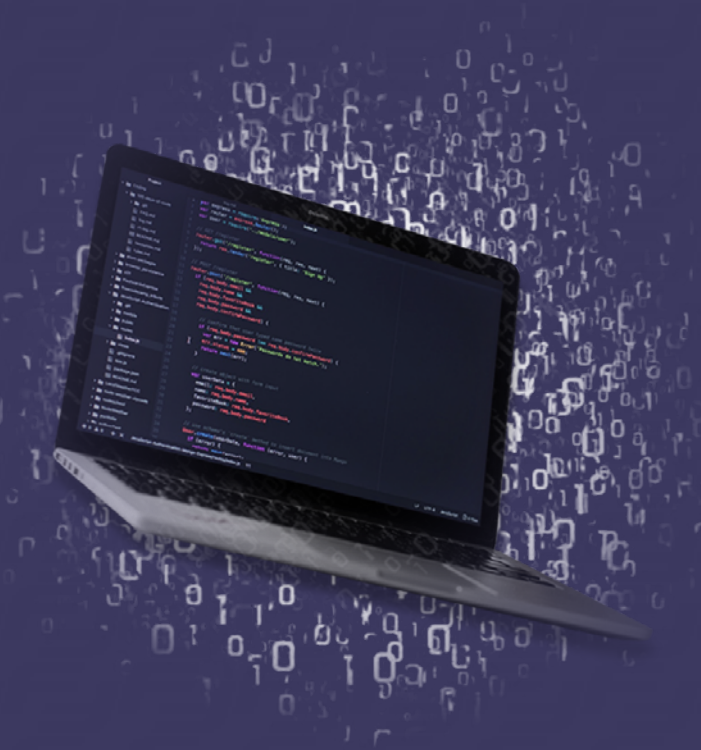
PARADIGMAS DE LINGUAGENS DE PROGRAMAÇÃO

MSC Rafael De Moura Moreira



FAM ONLINE

e-Book 1

Sumário

INTRODUÇÃO

Olá, estudante! Seja bem-vindo ao curso de Paradigmas de Linguagem de Programação!

Iniciaremos nossos estudos apresentando alguns conceitos básicos de linguagens de programação e diferentes maneiras de classificá-las. Também iremos estudar a história da evolução de algumas das principais linguagens de programação desde a época da linguagem de máquina até os dias atuais, e por fim verificaremos o que é um paradigma de programação e como eles podem nos tornar programadores mais versáteis.

Vamos lá? Bons estudos!

LINGUAGENS DE PROGRAMAÇÃO

Motivação

Neste tópico, você irá conhecer um pouco sobre linguagens de programação: o que elas são, quais as principais características que podemos observar nelas e como classificá-las de diferentes maneiras. Esses conceitos serão a base para o restante da disciplina.

O estudo dos conceitos de linguagem de programação é fundamental no pleno desenvolvimento de um profissional de programação. Muitos cursos "rápidos", antes vendidos em revistas e CDs em bancas de jornal e atualmente na internet, dão a entender que aprender a programar equivale a aprender uma linguagem de programação. Essa visão não corresponde à realidade de um programador.

Na prática, programadores frequentemente precisam alternar entre diferentes linguagens, trabalhando em projetos com mais de uma linguagem simultaneamente, ou precisam tomar decisões importantes sobre qual ou quais linguagens utilizar em um novo projeto, e essas decisões podem ter um grande impacto no desempenho computacional (ou seja, se a aplicação vai ser rápida ou lenta), no trabalho dos desenvolvedores (que podem ter

facilidade ou muita dificuldade para desenvolver o projeto na linguagem escolhida) ou até mesmo em aspectos de segurança do projeto.

Além de escolher entre diferentes linguagens no início de um projeto, muitas vezes por necessidades profissionais, é necessário que um programador aprenda rapidamente uma nova linguagem. Isso pode parecer assustador para alguém começando a programar, mas tendo uma boa base conceitual de linguagens de programação, após um estudo breve e superficial sobre as características principais da nova linguagem, o programador experiente já sabe quais estruturas e comportamentos esperar e saberá exatamente o que procurar na documentação, conseguindo iniciar o desenvolvimento na nova linguagem rapidamente.

Assim como um poliglota aprende línguas novas com mais facilidade do que alguém que ainda está aprendendo sua primeira língua estrangeira por conseguir identificar diversos elementos em comum com outras línguas que já estudou, espera-se que um programador consiga facilmente estabelecer paralelos entre linguagens já conhecidas e novas linguagens e se adaptar rapidamente.

Por fim, conhecendo os conceitos teóricos por trás das linguagens que utilizamos em nosso dia a dia, é mais fácil identificar os objetivos de seus

criadores, seus pontos fortes e seus pontos fracos. Isso nos permite fazer um uso mais eficiente, seguro e até mesmo correto da linguagem escolhida, utilizando os recursos disponíveis com sabedoria e evitando cair em “armadilhas” ligadas ao projeto da linguagem.

Domínios de programação

Um fator que impacta diretamente na escolha da linguagem disponível será o domínio correspondente ao nosso projeto, isto é, em qual campo da computação nosso programa irá atuar. Diferentes áreas possuem diferentes objetivos.

Podemos citar alguns domínios diferentes, como a computação científica, a programação web, a programação embarcada, a programação para dispositivos móveis (celulares e tablets), entre outros.

A computação científica, por exemplo, serve para auxiliar cientistas a realizar cálculos, fazer previsões e extrair informações a partir de grandes volumes de dados. A alta precisão matemática, uma boa variedade de funcionalidades para automatizar o uso de fórmulas científicas comuns, ferramentas estatísticas, recursos para visualização de dados e uma performance rápida são fatores bastante importantes na computação científica. A linguagem FORTRAN já foi imbatível nesse domínio, que

atualmente pende bastante para linguagens como R e Python.

Quando partimos para a programação web, notamos uma mudança dramática nas prioridades. O foco passa a ser outro: as páginas precisam ser responsivas (ou seja, se adaptar a diferentes formatos de tela, como tamanhos variados de celular e telas de computador), interativas, intuitivas e fáceis de serem usadas.

Atualmente, o próprio domínio web frequentemente é subdividido em diferentes categorias no mercado. É comum que um conjunto de linguagens seja utilizado para fazer o *front-end*, a parte visível para o usuário – normalmente, as páginas em si. As linguagens utilizadas focam no posicionamento de elementos gráficos e no comportamento deles diante da interação dos usuários. É comum o uso de linguagens como o HTML, feito especificamente para criar páginas dinâmicas, o JavaScript, capaz de executar lógica diretamente no navegador, e do CSS, que dita o comportamento de elementos gráficos.

Já o *back-end* acontece longe do usuário, muitas vezes sendo executado em um servidor – um computador rodando uma aplicação que irá fornecer os recursos solicitados pelo cliente, a máquina do usuário. No back-end o importante não é mais

usabilidade ou interatividade, mas a facilidade com que os desenvolvedores poderão implementar novos recursos e expandir o serviço, bem como a velocidade com a qual o cliente conseguirá se comunicar com o servidor, acessar o banco de dados e obter ou enviar novas informações. Aqui existe uma grande variedade de linguagens populares, como Java, Python e Ruby, geralmente equipadas com várias ferramentas otimizadas para esse tipo de tarefa e frequentemente interagindo com bancos de dados através de linguagens como SQL, feita especificamente para buscar dados.

Ao mudarmos da programação web para a programação embarcada, nós mudamos completamente de mundo. Chamamos de sistemas embarcados aqueles equipamentos ou dispositivos que possuem computadores embutidos realizando as tarefas do dispositivo. Isso vai desde eletrodomésticos "inteligentes" até sistemas robóticos industriais, equipamento hospitalar, máquinas de pagamento eletrônico e sistemas de controle automotivo ou de voo.

Na programação embarcada é comum que haja restrições sérias de recursos – muitas vezes os chips envolvidos são lentos, há pouca memória disponível ou até mesmo o equipamento usa bateria, o que significa que atividade computacional intensa pode fazer com que o dispositivo acabe desligando antes

da hora por excesso de consumo de eletricidade. Por conta disso, sistemas embarcados devem ser extremamente eficientes e econômicos, fazendo o máximo possível com a menor quantidade possível de instruções complexas e o menor uso possível de memória e outros recursos.

Além disso, por controlarem muitas coisas importantes, eles precisam ser mais robustos e reagir bem a falhas de operação ou erros. As linguagens mais populares aqui costumam permitir um controle muito próximo da máquina e/ou incluem mecanismos sofisticados de segurança. No passado essa área era dominada quase exclusivamente por linguagens como C e Assembly, mas hoje as portas têm se aberto para linguagens como Rust e Go.

Uma lição importante que podemos tirar da existência de diferentes domínios de programação é que não existe uma linguagem superior a todas as outras, e sim ferramentas diferentes para tarefas diferentes. Uma linguagem excelente para fazer sites bonitos e interativos pode ser uma péssima linguagem para desenvolver o sistema de controle de temperatura de um forno industrial.

Critérios de avaliação de linguagens

Existe uma grande variedade de critérios que poderiam ser utilizados para avaliar linguagens de programação. Diferentes programadores podem discutir sobre a importância relativa de cada um desses critérios. Porém, de maneira geral, conhecer esses critérios nos ajuda a comparar diferentes linguagens e identificar com mais facilidade os benefícios que cada linguagem poderia trazer para o nosso projeto. Discutiremos brevemente três critérios: legibilidade, facilidade de escrita e confiabilidade.

A **legibilidade** está relacionada a quão fácil é para você ler um código pronto e compreender o que ele faz. Isso pode não parecer importante para quem está começando agora, afinal, nós escrevemos código para o computador, e não para seres humanos, certo? Não exatamente.

É comum que diversas pessoas trabalhem em um mesmo projeto. Muitas vezes haverá até mesmo rotatividade de pessoas: colegas de equipe podem trocar de projeto ou de empresa, novos programadores podem ser contratados, pessoas podem sair de férias e delegar parte de suas tarefas para outras pessoas, e assim sucessivamente. Além disso, é comum que uma mesma pessoa trabalhe com muitas coisas diferentes, e passe meses distante de um código que produziu até que surja

a necessidade de retornar até ele para corrigir um erro ou acrescentar novas funcionalidades.

Quando o código é legível, todas as situações descritas acima se tornam mais fáceis. Uma pessoa nova lendo um código bem legível terá facilidade em compreender o que ele faz, e consequentemente poderá rapidamente identificar possíveis erros ou acrescentar novas funcionalidades. Código pouco legível pode dar muito trabalho para ser compreendido, o que implica em mais tempo para que um programador consiga realizar seu trabalho – portanto, mais custo para o projeto – e um risco maior de que erros sejam cometidos e passem despercebidos.

Parte da legibilidade depende da adoção de boas práticas por parte do programador, mas a linguagem também possui grande impacto. O conjunto de regras das linguagens (sintaxe) pode ser simples e enxuto, levando os programadores a desenvolverem linhas de código curtas e intuitivas, ou pode ser extremamente complexa, resultando em código grande, confuso e possivelmente ambíguo. Os comandos e até mesmo os tipos da linguagem podem ser intuitivos de utilizar e muito próximos de palavras do vocabulário humano, tornando o código mais próximo de um texto escrito para humanos, ou podem ser mais carregados em símbolos, nomes complicados e números no lugar de

palavras, exigindo um esforço muito maior para a compreensão de seu significado.

Tabela 1: O programa “Olá mundo” em diferentes linguagens de programação.

Linguagem	Programa “Olá mundo!”

C	`#include <stdio.h>`
	
	`int main()`
	`{`
	`  printf("Ola mundo!");`
	`  return 0;`
	`}`

C++	`#include<iostream>`
	
	`int main()`
	`{`
	`  std::cout<<"Ola mundo!";`
	`  return 0;`
	`}`

Java	`public class Main`
	`{`
	`  public static void main(String[] args) {`
	`  System.out.println("Ola mundo!");`
	`  }`
	`}`

Python	`print ('Ola mundo!')`

Fonte: Elaborada pelo autor. (2021).

A **facilidade de escrita** é bastante relacionada com a legibilidade, mas vai um pouco além. Ela está relacionada a quanto esforço o programador irá fazer para traduzir suas ideias em código.

Um dos conceitos que afeta tanto a legibilidade quanto a escrita da linguagem é a chamada expressividade. Uma linguagem mais expressiva consegue realizar mais utilizando códigos mais enxutos. O uso de operadores que abreviam ou resumem a operação de diversos outros operadores e funcionalidades padrão da linguagem com nomes intuitivos que substituam grandes blocos de código contribui com a expressividade da linguagem.

Outra ideia que irá impactar a legibilidade e definitivamente impactar muito na facilidade de escrita é quão complexo é o conjunto de regras e instruções da linguagem. Certas linguagens, ao tentar oferecer uma boa variedade de ferramentas para o programador, acabam criando muitas formas diferentes de se realizar a mesma tarefa, com diferenças sutis. Isso pode comprometer até o aprendizado da própria linguagem, fazendo com que programadores experientes nunca aprendam de verdade como utilizar todos os recursos disponíveis. Outros programadores podem se confundir ao não saber diferenciar estruturas parecidas.

Além da legibilidade e da facilidade de escrita, é importante analisar uma linguagem do ponto de vista dos recursos que ela oferece para melhorar a **confiabilidade** dos programas escritos nela.

A confiabilidade diz respeito à garantia de que um programa se comporta de acordo com a sua especificação. Quão certos nós estamos de que não há casos onde ele irá se comportar de maneira imprevista ou errada? Uma linguagem de programação pode oferecer recursos para auxiliar o programador na produção de um programa confiável e "bem-comportado".

Uma das ferramentas que podem prevenir erros e falhas de segurança é a verificação de tipos. Se um trecho de código foi projetado para trabalhar com números inteiros, por exemplo, a linguagem não deve permitir que um número de ponto flutuante (uma forma de representação de número com casas decimais) seja passado para esse código.

Outro exemplo é a ideia de tratamento de exceções. O tratamento de exceções é uma espécie de tolerância a erros que podem ocorrer durante a execução do programa. O programador tem a possibilidade de prever certos erros de execução e criar soluções para elas, impedindo que o programa simplesmente trave ou realize operações ilegais.

Existem vários outros recursos importantes que diferentes linguagens oferecem em maior ou menor grau, como controle de acesso à memória, mecanismos para evitar o acesso à memória não alocada para o programa, controle automático de memória não utilizada (coletor de lixo), entre outros. Os outros critérios que avaliamos legibilidade e facilidade de escrita influenciam na confiabilidade de um programa até certo ponto, pois elas influenciam nas chances de o programador cometer erros ou fazer mau uso de recursos da linguagem.

Outros critérios podem ser utilizados. Entre eles o custo de uma linguagem. Uma linguagem com baixa legibilidade, isto é, difícil de ser escrita, irá demandar muito tempo para que os programadores a aprendam e muito tempo para que o código seja produzido e testado, e tudo isso impacta no custo do projeto. Além disso, aumenta-se a possibilidade de erros, a necessidade de manutenção e o tempo para manutenção. Outros fatores podem impactar no custo da linguagem, como o seu desempenho computacional. Se os programas gerados demorarem muito tempo no processo de compilação ou tiverem mau desempenho no computador do usuário final, também temos um aumento de custo.

Outro critério que pode ser importante em certos projetos é a portabilidade de uma linguagem, ou seja, quão fácil é pegar o programa escrito para

uma plataforma e executar em outra. Por exemplo, se você desenvolve aplicativos para smartphones, é provável que você prefira uma linguagem que permita gerar simultaneamente uma versão da aplicação para iOS e para Android do que ter que desenvolver do zero dois códigos diferentes para o mesmo aplicativo.

Uma consideração final relevante é que os critérios mencionados geralmente são relativos, e dificilmente podemos medi-los de maneira absoluta para uma linguagem. Também precisamos levar em conta o domínio de nossa aplicação antes de avaliar linguagens. Uma linguagem pode ser extremamente simples de ser utilizada em um certo domínio, mas exigir uma quantidade proibitiva de "jeitinhos" para funcionar em outro domínio, tornando-a complicada, pouco confiável e custosa de usar.

EVOLUÇÃO DAS LINGUAGENS DE PROGRAMAÇÃO

Diferentes aspectos de linguagens de programação modernas podem parecer arbitrários ou confusos para quem está começando agora. Muitas vezes, a explicação sobre por que algo é da maneira que conhecemos é histórica: aquele recurso é uma herança de linguagens anteriores que por algum motivo foram bem-sucedidas em nichos específicos e a geração seguinte de linguagens optou por incorporar elementos delas para facilitar a transição de programadores dessas linguagens.

Paradigmas de programação

Ao longo da história da programação, conforme novas linguagens foram surgindo, não apenas facilidades e recursos novos foram criados, mas também maneiras completamente diferentes de se pensar e organizar um programa.

Essas diferentes formas de se pensar um programa são conhecidas como **paradigmas de programação**. A possibilidade de se aplicar ou não um paradigma está intimamente ligado aos recursos oferecidos pela linguagem de programação.

Alguns exemplos de paradigmas de programação são o paradigma imperativo, o paradigma procedural, o paradigma funcional, o paradigma lógico e a programação orientada a objeto.

É importante conhecer esse conceito para poder entender o que diferentes linguagens trouxeram de novo para a programação.

História das linguagens de programação

Ao longo das próximas páginas, estudaremos a história de algumas linguagens de programação e de suas contribuições. Essa história é bastante resumida e obviamente mal cobrirá uma fração das linguagens de programação já desenvolvidas, mas testemunharemos alguns marcos importantes que moldaram as linguagens modernas, bem como os paradigmas em alta atualmente.

Linguagem de máquina (binário)

Podemos imaginar qualquer sistema computacional, de maneira simplificada, em termos de três tipos de componentes: uma unidade de processamento (*CPU – Central Processing Unit*), uma memória de trabalho (tipicamente a memória RAM nos computadores modernos) e dispositivos de entrada e saída de dados.

A Unidade Central de Processamento, ou CPU é a parte mais importante, e também mais complexa

de um computador. Ela tipicamente possui uma unidade de processamento aritmético e lógico (*ALU – Arithmetic and Logic Unit*), além de outros módulos auxiliares, módulos de controle, entre outros.

A ALU, além de vários dos outros módulos que uma CPU pode conter, é constituída por diversos circuitos eletrônicos para realizar diferentes operações, incluindo as mais básicas, como somas aritméticas.

Os circuitos básicos da CPU são compostos por componentes eletrônicos capazes de realizar chaveamento: permitir ou bloquear a passagem de sinal elétrico. No passado, os componentes utilizados para o chaveamento eram as válvulas termiônicas, que lembram muito as lâmpadas incandescentes: um bulbo de vidro que emite calor ao receber corrente elétrica. Porém, elas eram projetadas de tal modo que a alta temperatura provocava uma fuga de elétrons de um catodo para um ânodo, permitindo a passagem de corrente.

Atualmente, os componentes utilizados são os transistores de efeito de campo (FET), que ao receber um certo nível de tensão em um de seus terminais cria uma região polarizada permitindo a passagem de corrente entre os outros dois terminais. Cortando-se a tensão do terminal de

controle, interrompe-se a passagem de corrente entre os outros dois terminais.

A famosa linguagem binária que tanto ouvimos falar vem do fato de que informações no computador, em seu nível mais fundamental, são representadas por sinal elétrico, sendo transmitido ou bloqueado pelos transistores (ou válvulas, nos primeiros computadores eletrônicos). Convencionou-se que o **nível lógico 0** ou **nível lógico baixo** corresponde à ausência de sinal elétrico e o **nível lógico 1** ou **nível lógico alto** corresponde à presença de sinal elétrico.

Em nosso dia a dia, utilizamos a chamada "base 10" para contar. Isso significa que temos 10 dígitos numéricos: os dígitos de 0 até 9. Sempre que precisamos representar um número superior à quantidade de dígitos que temos disponíveis, utilizamos outras "casas" (dezena, centena, milhar etc.), que consistem em multiplicar nossos dígitos básicos por potências de 10 (10, 100, 1000...).

Os mesmos números que utilizamos em nosso dia a dia podem ser reescritos utilizando diferentes bases numéricas, utilizando a mesma lógica. A **base binária**, ou **base 2**, é um sistema de contagem que possui apenas dois dígitos: o 0 e o 1. Sempre que precisamos contar valores superiores, utilizamos outras casas, cada qual correspondente a

multiplicar os dígitos básicos por potências de 2 (2, 4, 8, 16, 32...).

A álgebra booleana, desenvolvida por George Boole, auxiliou os cientistas da computação e engenheiros eletrônicos a criar circuitos para fazer cálculos em base 2 utilizando os dispositivos de chaveamento básicos. Toda informação é representada numericamente utilizando a base binária. Circuitos recebem um conjunto de entradas que podem ou não conter sinais elétricos (ou seja, “0” ou “1”), realizam operações lógicas entre essas entradas e temos outro número como resultado.

Assim como os dados, as próprias instruções do computador podem ser mapeadas por números binários. Se um certo conjunto de zeros e uns for lido da memória, um circuito específico é ativado – por exemplo, a soma de dois números inteiros. Em seguida, os próximos dois blocos de zeros e uns lidos da memória serão interpretados como os operandos dessa conta e serão somados.

Chamamos de **linguagem de máquina** os conjuntos de números binários que mapeiam todas as instruções que uma CPU é capaz de realizar, e os primeiros programas de computador eram escritos diretamente em binário.

Linguagem assembly

Como você pode imaginar, escrever programas completos utilizando apenas conjuntos de zeros e uns pode ser muito difícil e trabalhoso. Mas ainda há um agravante: cada modelo de CPU possui a sua própria linguagem de máquina! O programa que você escreve para um tipo de computador não irá rodar em outros computadores.

Além disso, como as instruções correspondem a circuitos específicos da CPU, para aprender bem a linguagem de máquina, é necessário ter um conhecimento profundo da arquitetura utilizada no computador.

Na virada da década de 40 para a década de 50, o trabalho teórico da matemática Kathleen Booth propôs a criação de uma linguagem simbólica para substituir a linguagem de máquina. Esse trabalho é considerado a invenção da chamada linguagem ***Assembly***, também conhecida como "linguagem de montagem".

Pouco depois, David Wheeler desenvolveu um programa considerado o primeiro ***assembler***, um tradutor de linguagem Assembly para linguagem de máquina.

Ao invés de escrever sequências de números representando instruções, os programadores passaram a usar **mnemônicos**, sequências de

letras curtas representando os diferentes comandos. Adicionalmente, em *Assembly* é possível dar nome a diferentes regiões do código ou dados do programa, facilitando as operações com dados e saltos no código, além de reduzir a necessidade de se fazer cálculos complicados de posições relativas na memória.

Apesar de ser mais fácil do que programar diretamente em linguagem de máquina, a linguagem *Assembly* ainda é traduzida "de um para um" para binário. Isto é, cada instrução em *Assembly* corresponde a exatamente uma instrução em binário. Na prática, ainda estamos escrevendo programas específicos para uma arquitetura de processadores e precisamos ter um domínio muito grande sobre detalhes dessa arquitetura para que nossos programas funcionem. Vale destacar que essa equivalência do *Assembly* à linguagem de máquina implica na existência de várias linguagens *Assembly diferentes*. Cada processador diferente possui a sua própria versão da linguagem, que pode ter mnemônicos radicalmente diferentes de outros.

O uso de *Assembly* foi bastante reduzido com a chegada das próximas linguagens, mas ele ainda sobreviveu por bastante tempo em alguns nichos, normalmente onde era necessário ter um controle mais preciso de cada operação que a máquina irá realizar, como em sistemas operacionais e drivers

de dispositivos, ou em situações onde era necessário ter muito desempenho computacional, como alguns videogames.

Atualmente mesmo nesses nichos é raro encontrar programas completos em *Assembly*, sendo o mais comum encontrarmos um ou outro trecho curto de *Assembly* realizando uma operação mais crítica misturado com outras linguagens mais modernas.

As primeiras linguagens de alto nível

Nos anos 50, com o Assembly ainda sendo novidade, algumas pessoas começaram a desenvolver a noção de que programar deveria ser mais fácil. Foi nesse período que surgiram as primeiras linguagens de **alto nível**.

Uma linguagem de alto nível não significa, necessariamente, uma linguagem *melhor* do que outra. O "nível" de uma linguagem diz respeito à sua proximidade da máquina ou do ser humano. Linguagens de alto nível são linguagens mais próximas do ser humano, ou seja, que geram códigos que são facilmente compreendidos por pessoas. Já as linguagens de **baixo nível** são mais próximas da máquina, e seus comandos manipulam recursos bastante específicos do computador. A linguagem Assembly possui o nível mais baixo possível, pois estamos manipulando diretamente quais componentes do processador serão utilizados.

Uma dessas pessoas foi Grace Hopper. Enquanto ela trabalhava no projeto do computador UNIVAC I, defendeu uma ideia radical: programas deveriam ser escritos em língua inglesa, e não apenas em termos de símbolos matemáticos e lógicos. A ideia foi ridicularizada por seus colegas e superiores, porque "computadores não entendem inglês". Ela insistiu na ideia e desenvolveu um compilador (um programa que gera um código Assembly a partir de instruções escritas em outra linguagem) para uma linguagem que ela desenvolveu, conhecida por FLOW-MATIC, ou B-0 (*Business Language version 0*). A partir desse projeto surgiu a linguagem COBOL, que foi adotada em larga escala em projetos corporativos. Até hoje alguns programas em COBOL sobrevivem em sistemas antigos em bancos e outras grandes empresas com aplicações críticas.

Outra linguagem que surgiu na época foi o FORTRAN. Desenvolvida na IBM, ela ganhou popularidade rapidamente para computação científica de maneira geral. Comparada com outras linguagens surgindo na mesma época e mesmo com linguagens mais modernas, ela possui um ótimo desempenho computacional, sendo muito mais próximo do desempenho de programas escritos diretamente em Assembly do que várias outras linguagens de alto nível. Essa linguagem ainda sobrevive nos meios científicos devido à grande variedade de

bibliotecas de código pronto escritos para ela, o que facilita bastante o trabalho dos cientistas, que não precisam reimplementar todas as fórmulas e modelos do zero.

Ainda na década de 50 surgiu outra linguagem influente até os dias de hoje, o LISP. Ela foi criada originalmente por John McCarthy como uma maneira de se descrever matematicamente um programa de computador, e toda sua ideia foi baseada em uma área de estudo conhecida como **cálculo lambda**. Steve Russell conseguiu desenvolver em *Assembly* um interpretador de LISP, isto é, um programa que lê um código escrito em LISP e executa suas instruções. A linguagem ganhou popularidade rapidamente no meio acadêmico, especialmente entre os primeiros estudiosos de Inteligência Artificial. Ela foi a primeira representante de uma forma muito diferente e baseada em matemática de se pensar os programas, conhecida por **paradigma funcional**. Algumas linguagens utilizadas hoje em dia são consideradas dialetos de LISP, como o Clojure, e outras, também funcionais ou não, foram influenciadas por conceitos do LISP, como Haskell, Elixir, Python e Julia.

Programação estruturada

Ao final da década de 50, um comitê de cientistas americanos e europeus produziu a especificação ALGOL-60, que deu base à linguagem de mesmo nome. Dentre as novidades produzidas por essa linguagem, duas se destacam:

- Blocos aninhados: a ideia de que podemos ter trechos de código com início e fim, e esses blocos podem estar contidos dentro de outros blocos.
- Escopo: a ideia de que um bloco poderia ter seus próprios dados (variáveis), e esses dados seriam inacessíveis fora desse bloco.

Ao longo da década de 60, surgiram outras versões do ALGOL, bem como novas linguagens incorporando novidades trazidas pelo ALGOL e apresentando novos conceitos e recursos.

Foi nessa época que se consolidou a ideia de programação estruturada. Foram definidas algumas estruturas "padrão", como:

- Estruturas condicionais: blocos de códigos que podem ou não ser executados dependendo de uma ou mais condições. É uma forma de ramificar o código.
- Malhas de repetição: blocos de código que podem se repetir até que uma certa condição mude.
- Funções: blocos de código que podem ser chamados pelo nome.

Junto dessas ideias, também veio a rejeição a outros recursos utilizados nas linguagens da época. O exemplo mais conhecido são as instruções do tipo **goto**. O goto (do inglês "go to" – "ir até") é um tipo de instrução que faz com que o programa "salte" diretamente para um trecho específico, sem qualquer tipo de controle condicional ou estrutura para providenciar um retorno seguro para o ponto de origem. As estruturas de programação estruturada são muito mais legíveis e menos propensas a erros, e o uso delas no lugar do goto passou a ser recomendado cada vez mais veementemente, ao ponto de algumas linguagens surgirem sem sequer ter um comando "goto" ou equivalente.

Programação imperativa

Já a partir da década de 60 e ao longo da década de 70, o surgimento de algumas linguagens que viriam a se tornar extremamente populares consolidou um paradigma de programação conhecido por "imperativo". Estudaremos esse paradigma em mais detalhes posteriormente, mas seus programas são caracterizados por blocos sequenciais de instruções.

Uma linguagem muito influente na época foi o Pascal. Uma das motivações de seus criadores foi aproveitar os importantes conceitos trazidos pelo ALGOL-68 (uma evolução do ALGOL-60), mas

sem a complexidade da linguagem. Basicamente, o Pascal foi feito para ser tão bom quanto o ALGOL-68, mas mais fácil de ser utilizado.

O Pascal foi tão bem-sucedido em ser intuitivo que rapidamente ganhou um nicho: o ensino de programação. Diversas universidades pelo mundo adotaram o Pascal por vários anos como a primeira linguagem a ser ensinada para engenheiros e cientistas da computação. Futuramente, incorporando conceitos de programação orientada a objeto, o Pascal deu origem a outra linguagem, Delphi, que foi muito popular entre os anos 90 e início dos anos 2000 para o desenvolvimento de programas em Windows.

Outra linguagem imperativa muito importante que surgiu na década de 60 foi a linguagem C. Os programadores Dennis Ritchie e Ken Thompson, trabalhando para o Bell Labs, precisavam portar o sistema operacional Unix escrito em *Assembly* para um computador diferente. Cientes do problema que seria ter que reescrever o código Assembly do zero para cada novo computador que surgisse, eles decidiram que deveriam reescrevê-lo utilizando uma linguagem de alto nível. Assim, qualquer computador que tivesse um compilador para aquela linguagem poderia facilmente receber uma versão do Unix.

Eles partiram da linguagem BCLP, removeram o que consideravam excesso de complexidade e chegaram a uma linguagem chamada “B”. Porém, a linguagem B não permitia que eles explorassem alguns recursos específicos e muito úteis de alguns processadores, e os programas produzidos por ela eram lentos e ineficientes. Assim, eles desenvolveram uma versão melhor dessa linguagem, chamada C.

A linguagem C se tornou extremamente popular por ser uma linguagem de alto nível e, ao mesmo tempo, alto desempenho. Compiladores de linguagem C rapidamente surgiram para diversas plataformas diferentes.

Hoje em dia, outras linguagens com mais recursos e facilidades substituíram a linguagem C na maioria das aplicações, mas ela ainda sobrevive com força em domínios onde o acesso a recursos de baixo nível se faz necessário, como o desenvolvimento de sistemas operacionais e os sistemas embarcados, por conta de sua ótima performance e sua capacidade de manipular recursos específicos de memória e processador mesmo sendo uma linguagem de alto nível.

Muitas das linguagens modernas que utilizamos hoje em dia foram fortemente influenciadas em linguagem C, utilizando os mesmos comandos bá-

sicos, operadores e sintaxe em geral. A expressão "C-like" (do inglês "parecido com C") é frequentemente usada para descrever essas linguagens, normalmente caracterizadas por ter comandos como "if", "else", "while", "for", utilizar chaves para delimitar blocos e ponto-e-vírgula para encerrar linhas de código.

Programação orientada a objeto

Também na década de 70 começou a surgir um outro conceito, uma forma diferente de pensar os programas: a programação orientada a objetos. Ela também é um paradigma diferente, que será estudado posteriormente. Mas ela traz conceitos e regras que facilitam bastante a modularização de códigos, permitindo um grande reaproveitamento de código pronto, evitando retrabalho e aumentando a produtividade.

Após alguns experimentos bem-sucedidos, começando com um programa de design auxiliado por computador chamado de Sketchpad e linguagens como Simula e Smalltalk, algumas linguagens novas começaram a surgir introduzindo os conceitos da programação orientada a objeto em linguagens já existentes.

Um exemplo notório foi o surgimento da linguagem C++, que acrescenta ideias como classes, objetos, herança e polimorfismo à já existente linguagem

C. Ela combina a sintaxe já familiar da linguagem C, bem como seu bom desempenho computacional, facilitando a adaptação de programadores já familiares com o C.

Várias outras linguagens orientadas a objeto surgiram entre os anos 80 e 90 e, junto com os benefícios da programação orientada a objeto, frequentemente traziam junto um conjunto de ferramentas para auxiliar na produção cada vez mais fácil e rápida de aplicações completas, como editores de código, ferramentas para auxiliar nos testes e quantidades enormes de código pronto (as famosas "bibliotecas") automatizando boa parte das tarefas. Dentro dessa tendência podemos destacar iniciativas como o Delphi, da Borland, e as linguagens da Microsoft, como o Visual Basic e o Visual C++ (o C++ junto com diversas ferramentas proprietárias da Microsoft para facilitar o desenvolvimento de aplicações gráficas para Windows).

Uma linguagem muito popular e influente na época, também "descendente" da linguagem C, foi a linguagem Java. Parte de sua proposta era a ideia "*write once, run anywhere*" ("escreva uma vez, rode em qualquer lugar").

Programas escritos em Java não são compilados diretamente para o computador. Isto é, a linguagem não gera um binário específico para cada máquina.

Ao invés disso, o Java vem com uma máquina virtual. Essa máquina virtual age como um interpretador para uma linguagem de máquina própria, como se ela mesma fosse um computador idealizado. Qualquer arquitetura de computador ou sistema operacional que possua a máquina virtual Java pode, em tese, executar qualquer programa escrito em Java sem a necessidade de ajustes no código.

Programação web

Com a presença cada vez maior da internet no dia a dia dos usuários, ficou cada vez mais comum também a migração da computação dos computadores pessoais diretamente para a internet. Muitos recursos utilizados pelos usuários finais deixaram de ser programas instalados em seus computadores pessoais e passaram a ser serviços que podem ser acessados de qualquer dispositivo, incluindo smartphones, através de um navegador na internet. Outras mudanças ocorrendo simultaneamente no mundo da computação incluem o processamento paralelo cada vez mais disponível (através de processadores com múltiplos núcleos e a partir de clusters formados por várias máquinas em rede) e preocupações cada vez maiores com segurança.

Essa grande mudança na forma de se consumir software também impacta na escolha das lingua-

gens adotadas. Tendo a internet como ponto de partida, é necessário um grande uso de linguagens suportadas pelos navegadores da internet, como o HTML (HyperText Markup Language, utilizada para estruturar páginas) e o JavaScript (linguagem interpretada por navegadores).

Do "outro lado", nos servidores responsáveis por oferecer os serviços, é importante que as linguagens apresentem bom desempenho, fácil integração com outros serviços, um bom nível de segurança (afinal, falhas podem comprometer milhares de usuários de uma vez) e, dada a natureza cada vez mais volátil do mercado de software, muita facilidade de manutenção e atualização dos códigos. Inicialmente esse movimento foi dominado por linguagens como Java e as linguagens de web da Microsoft.

Atualmente, linguagens de script, tipicamente linguagens interpretadas com grandes facilidades para produzir pequenos pedaços de lógica, se tornaram extremamente relevantes nesse meio. Linguagens como Python e Ruby são utilizadas para colocar sistemas completos no ar utilizando relativamente poucas linhas de código.

Além disso, com a procura cada vez maior de empresas e pessoas físicas por conhecimentos básicos de programação para automatizar peque-

nas tarefas e realizar análises pontuais de dados, linguagens de script com sintaxe fácil e recursos para auxiliar com estatística e visualização de dados começaram a ser bastante procuradas. Isso provocou nesse público de "programadores não profissionais" um grande interesse em linguagens como Python, R e Julia.

Também nessa linha sobre a grande disponibilidade de dados, temos uma demanda cada vez maior por inteligência artificial e aprendizado de máquina, onde predominam linguagens que apresentem grande facilidade para utilizar esses algoritmos, como o próprio Python, quanto, em casos específicos, linguagens com alto desempenho computacional, como o C++, quando as linguagens mais "modernas" não oferecem o desempenho necessário.

Os requisitos atuais de segurança, devido à presença cada vez maior do software em nossas vidas e a preocupação com sua indisponibilidade ou com vazamento de dados, também levaram ao desenvolvimento de novas linguagens com recursos específicos.

Programadores de "baixo nível", que tipicamente utilizavam linguagem C e resistiam a adotar linguagens com mais recursos de segurança por conta de seu desempenho computacional pior começam a usar linguagens como Rust e Go, que

possuem a maioria das vantagens do C e recursos adicionais de segurança, tornando os programas mais confiáveis.

Em outras áreas, como o processamento de transações financeiras, as linguagens funcionais, como o Clojure, se tornaram extremamente atraentes devido à sua simplicidade e rigor matemático, gerando programas legíveis e de comportamento extremamente determinístico (ou seja, previsível).

Um último efeito que vale à pena ser mencionado é a influência "cruzada" entre diferentes linguagens, fazendo com que versões modernas de diversas linguagens tenham suporte em maior ou menor grau a diversos recursos originalmente associados a outras linguagens, ou até mesmo suporte à adoção parcial ou total de paradigmas completamente distintos do original, por exemplo a linguagem C++, originalmente orientada a objetos, oferecendo recursos característicos de linguagens funcionais.

CONSIDERAÇÕES FINAIS

As linguagens de programação foram e são ferramentas essenciais na vida moderna. A evolução das linguagens ao longo do tempo permitiu aos desenvolvedores darem passos cada vez maiores e produzirem programas cada vez mais completos e em menor tempo, resultando em nossa vida atual regida por software.

Elas existem em grande variedade, e diferentes linguagens oferecem diferentes recursos. Estudamos algumas formas de comparar diferentes linguagens, mas não podemos nos esquecer de que essa comparação deve ser sempre do ponto de vista do tipo de aplicação que gostaríamos de desenvolver – o domínio de programação ao qual nosso projeto pertence. Diferentes linguagens oferecerão vantagens em diferentes situações.

Além de linguagens diferentes, temos também formas completamente distintas de planejar o nosso programa, conhecidas como paradigmas de programação. Cada linguagem oferece recursos que suportam um ou mais paradigmas diferentes, e assim como escolhemos diferentes linguagens de acordo com a situação, também podemos escolher diferentes paradigmas conforme a nossa necessidade. Com isso, verificamos de maneira panorâmica diversas linguagens de programação e alguns paradigmas.

Referências Bibliográficas & Consultadas

FCORREA, A. G. D. **Programação** I. São Paulo: Pearson, 2015. [Biblioteca Virtual].

FELIX, R. (Org.). **Programação orientada a objetos**. São Paulo: Pearson, 2016. [Biblioteca Virtual].

LEAL, G. C. L. **Linguagem, programação e banco de dados:** guia prático de aprendizagem. Curitiba: Intersaberes, 2015. [Biblioteca Virtual].

SANTOS, M. G. dos; SARAIVA, M. de O.; GONÇALVES, P. de F. **Linguagem de programação**. Porto Alegre: SAGAH, 2018. [Minha Biblioteca].

SEBESTA, R. W. **Conceitos de linguagens de programação**. 11. ed. Porto Alegre: Bookman, 2018. [Minha Biblioteca].

SILVA, E. A. da. **Introdução às linguagens de programação para CLP**. São Paulo: Blucher, 2016. [Biblioteca Virtual].

SILVA, F. M. da; LEITE, M. C. D.; OLIVEIRA, D. B. de. **Paradigmas de programação**. Porto Alegre: SAGAH, 2019. [Minha Biblioteca].

TUCKER, A.; NOONAN, R. **Linguagens de programação:** princípios e paradigmas. 2. ed. Porto Alegre: AMGH, 2014. [Minha Biblioteca].

FAM
ONLINE