IFC CAMPUS BLUMENAU
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

PARADIGMAS DE PROGRAMAÇÃO

apostila

Blumenau
2022

SUMÁRIO

5.2. Exercícios - (Luana)

6. Paradigma Imperativo - (Leonardo)
   6.1. Paradigma Procedural - (Eduardo)
      6.1.1. Linguagem C
      6.1.2. Exercícios
   6.2. Paradigma Orientado a Objetos - (Henrique)
      6.2.1. Linguagem Java
      6.2.2. Exercícios
7. Paradigma Declarativo - (intro - Vítor)
   7.1. Paradigma Lógico - (Erick)
      7.1.1. Linguagem Prolog - (Erick)
         7.1.1.1. Fatos e Regras - (Erick)
         7.1.1.2. Exemplos de Consulta - (Erick)
         7.1.1.3. Operadores - (Erick)
         7.1.1.4. Desvantagens - (Erick)
         7.1.1.5. Principais Usos do Paradigma Lógico - (Erick)
         7.1.1.6. Exercícios - (Erick)

   7.2. Paradigma Funcional - (Gabriel)
      7.2.1. Linguagem Haskell - (Gabriel)
      7.2.2. Operadores - (Gabriel)
      7.2.3. Sistema de Tipos - (Gabriel)
      7.2.4. Funções - (Gabriel)
      7.2.5. Criando uma Função - (Gabriel)
      7.2.6. Agregação de vários componentes (DU BOIS, s.d.) - (Gabriel)
      7.2.7. Avaliação Preguiçosa (lazy evaluation) - (Gabriel)
      7.2.8. Condicionais - (Gabriel)
      7.2.9. Let - (Gabriel)
      7.2.10. Listas - (Gabriel)
      7.2.11. Ferramentas - (Gabriel)
      7.2.12. Desvantagens - (Gabriel)
      7.2.13. Exercícios - (Gabriel)

1. INTRODUÇÃO

Essa apostila visa na construção de conhecimentos sobre paradigmas de programação. Dada como um método de avaliação durante o segundo semestre de 2021 pelo professor Ricardo de la Rocha Ladeira. Abordaremos os principais conceitos vistos e comentados em sala, apresentando imagens e dando exemplos.

Segue alguns tópicos desenvolvidos na apostila, como: (a sua função, o que é, como começou, para que serve, quais são os tipos existentes, linguagens de programação e suas gerações, classificação das linguagens, métodos, sistemas, arquitetura de computadores, qual é a importância de uma linguagem de computadores, quais são os paradigmas existentes, etc.). Em alguns tópicos será encontrado alguns exercícios propostos para entender melhor o conceito de paradigmas de programação.

2. LINGUAGENS DE PROGRAMAÇÃO: EVOLUÇÃO E CARACTERÍSTICAS

O presente tópico aborda de forma introdutória o conteúdo de Paradigmas da Programação. Algumas palavras chave e noções acerca das linguagens de programação são estudados neste capítulo, como suas características, divisões e classificações, que serão aprofundadas nos tópicos posteriores.

O conteúdo deste capítulo está dividido em: palavras-chave e conceitos iniciais das linguagens de programação, motivos para se estudar tais conceitos, domínios (aplicações) das linguagens de programação e classificação delas por gerações. Por fim, algumas dicas de leituras complementares e exercícios também estão disponíveis.

2.1. PALAVRAS-CHAVE E CONCEITOS

Antes de começar a estudar os conceitos de linguagens de programação é importante estar a par de alguns termos:

Algoritmo: uma sequência suficientemente detalhada de passos a serem executados para se cumprir uma tarefa (CESAR; MEDINA; FERTIG, 2005). Não se restringe a programação, pois é utilizado em outras áreas (engenharia, administração…). Um exemplo clássico pode ser o de uma receita culinária.

Programa: um algoritmo que foi convertido para uma linguagem compreendida pelo computador. Pode ser entendido como um conjunto de instruções que será executado pelo processador em uma determinada ordem (CESAR; MEDINA; FERTIG, 2005).

Linguagem de programação (de alto nível): linguagem com um conjunto de regras que permite a conversão de um algoritmo para um programa. Se tratam de linguagens mais convenientes para o ser humano escrever um algoritmo quando comparado à linguagem de máquina, mas que precisam ser traduzidos ou interpretados para esta segunda, tendo em vista que é apenas o que o computador “compreende” (TANEMBAUM, 2009). Segundo Ladeira (2021), todas as linguagens de programação têm:

- instruções: print;
- palavras e símbolos reservados: break, =;
- regras: while () {};

- significado: os elementos da linguagem possuem significado (“+” pode fazer soma e/ou concatenação dependendo da linguagem);
- sistemas de tipos (será explicado em outro tópico);

2.2. MOTIVOS PARA ESTUDAR CONCEITOS DE LINGUAGENS DE PROGRAMAÇÃO

De acordo com Sebesta (2011), é comum que os estudantes se perguntem qual a importância de se estudar linguagens de programação, por isso ele cita alguns dos principais motivos:

Incrementar a capacidade de expressar ideias: a capacidade de uma pessoa pensar e expressar ideias depende do seu conhecimento na linguagem natural, que por sua vez possuem limitações impostas pelas regras gramaticais. Da mesma forma, as linguagens de programação também possuem limitações, como estruturas de controle, estruturas de dados ou abstrações que podem ser usadas. Sebesta defende o aprendizado de conceitos de linguagens de programação, pois dessa forma o programador pode aprender novos recursos ou conceitos que poderão ser agregados ao seu arsenal de resolução de problemas e possivelmente aplicados mesmo nas linguagens que não as suportam diretamente.

Incrementar o conhecimento para escolher linguagens apropriadas: Sebesta defende que os programadores que conhecem poucas linguagens de programação têm a tendência de sempre escolher a que possuem mais afinidade, mesmo se for pobre em adequação ao projeto. Aprender sobre várias linguagens e paradigmas de programação possibilita escolher uma mais adaptada às características de cada problema, dado que, certas linguagens resolvem um mesmo problema mais facilmente que a outra. Em virtude disso, não é preciso que o programador simule seus recursos em certa linguagem apenas por estar mais familiarizado com ela.

Habilidade em aprender novas linguagens: quanto mais se aprende sobre a gramática de sua língua nativa, mais fácil é aprender novos idiomas. Da mesma forma, quanto mais se aprende sobre conceitos de linguagens de programação, mais fácil é migrar de uma para outra, habilidade importante tendo em vista que a

tecnologia e as tendências de linguagens de programação mudam constantemente, o que requer aprendizado contínuo.

Incrementar vocabulário técnico: conhecer os termos técnicos é uma parte importante para o aprendizado de novos assuntos. Ter contato com várias linguagens de programação é uma forma de aumentar esse vocabulário.

Melhor entendimento da importância da implementação: compreender os detalhes de implementação de uma linguagem contribui para o entendimento de porque ela foi criada dessa forma. Portanto, ter conhecimento sobre os conceitos de linguagens de programação facilita utilizá-las de forma mais inteligente e como foi pensada para ser usada. Além disso, implementar um algoritmo em várias linguagens permite perceber os detalhes que tornam uma solução mais eficiente que as outras.

Melhor uso das linguagens já conhecidas: as linguagens de programação costumam ser complexas e dificilmente são utilizadas por completo por um programador. Ao aprender sobre os conceitos de linguagens de programação o programador pode compreender as partes desconhecidas das linguagens que já utilizava, reaproveitando os conceitos.

2.3. DOMÍNIOS: ÁREAS DE APLICAÇÕES DAS LINGUAGENS

O computador possui uma grande variedade de aplicações, portanto existe uma grande variedade de linguagens de programação com objetivos distintos. As principais aplicações dos computadores de acordo com Sebesta (2011) são:

Aplicações Científicas: aplicações importantes desde o primórdio da computação na década de 40. Utiliza estruturas de dados simples, sendo as mais comuns as matrizes e os vetores, mas muitas operações aritméticas com números flutuantes. As estruturas de controle mais comuns são os laços de contagem e a seleção. A velocidade era uma grande preocupação, principalmente porque no começo da computação o grande concorrente desse tipo de linguagem era o Assembly, que pode ser extremamente veloz. As primeiras linguagens de alto nível para aplicações científicas foram o Fortran, que ainda é utilizado por ser muito eficiente, e o ALGOL 60, que não era exclusivamente para aplicações científicas.

Aplicações Empresariais: os computadores passaram a ser utilizados para aplicações empresariais na década de 1950. No início as empresas compravam grandes computadores que possuíam linguagens especiais próprias. A primeira linguagem de alto nível desse tipo foi o COBOL que surgiu na década de 60, mas que ainda é utilizado atualmente, principalmente em *softwares* legados. Esse tipo de linguagem de programação precisa lidar com a criação de relatórios, representação e manipulação de dados de forma precisa e operações aritméticas decimais. Esse tipo de linguagem sofreu poucas mudanças ao longo dos anos.

Inteligência Artificial: caracterizado pelo uso da computação simbólica – nomes são manipulados ao invés de números e listas ligadas são mais utilizadas do que os convencionais vetores. Requer mais flexibilidade do que outras áreas da computação. A primeira linguagem popular para esse propósito foi o LISP (paradigma funcional), que surgiu em 1959. Outra linguagem comum para esse propósito é o PROLOG, que se trata da programação lógica (paradigma lógico). Até a década de 90 o LISP foi amplamente utilizado, mas a partir de então algumas aplicações de IA foram escritas em linguagens de sistema como C.

Programação de Sistemas: se tratam dos sistemas operacionais e ferramentas de suporte à programação de um sistema de computação. Tem como característica a necessidade de rodar continuamente e de forma eficiente. Também precisam ter recursos de baixo nível para fazer a ponte entre os programas aplicativos e o hardware. As primeiras linguagens de programação de sistemas eram das grandes fabricantes de computadores e rodavam em suas máquinas, a citar a PL/S da IBM, a BLISS da Digital e o ALGOL Estendido da Burroughs. O UNIX e o Linux são escritos quase inteiramente em C, o que traz uma série de vantagens como a sua portabilidade (pode ser compilado em várias arquiteturas). C é uma linguagem de baixo nível, eficiente e dá muita liberdade ao programador, com poucas restrições de segurança. Por um lado, o programador tem muito poder com a linguagem C, que é o que os desenvolvedores de sistemas operacionais procuram. Por outro lado, a falta de recursos de segurança é algo perigoso para softwares grandes e importantes, o que requer muito cuidado. Outra linguagem comum na programação de sistemas é o C++.

Software para a Web: aplicações que utilizam um conjunto de tecnologias, como linguagens de marcação, como o HTML (não é linguagem de programação), linguagens de estilização, como o CSS, e linguagens de script, como o JavaScript. Esse tipo de aplicação costuma requerer dinamicidade na apresentação dos conteúdos, que pode ser solucionado com tecnologias como o PHP, capaz de inserir conteúdo HTML antes de retorná-lo ao usuário (no back-end), ou com JavaScript, inserido-o com o carregamento da página (no front-end).

2.4. GERAÇÕES DAS LINGUAGENS DE PROGRAMAÇÃO

Uma das formas de se classificar uma linguagem de programação é quanto a sua geração. Entretanto, essa classificação não é totalmente precisa, pois o número de gerações varia de acordo com o autor e costuma variar entre 4 a 6 gerações. De acordo com Sebesta (apud OLIVETE JÚNIOR, 2022), a classificação em seis gerações classifica as linguagens da seguinte forma, como também observado na Figura 1:

Figura 1: diagrama linguagens de programação por geração

Fonte: https://apretatech.medium.com/linguagens-de-programa%C3%A7%C3%A3o-e-suas-classifica%C3%A7%C3%B5es-708cfa69fa3a

1ª geração - Binário (0 ou 1): linguagem de máquina em notação binária, ou seja, o programa é um arquivo não legível ao ser humano. Um programa escrito dessa forma consiste em uma série de instruções (sequências de zeros e uns), que em geral dizem ao processador qual operação fazer e onde estão os dados

(endereços de registradores ou memória). A implementação de um algoritmo complexo em linguagem de máquina é cansativo e sujeito a erros. Não obstante, esse tipo de programação também é dependente da arquitetura do computador, tendo em vista que as instruções disponíveis mudam de uma para a outra.

2ª geração - Assembly: trata-se da linguagem intermediária, também conhecida como linguagem de montagem, que foi criada para diminuir a dificuldade de se programar em notação binária. Ao invés de códigos para representar as operações e endereços, o Assembly usa mnemônicos (abreviações). Dessa forma, as instruções são representações simbólicas das instruções de máquina, sendo necessário realizar a tradução do código para linguagem de máquina antes de sua execução. É legível para o ser humano, mas por ainda ser considerada uma linguagem de programação de baixo nível, ainda é dependente da arquitetura.

3ª geração - Linguagens Procedurais: tratam-se das linguagens de propósito geral também chamadas de linguagens orientadas ao usuário. Surgiram na década de 60 e são linguagens de alto nível multipropósito, legíveis e mais próximas da linguagem natural, se comparada à notação binária e ao Assembly. Não são dependentes da arquitetura, mas precisam ser convertidos para a linguagem de máquina, seja por compilação ou interpretação. Por serem linguagens multipropósito, atuam em vários domínios de aplicação, ou seja, podem ser utilizadas desde a programação de um sistema até a aplicação de um negócio. Como exemplo de linguagens podemos citar C, Pascal, Java, Python, entre outras.

4ª geração - Linguagens aplicativas: linguagens de propósito específico também conhecidas como orientadas à aplicação. Enquadram-se nessa geração, as linguagens criadas para atuar em apenas um domínio, ou seja, que possuem uma área específica de atuação, em razão disso, costumam ter um nível ainda maior de abstração do que as linguagens de terceira geração, visto que, precisam de um menor número de linhas para realizar a mesma tarefa. Adequam-se nessa categoria linguagens como o MatLab, R e SQL, sendo esse último muito utilizado na gerência de Banco de Dados. Os principais objetivos dessa geração eram reduzir os erros de depuração, facilitar a resolução de problemas específicos e reduzir o custo de criação e manutenção de aplicações.

5ª geração - Lógica e Funcional: corresponde às linguagens de inteligência artificial. Em geral, são divididas em linguagens de paradigma funcional ou lógico, que se enquadram como subclassificações do paradigma declarativo. Facilitam a representação do conhecimento, que é algo essencial para a simulação de comportamentos inteligentes. São utilizados em sistemas que usam mecanismos de inteligência artificial (IA), como processadores de linguagem natural. Como exemplos de linguagens, podemos citar Miranda e Haskell do paradigma funcional e Lisp e Prolog do paradigma lógico.

6ª geração - Redes Neurais: linguagens relacionadas com aprendizado de máquina, ou seja, as soluções tecnológicas são capazes de entender o comportamento dos usuários. O objetivo principal dessa abordagem era criar um software que tivesse a capacidade de resolver o problema como um cérebro humano. Em consequência disso, o programa é capaz de se adaptar de uma forma a receber constantemente novos parâmetros.

2.5. EXERCÍCIOS

1 - É importante estudar conceitos de linguagens de programação mesmo se você nunca criar uma linguagem? Por quê?

2 - Cite duas vantagens de se estudar conceitos de linguagens de programação.

3 - Em que geração se enquadra o PHP? E o Miranda?

5 - Cite dois domínios das linguagens de programação.

6 - Quais são as características dos sistemas web? Cite algumas das principais tecnologias utilizadas nesse tipo de sistema.

7 - Qual a diferença entre as linguagens da primeira e da segunda geração?

8 - Por que foram criadas linguagens de alto nível se é possível programar em notação binária em linguagem de máquina?

9 - Quais foram as primeiras linguagens voltadas para aplicações científicas? E para aplicações empresariais?

10 - Por que existem sistemas legados? Por que não são simplesmente modernizados?

11 - Cite duas características que todas as linguagens de programação têm.

12 - Qual a diferença entre algoritmo e programa?

Leitura complementar e links externos

- Organização estruturada de computadores - Andrew S. Tanenbaum e Todd Austin: capítulo 1, tópico 1.1.1 (linguagens, níveis e máquinas virtuais).
- Popularidade das linguagens de programação ao longo dos anos - Tiobe: http://www.tiobe.com/tiobe_index/index.htm
- Popularidade das linguagens de programação 2020 - StackOverflow: https://insights.stackoverflow.com/survey/2020#most-loved-dreaded-and-wanted
- Popularidade das linguagens de programação 2019 - StackOverflow: https://insights.stackoverflow.com/survey/2019#most-loved-dreaded-and-wanted

3. CARACTERÍSTICAS DESEJÁVEIS (CRITÉRIOS DE EVOLUÇÃO)

Assim como o *hardware* dos computadores, que tem evoluído de forma assustadora se tornando continuamente mais poderoso e acessível ao público, conforme observado por Moore em sua “lei” (TANENBAUM, 2009), as linguagens de programação também estão em constante mudança. De acordo com Sebesta (2011), é desejável que as linguagens sejam confiáveis, robustas, tolerantes a falhas, legíveis e portáteis. Tendo isso em vista, é possível concluir de forma resumida que, as próximas gerações de linguagens de programação apresentam uma predisposição a serem mais fáceis de se manipular, e isso vai desde elementos mais difíceis até os mais fáceis, como a própria sintaxe da linguagem.

3.1. CLASSIFICAÇÃO DAS LINGUAGENS DE PROGRAMAÇÃO

Existem inúmeras divergências quanto à classificação das linguagens de programação e, de fato, não existe um consenso, por outro lado, Sebesta (2011) classifica-as em: Imperativas, Orientadas a Objetos, Funcionais e Lógicas. Até o momento, basta saber que a classificação das linguagens se dá dessa maneira, mas entraremos com mais detalhes em cada uma ao longo do desenvolvimento desta apostila.

3.2. O QUE INFLUENCIA O PROJETO DAS LINGUAGENS

Para que algo passe por uma evolução, há um processo onde novas ideias, modificações e adaptações são discutidas ou, de fato, realizadas. Nisso, temos que essas ideias podem ser descartadas, por não apresentarem uma fundamentação suficiente para que realmente houvesse uma contribuição no avanço daquele objeto, ou aprovada, por ostentar uma influência positiva e uma melhora acerca da versão existente até o momento do objeto.

Tendo isso em mente, é possível destacar inúmeras ideias de projetos que, ao longo da história, revolucionaram suas respectivas áreas de atuação. Uma amostra notória no campo da computação é o Modelo de Von Neumann, introduzido na Primeira Geração dos Computadores, que ainda consistia na utilização de tubos de vácuo na construção dos mesmos. Foi nessa geração, de 1946 até 1956, onde o Modelo de Von Neumann foi apresentado, aprovado e é, até hoje, base para a

arquitetura dos computadores atuais, passando por diversos processos de evolução, mas nunca sendo, de fato, descartado.

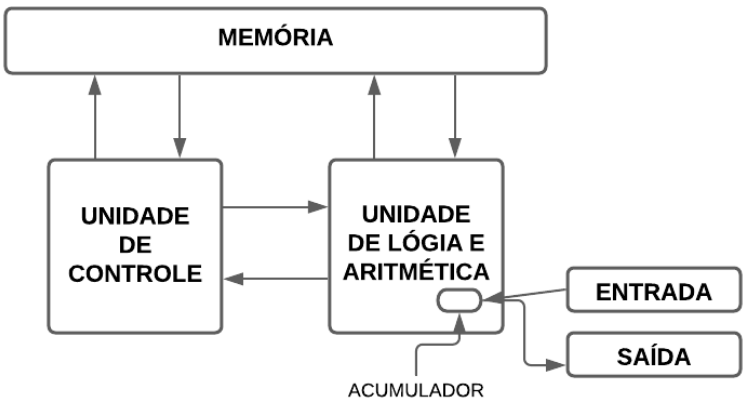
A partir do apresentado, pode-se afirmar que o projeto de uma linguagem de programação pode sim ter influência de uma metodologia já existente. Como exemplo, temos a linguagem de programação C, presente no tópico 5.1.1. Essa linguagem foi criada para o desenvolvimento do Sistema Operacional Unix, que até o momento era escrito em Assembly. Pode-se dizer que C influenciou diretamente linguagens como C++, C#, Objective C e até Java, que será apresentada mais à frente no tópico 5.2.1. Tanto a sintaxe quanto as estruturas presentes em C são populares até hoje, visto que foram extremamente importantes na criação de diversas linguagens que vieram mais tarde. A partir disso, conclui-se que C é uma das influências para o projeto de linguagens de programação.

3.3. ARQUITETURA DE COMPUTADORES

A arquitetura de um computador se trata da organização de seus componentes, e é responsável por definir a forma como o computador funciona, ou seja, como transfere e armazena seus dados e processa informações.

O modelo de arquitetura proposto por Von Neumann, é encontrado na maioria dos computadores atualmente. Consiste em uma entrada/saída de dados; uma memória; uma unidade de processamento (CPU), que contém os registradores; uma Unidade de Controle (CU), e uma Unidade de lógica e aritmética.

Figura 2: Modelo básico de arquitetura de von Neumann



Fonte: Retirado do livro Organização Estruturada de Computadores (2009)

Nesta arquitetura existem determinadas características básicas, como por exemplo: as instruções são executadas de forma sequencial; as instruções e os dados ficam armazenados no mesmo local da memória; cada local da memória possui um endereço, o mesmo será utilizado para identificar a posição de um determinado conteúdo.

A organização dos componentes de um computador tem impacto direto na forma como ele interpreta a linguagem a nível de máquina. Desta forma, mesmo pequenas diferenças podem causar problemas de compatibilidade, o código binário que opera uma em uma organização, não necessariamente será funcional em outra.

Assim sendo, organizações diferentes necessitam conjuntos de instruções com abordagens distintas. Logo a “arquitetura de computadores é a organização de componentes formando um sistema de computador e a semântica ou significado das operações que guiam seu funcionamento”. (FRONTIER e MICHEL, 2003, tradução nossa).

A combinação de uma arquitetura com suas instruções tem um alto impacto no desempenho, velocidade e consumo de energia do sistema. Onde certas arquiteturas se destacam, outras falham e vice-versa.

A arquitetura de computadores influencia na linguagem de programação. Essa influência se dá através de:

- Modelagem de células de memória por meio de variáveis;
- Transmissões de dados, por meio das atribuições;
- Repetições interativas, pois as instruções ficam em posições contíguas da memória;
- Dados e expressões transmitidos entre CPU e memória;
- Uma ação denominada ciclo de busca e execução, carregando o programa da memória na CPU, instrução por instrução, armazenando o endereço de memória da próxima instrução e executando a instrução atual.

3.4. MÉTODOS DE IMPLEMENTAÇÃO

Para que uma linguagem de programação seja implementada ao ponto do *hardware* conseguir executar em sua linguagem de máquina, onde se tem a forma mais primitiva das operações, é necessário que o Sistema Operacional atue como uma interface, fazendo o meio de campo entre o *hardware* e as linguagens, que possuem um maior nível de abstração. Com isso, a função do SO vai desde a gerência dos recursos do sistema, até o controle das operações de entrada e saída.

A partir disso, podemos destacar três métodos de implementação das linguagens de programação, sendo eles: compilação, interpretação e híbrido. Na sequência, será apresentado cada um deles com mais detalhes.

3.4.1. Compilação

Em linguagens compiladas, todo o processo é dividido em duas etapas, o primeiro que é a compilação, e o segundo que é a execução propriamente dita. Para se compilar um programa, é necessária a utilização de um compilador, que nada mais faz que traduzir o código-fonte para a linguagem de máquina, presente no processador. Após a compilação, o código pode ser executado quantas vezes forem necessárias, sem precisar compilar novamente, sendo necessário realizar o processo de novo somente após uma nova edição.

Utilizar linguagens compiladas é interessante, tendo em mente que, após a compilação, o código passa por uma otimização e, visto que, só é necessário compilar novamente após o mesmo ser editado, o processo se torna muito mais rápido e eficiente. Imagine que o intuito seja apenas executar o código, por exemplo, com isso, basta ele ser executado diretamente pelo processador, sem a necessidade de passar pelo processo de compilação novamente.

Exemplos de linguagens que utilizam esse método de implementação são: C, C++ e C#.

3.4.2. Interpretação

Em linguagens de programação que utilizam o método de implementação interpretado, a execução do programa é feita traduzindo o código-fonte linha por linha por um programa interpretador. Ou seja, toda vez que o programa for

executado ele será previamente traduzido. Em razão disso, a linguagem se torna mais flexível, visto que, erros são mais facilmente administrados pelo programador, pois mensagens de erro são mais precisas que o método de implementação compilado.

Apesar da análise do código ser feita de maneira mais rápida, o processo todo acaba se tornando mais lento, justamente pelo fato de ser necessário traduzir toda vez que o programa for executado.

Dentre as linguagens de programação que utilizam esse método, destacam-se: Python, PHP e Ruby.

3.4.3. Híbridos

Assim como seu próprio nome informa, linguagens de programação que utilizam o método de implementação híbrido são aquelas que implementam as duas formas, ou seja, compilado e interpretado, ao mesmo tempo. Nessa combinação, que utiliza vantagens de cada um dos métodos, o código é compilado através de um Just In Time (JIT) Compiler em um bytecode, que se assemelha ao processo de compilar um código-fonte para a linguagem de máquina, mas que, todavia, ainda não está pronto para ser executado diretamente pelo processador.

A partir desse bytecode gerado, que não depende de nenhuma arquitetura, será possível traduzi-lo instrução por instrução para a linguagem de máquina, dependente da arquitetura, através de uma máquina virtual que possui um interpretador, que fará uma série de otimizações.

Das linguagens que usam esse método, uma das que mais se destaca é Java, que possui a Java Virtual Machine (JVM) como sua máquina virtual.

3.5. RECURSOS E CARACTERÍSTICAS DE LINGUAGENS DE PROGRAMAÇÃO

A linguagem de programação é formada por um conjunto de regras sintáticas e semânticas, de implementação de um código fonte. É um sistema de comunicação estruturado, composto por conjuntos de símbolos, palavras-chaves que permitem o entendimento entre um programador e uma máquina.

A lista abaixo mostra algumas características das linguagens de programação:

- **Simplicidade:** quanto mais ortogonal o projeto de uma linguagem, menor é o número necessário de exceções às regras da linguagem. Tornando assim mais fácil de aprender, ler e entender.
- **Ortogonalidade:** um conjunto pequeno de construções primitivas e de maneiras de construir as estruturas de controle e de dados da linguagem que são combinados para a criação.
- **Portabilidade:** é a capacidade de ser compilado e executado em diferentes arquiteturas.
- **Confiabilidade:** quando está de acordo com todas as suas especificações em todas as condições
- **Robustez:** é a capacidade do software em reagir especificamente a defeitos externos.
- **Reusabilidade:** é a possibilidade da reutilização do mesmo código para diversas aplicações.

Segundo o professor Ricardo de la Rocha Ladeira, em uma de suas aulas de 2021, comenta que diversas linguagens diferentes podem resolver os mesmos problemas, mas oferecendo e trabalhando os recursos de formas diferentes. Por exemplo, em programas escritos em C, o gerenciamento de memória é responsabilidade do programador, enquanto programas escritos em C# executam sobre uma máquina virtual que é capaz de gerenciar memória.

Abaixo segue uma lista de alguns recursos disponíveis em linguagens de programação:

- **Coletor de lixo:** processo usado para automação do gerenciamento da memória
- **Loops rotulados:** auxilia na utilização de loops com um nível de aninhamento muito grande, dessa forma, rotula-se cada estrutura de repetição e dentro do loop com as instruções break e continue chama-se o rótulo que deseja.
- **Goto: comando utilizado para salto de instruções.**

- **Recursividade:** é uma função que chama a si mesma até chegar a uma condição de terminação.
- **Regex:** ou expressões regulares é uma forma de identificar expressões regulares, sejam elas cadeias de caracteres, palavras ou padrões de caracteres.
- **Ponteiros/Referências:** é uma variável que armazena o endereço de memória de outra.

Existem outros recursos como: polimorfismo, herança, classe, interface, interferência de tipos, list comprehension, expressões lambda, paralelismo e concorrência e mônadas.

3.6. EXERCÍCIOS

1 - Assinale abaixo a alternativa que apresenta uma característica INDESEJÁVEL acerca das linguagens de programação:

a) Confiabilidade.
b) Robustez.
c) Portabilidade.
d) Complexidade no aprendizado.
e) Reusabilidade.

2 - Assinale a alternativa correspondente à correta classificação das linguagens de programação:

a) Operacionais, Funcionais, Lógicas e Declarativas.
b) Orientadas a Objetos, Biológicas, Lógicas e Empresariais.
c) Imperativas, Orientadas a Objetos, Funcionais e Lógicas.
d) Declarativas, Lógicas, Imperativas e Instrucionais.
e) Imperativas, Científicas, Lógicas e Orientadas a Dados.

3 - Explique, com suas palavras, qual é a função do Sistema Operacional na implementação das linguagens de programação.

4 - Assinale abaixo o modelo de arquitetura mais conhecido no âmbito computacional, que serve, inclusive, como base para os computadores atuais:

a) Modelo de Moore.
b) Modelo de Washington.
c) Modelo de Jobs.
d) Modelo de Creutzfeldt-Jakob.
e) Modelo de Von Neumann.

5 - Explique como se dá a influência da arquitetura de computadores nas linguagens de programação.

6 - Quais são os três métodos de implementação?

7 - Qual é a finalidade de um compilador?

8 - Por qual motivo as linguagens de programação que utilizam o método de implementação se tornam mais flexíveis? Cite duas linguagens que utilizam esse método.

9 - O que seria métodos híbridos?

10 - O que é recursividade? Dê um código de exemplo.

4. SISTEMAS DE TIPOS

Em linguagens de programação, tipos são trabalhados a todo momento, sejam em variáveis, condições, retorno de métodos, parâmetros, estrutura de dados etc. Os tipos podem variar desde um número inteiro, como o tipo primitivo int, até um valor verdade, cuja representação é feita a partir do tipo boolean, a qual podem ser atribuídos dois valores, sendo um verdadeiro e o outro falso. E é a partir disso que existe o sistema de tipos, um conjunto de regras atribuídas aos recursos da linguagem em questão.

Cada linguagem possui seu sistema de tipos, responsável por dar um conjunto de valores possíveis e operações permitidas a determinado tipo. Uma operação possível em inúmeras linguagens é a expressão “2 + 3”, por exemplo. Nela, é realizada uma simples adição através da notação infixa entre dois números. Por outro lado, a notação prefixa não está disponível em tantas linguagens assim como a infixa. Python é uma das diversas linguagens que não aceita essa forma, sendo necessário utilizar outros meios para isso, como recursividade, por exemplo. Já a linguagem Haskell é uma que, dentro do seu sistema de tipos, suporta a notação prefixa.

Os tipos de dados podem ser divididos em numéricos, literais ou lógicos. A classificação deles varia de acordo com cada linguagem e paradigma, por exemplo, a respeito do aprofundamento e da variação de alguns tipos primitivos, mas, de maneira geral, acontece da seguinte forma:

- **Numéricos:** Podem ser divididos em `int`, inteiros, negativos ou positivos; `float`, reais, negativos ou positivos com casa decimal, também chamada de ponto flutuante, ou `double`, reais, negativos ou positivos, com uma precisão maior que o tipo float.

- **Literais:** Abrangem o `char`, um caracter, e a `string`, uma cadeia de caracteres.

- **Lógicos:** Também chamados de booleanos, possuem os valores verdadeiro e falso, que podem ser representados de diferentes formas, como sim/não, 1/0, true/false (SOUZA, 2012).

Ainda podem haver outros modificadores de tipo, como signed, unsigned, long ou short. Sua escolha depende do tamanho desejado para adequar-se ao armazenamento de dados (CASAVELLA, 2019).

Definir a propriedade tipo, ou *type*, em uma linguagem, tem como principal objetivo evitar ou reduzir erros e atividades incoerentes e, assim, garantir um programa mais confiável e eficiente. Veremos a seguir como pode ser feita a classificação dos sistemas de tipos.

4.1. MONOMÓRFICOS

Em um sistema de tipos monomórfico, cada recurso da linguagem como variável, retorno de função, constante ou parâmetro deve ser declarado com um tipo específico. Ele exige que, para cada elemento, existam representação e operações distintas e pode ser encontrado em códigos como os de Pascal ou Modula-2. Entretanto, segundo o autor Leandro Fernandes Vieira (2015), em seu artigo Paradigmas de Programação: Uma Abordagem Comparativa, mesmo que essas linguagens sejam essencialmente monomórficas, ainda há um grau de sistema polimórfico nelas, como no operador “+”, que pode agir com números inteiros, reais ou até strings.

O professor Hermano Perrelli de Moura (2003), da Universidade Federal do Pernambuco, utiliza em seu material didático de Linguagens Computacionais a seguinte função para exemplificação:

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

Ela tem como tipo int x int → int e não poderia ser aplicada da mesma forma em argumentos de outros tipos como string.

4.2. POLIMÓRFICOS

Em um sistema de tipos polimórficos, um mesmo recurso, como funções e variáveis, pode funcionar para valores de tipos diferentes e agir uniformemente perante esses argumentos. Ele pode ser dividido entre duas classificações, sendo universal, quando a função atua, de forma uniforme, sobre infinitos tipos através de

uma única implementação, e ad-hoc, quando é preciso de mais de uma implementação para que a função possa agir com cada tipo de maneiras diferentes (DIAS, 2019).

Exemplos utilizados para o polimorfismo pelo professor Hermano Perrelli de Moura (2003), são os seguintes casos:

```
fun id (x) = x
```

Ela se trata de uma função Identidade e tem como tipo A → A, ou seja, apenas reproduz o tipo do argumento que recebeu e, assim, pode ser utilizada da mesma maneira para um inteiro, uma string, entre outros.

```
fun second (x, y) = y
```

Nesse caso, o tipo é T x S → S e, assim, não importando quais sejam os tipos dos argumentos fornecidos, o do segundo sempre será também o tipo do valor que a função vai retornar.

4.2.1. Polimorfismo ad-hoc

O polimorfismo *ad-hoc* (do latim, com sentido de “para este fim”) “permite que um valor polimórfico exibe comportamentos diferentes quando “visto” em tipos diferentes” (PIERCE, 2002, tradução nossa). As variações mais comumente encontradas deste tipo de polimorfismo são a sobrecarga de operadores e a coerção.

4.2.1.1. Overloading

O overloading, ou a sobrecarga de operadores, trata-se de um mesmo valor poder agir de acordo com qualquer tipo. Isso acontece quando ele recebe uma definição diferente para cada caso, ou seja, ele se refere a duas ou mais funções com o mesmo nome. O compilador decide qual delas vai ser usada dependendo do contexto encontrado. Um exemplo é o operador “+” no seguinte caso:

```
int a, b;
printf(a + b);
float x, y;
printf(x + y);
```

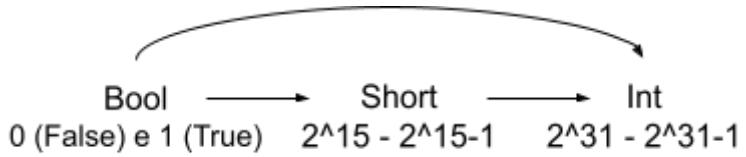
No código acima, o operador sobrecarregado trabalha com dois tipos diferentes, primeiro com números inteiros e depois com números reais (com ponto flutuante). De acordo com esses argumentos fornecidos, o compilador decide qual função do “+” deve agir e imprime o resultado correto.

4.2.1.2. Coerção

A coerção trata-se da conversão implícita de um valor a partir de um tipo para outro iniciada pelo compilador (SEBESTA, 2011). Isso acontece comumente em expressões aritméticas, como quando entre dois operandos, um possui tipo double e, por isso, o outro é convertido para a mesma forma, tendo como resultado mais um tipo double. Outro exemplo visto com frequência é o de números inteiros assumindo o papel de valores booleanos, ou lógicos, em que 0 representa False e 1 representa True.

O professor do Instituto Federal Catarinense, Ricardo de La Rocha Ladeira, em uma de suas aulas do ano de 2021, aprofunda a análise desse caso, seguindo para o fato de que, dessa forma, o tipo lógico se torna um subtipo contido no tipo inteiro, que representa valores entre 2^31 e 2^31-1. Isso também acontece com o tipo short que, abrangendo valores entre 2^15 e 2^15-1, se torna um subtipo dos inteiros e, consequentemente, o tipo lógico, com 0 e 1, torna-se um subtipo do tipo short.

Figura 3: Esquema do subtipo bool contido nos tipos short e int.



Fonte: Halena Kulmann Duarte

Dentro da coerção podem acontecer dois processos. O primeiro é chamado de ampliação, quando um certo tipo A recebe um valor de um dos seus subtipos B e a conversão entre eles é segura, pois é conhecido que o valor de B é do tipo A. No inverso da situação, o processo é diferente, chamado de estreitamento, em que o subtipo B recebe um valor de A, sendo necessária a verificação durante a execução do código, pois o valor de A pode não estar contido em B. Outra possibilidade é não

haver nem ampliação, nem estreitamento, como no caso um int ser convertido para unsigned, inteiros sem sinal.

4.3. TIPAGEM

4.3.1. Estática

Em uma linguagem estaticamente tipada, a checagem dos tipos é feita durante o processo de compilação. Eles são fixos para variáveis, parâmetros e outros, ou seja, não podem ser alterados depois de deduzidos ou declarados pelo desenvolvedor.

De acordo com o professor Alberto Costa Neto, da Universidade Federal do Sergipe, apesar de fazer com que a flexibilidade do programador seja reduzida, o método estático torna os programas mais eficientes, pois os erros são descobertos antes da execução. Um exemplo de código que apresenta erro na compilação, pois foi atribuída uma string a uma variável inteiro, é o seguinte:

```
int a = 22;
a = “22”;
```

A tipagem estática é utilizada na maioria das linguagens de alto nível e pode ser dividida em:

- **Explícita:** Uma sentença declarativa especifica o tipo da variável logo no momento de sua criação. Essa classificação pode ser encontrada em linguagens como: C, C++ e Java.

- **Implícita:** Na primeira vez em que a variável aparece, através de uma base padrão, ou *default*, é associado um tipo a ela baseado em uma convenção. Embora essa forma de declaração tenha sido praticamente abandonada, é possível achá-la presente na linguagem Fortran.

4.3.2. Dinâmica

Quando a linguagem apresenta uma tipagem dinâmica, não é exigido do programador a declaração do tipo dos dados, pois a vinculação é realizada em tempo de execução pela própria linguagem de forma dinâmica, assim como o próprio nome informa. Essa característica permite que uma mesma variável tenha o

seu tipo alterado infinitas vezes durante a execução do programa. Em virtude disso, é possível que uma variável receba um valor verdade, do tipo boolean, e logo na sequência uma cadeia de caracteres, representado por uma string.

Dentre as linguagens dinamicamente tipadas, destacam-se: Python, JavaScript e PHP. Como vantagem, elas apresentam uma maior flexibilidade para o programador, apesar de poderem tornar os programas menos eficientes, pois os erros demoram mais para serem descobertos (NETO, 2014).

4.3.3. Forte x Fraca

No tópico anterior, foram apresentados os conceitos de tipagem estática e dinâmica, que estão diretamente associados ao fato de uma linguagem ser compilada ou interpretada. Nesse momento, será definido o que são linguagens fortemente e fracamente tipadas, que está relacionado às operações permitidas entre dados de diferentes tipos. É importante frisar que não existe um consenso geral acerca dessa classificação, portanto, utilizaremos a classificação dada por Sebesta (2011).

Linguagens fortemente tipadas são aquelas que apresentam restrições à determinadas operações que violem as regras de um certo tipo. Um exemplo simples seria a subtração entres dois números inteiros. Suponha que seja subtraído o valor 3 do número 10. O resultado final esperado é simplesmente o valor 7. Isso acontece em virtude do tipo em questão aceitar esse tipo de operação. Porém, qual será o resultado caso se deseje somar uma string com um int? O resultado dessa operação seria algo totalmente irregular, e por conta disso, um erro é apresentado na saída do programa. Em razão disso, caracteriza-se como uma linguagem fortemente tipada aquela que, ao invés de retornar resultados anômalos, imprime um erro ao final. Exemplos de linguagens que apresentam essa característica são: Python e C++.

Por outro lado, existem as linguagens que apresentam uma tipagem fraca, que englobam todas aquelas que permitem as operações com tipos diferentes sem a apresentação de erros, como acontece em linguagens fortemente tipadas. Contudo, mesmo que essas operações sejam autorizadas, não existe necessariamente uma

definição de qual comportamento exatamente poderá ocorrer ao programa. Dentre as linguagens que são classificadas dessa forma, destacam-se: C e JavaScript.

4.4. INFERÊNCIA DE TIPO

Em linguagens de programação com tipagem estática, é permitido que, ao invés de o tipo de variáveis e parâmetros serem declarados explicitamente, eles sejam inferidos. A dedução do tipo do recurso é feita pelo compilador dependendo do contexto encontrado e, tratando-se de uma tipagem estática, não é possível alterar o que foi escolhido depois. Dentre as linguagens que utilizam a inferência de tipo, destacam-se: Haskell e C#, onde se utiliza a palavra reservada *var* para que o sistema de tipos deduza o tipo em questão.

4.5. EXERCÍCIOS

1 - O que é uma coerção? (SEBESTA, 2011)

2 - O que é um operador sobrecarregado? (SEBESTA, 2011)

3 - [POSCOMP 2016 - questão 28] – Assinale a alternativa que apresenta o nome de uma linguagem de tipagem dinâmica.

a) Java.
b) C.
c) Python.
d) Pascal.
e) C#.

4 - [POSCOMP 2018 - questão 38] – Sobre tipos de dados, é correto afirmar que:

a)Tipos booleanos são valores que são mantidos fixos pelo compilador.
b) O double é um tipo inteiro duplo com menor precisão do que o tipo inteiro.
c) A faixa de valores dos tipos inteiros tem somente dois elementos: um para verdadeiro e outro para falso.
d) Uma conversão de tipos implícita consiste em uma modificação do tipo de dados executado, automaticamente, pelo compilador.
e) Vetores, matrizes e ponteiros são exemplos de tipos de dados primitivos (básicos).

5 - [POSCOMP 2019 - questão 31] De acordo com a Teoria de Sistema de Tipos, classifique a função a seguir:

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

}

(a) Função Somadora.
(b) Função Polimórfica.
(c) Função Monomórfica.
(d) Função Sobrecarregada.
(e) Função Abstrata.

5. PARADIGMAS DE PROGRAMAÇÃO

5.1. CLASSIFICAÇÃO

Ao se elaborar um programa no meio da computação é ideal que seja pensado qual é a melhor maneira para fazê-lo e quais serão os recursos necessários para criá-lo de forma eficiente, e para isto pode ser feito o uso do conceito de paradigmas.

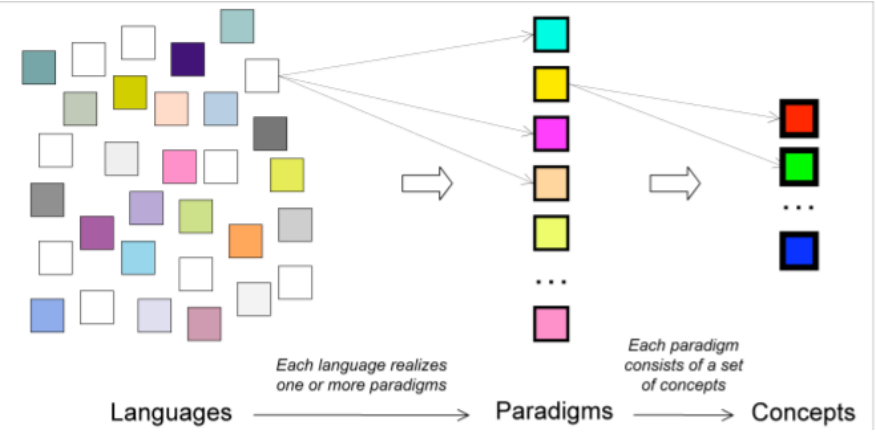
O paradigma da programação é um tipo de estruturação atribuída às linguagem conforme suas funcionalidades. É uma classificação com base em características específicas para resolver certos tipos de problemas, usando um conjunto de princípios (ROY, 2008). Pode ser também considerado como um jeito de pensar, levando em conta os recursos que as linguagens de programação possuem (LIERET, 2020).

Os paradigmas se dividem em dois grandes grupos: Paradigma Declarativo e Paradigma Imperativo, e dentro desses se tem muitas outras subdivisões. A grande diferença entre esses dois grupos é que no paradigma declarativo é característico que o programa siga de acordo com a lógica, fazendo o programador definir "o que" mostrar, já no Imperativo a ordem de funcionamento do programa deve seguir uma série de passos lógicos, definindo "como" mostrar o resultado (VIEIRA, 2015).

Uma linguagem de programação pode ter mais de um paradigma ligado a ela, como exemplificado na figura 4. Quando se tem mais de um, a linguagem pode ser chamada de linguagem multiparadigma.

Um exemplo de linguagem multiparadigma é o Python, pois este permite utilizar laços de repetição e condicionais (característico da programação imperativa) e pode-se adicionar funções lambda (típico da programação declarativa).

Figura 4: Funcionamento da lógica de linguagem, paradigmas e conceitos



Fonte:

http://www.sm.luth.se/csee/courses/timber/reading/VanRoy.pdf

Ao saber sobre os paradigmas, pode-se aprender novos jeitos de pensar em como fazer aplicações de maneira mais eficiente sabendo aproveitar os recursos que a linguagem de programação consegue proporcionar.

Por exemplo, caso um usuário queira fazer um programa de cálculos que utiliza muitas operações complexas é interessante pensar em utilizar alguma linguagem que possua o paradigma procedural, como por exemplo a linguagem C.

Se o programador quiser fazer um programa que utiliza entidades, como por exemplo uma aplicação *CRUD* (Create, Remove, Update, Delete) é interessante utilizar alguma linguagem que se enquadre no paradigma orientado a objetos, ou seja, que utilize classes, como por exemplo o Java.

5.2. EXERCÍCIOS

1 - O que é uma linguagem multiparadigma?

2 - Qual é a diferença entre o paradigma declarativo e o imperativo?

6. PARADIGMA IMPERATIVO

Considerado um dos primeiros a surgir, e, inclusive, o mais utilizado até hoje, o paradigma de programação imperativo se baseia na ordenação de um programa através de enunciados e ações que alteram o estado que o mesmo se encontra. Assim como na linguagem natural, “imperativo” está relacionado ao comportamento de expressar uma ordem que deverá ser realizada. Ou seja, no caso da computação, um programador expressa as ordens, que seria a sequência de comandos, a um computador, que as executará por meio da CPU.

Baseado na arquitetura de Von Neumann, já mencionada anteriormente, um programa que segue esse paradigma sempre irá apresentar uma sequência de instruções, e é justamente essa sequência que irá descrever o fluxo de controle do programa. Como sua construção se fundamenta basicamente em variáveis, atribuições e sequências, dados estes que são armazenados na memória, o paradigma imperativo acaba se destacando pela simplicidade de se programar, visto que, ele se apoia na ideia do “faça isso, depois aquilo!”.

Esse paradigma pode ser dividido em duas subcategorias, sendo elas:

- **Paradigma estruturado:** Conta com estruturas condicionais (*if* e *else*) e estruturas de repetição (*while* e *for*). A sequência e essas estruturas mencionadas são a base de um programa, e são elas que especificam o passo a passo para que o mesmo alcance o resultado aguardado. As linguagens que apresentam esse paradigma são: C, Java, Python e ALGOL.
- **Paradigma não estruturado:** Diferentemente do paradigma estruturado, este não conta com estruturas de repetição e condicionais. Em virtude disso, a programação é feita através do *jump*, ou popularmente conhecido como *goto*, recurso este que também está presente na programação estruturada. Por apresentar essa limitação, a forma de programar nesse paradigma é feita “saltando” de uma parte do programa para outra. Em razão disso, fatores como legibilidade e manutenibilidade acabam sendo comprometidos, consequentemente, deixando o código difícil de manter e encontrar erros. Essa característica de código é denominada “Código Espaguete”, e pode ser encontrada em Assembly e versões mais antigas de linguagens como: COBOL e Fortran.

6.1. PARADIGMA PROCEDURAL

O paradigma Procedural possui as três principais estruturas, a sequência, a repetição e a decisão. Tem como unidade básica as funções, que se resumem a um pequeno bloco de código coeso destinado a resolver um problema específico, como por exemplo somar dois números ou uma operação que não vá muito além disso, porém serve também para resolver um problema maior, mas não é recomendável para tal fim.

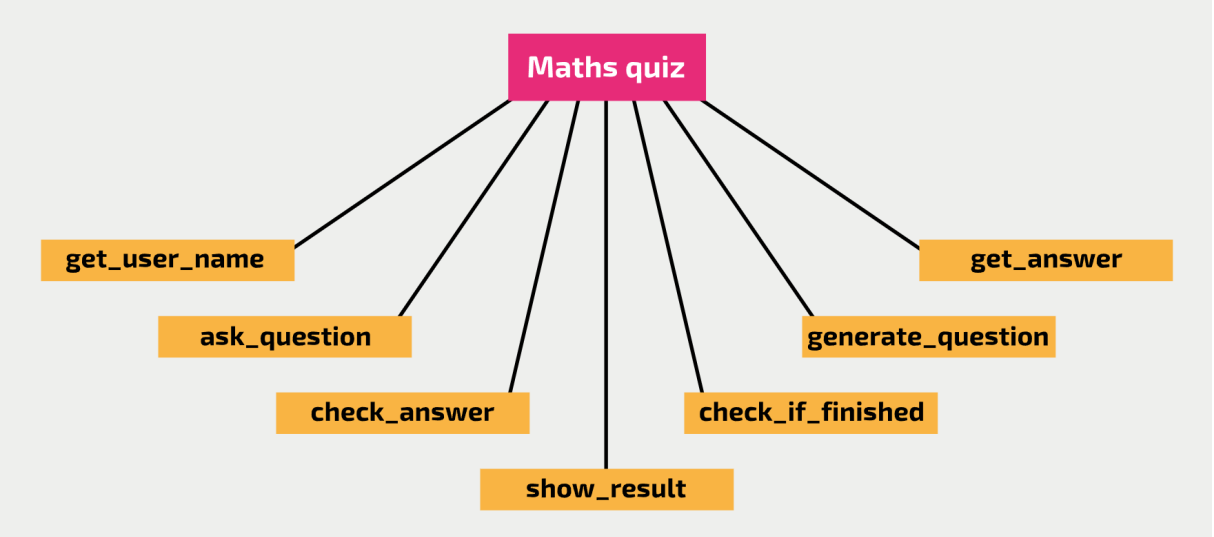
Para cada função existe a possibilidade de criar uma lista com parâmetros que ela irá receber, podendo até mesmo ser vazia, não tendo a obrigatoriedade de passar algum valor. Também é esperado algum tipo de retorno no final de uma função, sendo ele o resultado dos processos realizados dentro da função.

Após ser declarada uma vez, uma função pode ser usada continuamente até o fim do código, dentro de outras funções ou dentro dela mesma (essa última prática é conhecida como recursividade).

Adotar o paradigma procedural para resolver problemas através do código pode ter benefícios, já que quando acontece a fragmentação do problema em partes menores fica muito mais fácil e rápido para depurar. As maiores vantagens estão relacionadas ao que as funções podem proporcionar, como tornar o código reusável já que podem ser utilizadas em diversos arquivos diferentes quando salvas em bibliotecas e importadas nos programas.

Quando os nomes dados às funções são coerentes com o que elas são propostas a fazer, tornando assim muito mais legível e intuitivo para futuras manutenções do código.

Figura 5: Estrutura para um teste de matemática



Fonte: https://isaaccomputerscience.org/concepts/prog_pas_paradigm?examBoard=all&stage=all

A imagem acima exemplifica como pode ocorrer a divisão do que cada função será responsável, tornando assim muito mais fácil ler, depurar e corrigir erros no futuro do código.

São relatadas algumas desvantagens quanto ao uso do paradigma procedural são por exemplo, a dificuldade de relacionar com objetos do mundo real (para isso seria melhor usar o paradigma orientado a objeto, já que serve justamente para esse fim), além disso, em alguns casos em que os dados são sensíveis é necessário tomar cuidado, pois o foco acaba sendo na operação e não tanto no dado, além de estar exposto para todo o código, justamente por não ser encapsulado em um único lugar.

Para poder ser considerada procedural, a linguagem precisa suportar o conceito de funções e uma forma para defini-las, além de alguns tipos de argumentos, entre outros detalhes. Linguagens que atendem esses requisitos são por exemplo: ALGOL, C, Python, PHP entre diversas outras.

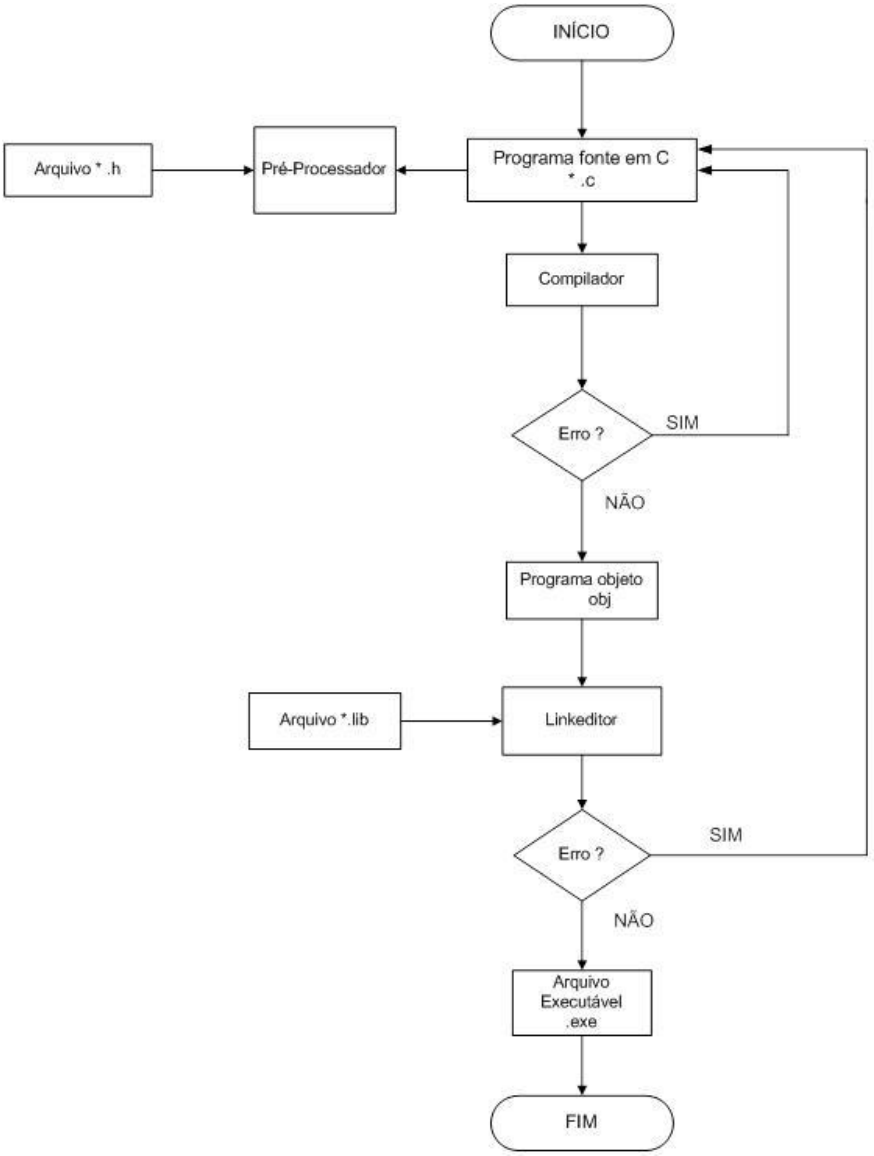
6.1.1 Linguagem C

A linguagem C foi criada em 1972 por Dennis Ritchie nos laboratórios da empresa Bell Telephone, com o intuito de ser usada para desenvolver a nova versão do sistema Unix, já que sua primeira versão era feita em Assembly. C seria uma

evolução da Linguagem B, que é uma adaptação feita a partir da linguagem BCPL por Ken Thompson.

C é uma linguagem de propósito geral, ou seja, é possível usá-la em diversos tipos de projetos, como por exemplo sistemas operacionais, ou programas que resolvam problemas físicos, químicos e matemáticos, pois tem elementos de baixo nível como a manipulação de bits, e também de alto nível como a tipagem de dados.

Figura 6: Diagrama de blocos do ciclo de desenvolvimento em C



Fonte:
http://linguagemc.com.br/desenvolvendo-um-programa-em-linguagem-c

Na figura 5 é possível verificar as etapas de desenvolvimento de um programa na linguagem C, comparada às linguagens atuais pode ter certa

desvantagem por ter que passar por todo o processo de compilação e etc, mas ainda tem seus pontos fortes.

Existe a fase onde começa tudo, que seria o desenvolvimento do código fonte, salvando o arquivo com a extensão .c, após isso ocorre o pré-processamento, que é feito antes da compilação do programa onde são executadas as diretivas. O processo de compilação é realizado em seguida, é onde é feita a verificação da sintaxe e caso esteja errada irá gerar um erro na compilação e então deve-se corrigir tal erro.

Por fim ocorre a linkedição do código, sabendo que C faz referências a funções que podem estar em outras bibliotecas, sendo elas privadas ou não, então é importante juntar todo esse código em um local só e criar o arquivo executável para assim finalmente poder rodar o programa, caso ocorra algum erro é necessário corrigir e passar por todo o processo novamente.

6.2. PARADIGMA ORIENTADO A OBJETOS

Na programação estruturada, existe uma massa de dados comuns e procedimentos que utilizam esses dados, utilizando apenas o necessário para o processo em si. O paradigma Orientado a Objetos por sua vez divide o projeto em objetos e métodos que manipulam os dados pertencentes ao seu determinado objeto, além de também poder existir a troca de dados entre eles.

Segundo Sebesta (2011), Smalltalk foi a primeira linguagem que oferecia suporte completo para a programação orientada a objetos, que teve suas raízes no SIMULA 67. Smalltalk utilizava de mensagens enviadas a um objeto, que retornava a informação ou a notificação requisitada pela mensagem, essas mensagens são enviadas para um dos métodos de um objeto, que ao ser executado, normalmente modifica os dados do objeto. O pioneirismo da Smalltalk, proporcionou uma base, junto com recursos do SIMULA 67 e da linguagem C para o desenvolvimento do C++ que a partir do melhorando recursos do C e permitindo suporte para a programação orientada a objetos recebendo assim muita atenção, pois na época era a única linguagem disponível com recursos para realizar grandes projetos comerciais.

A seguir os principais conceitos da programação orientada a objetos:

Classe: É como um modelo para o objeto, seus atributos e métodos. Que pode ser instanciada em um objeto. Por exemplo: Uma classe “veículo” pode possuir os atributos de nome, velocidade máxima, capacidade, modelo, entre outros que podem ser definidos. E também possuir o métodos de acelerar, frear, descarregar entre outros.

Objeto: Uma determinada classe instanciada com valores específicos para seus atributos. Por exemplo uma instância da classe veículo, com o nome de “Carro”, velocidade máxima: 200km/h, e capacidade: 4 pessoas.

Abstração: “Uma abstração é uma visão ou representação de uma entidade que inclui apenas os atributos mais significativos.” Sebesta(2011), assim somente o necessário ao restante do código é representado para a classe.

Método: Definido dentro das classes, funções que descrevem o comportamento de um objeto. Por exemplo, a classe Veículo possui o método acelerar, que aumenta a velocidade do objeto instanciado.

Atributos: Onde os objetos armazenam os dados. Por exemplo, o atributo nome na instância da classe veículo ser “Carro”.

Encapsulamento: Forma de manter os dados manipulados visíveis mantidos, sendo exposto apenas os dados selecionados. Impedindo assim o acesso externo ou de qualquer meio que não for autorizado aos dados da classe. Dessa forma o programa se mantém seguro e diminui as chances de falhas e invasões.

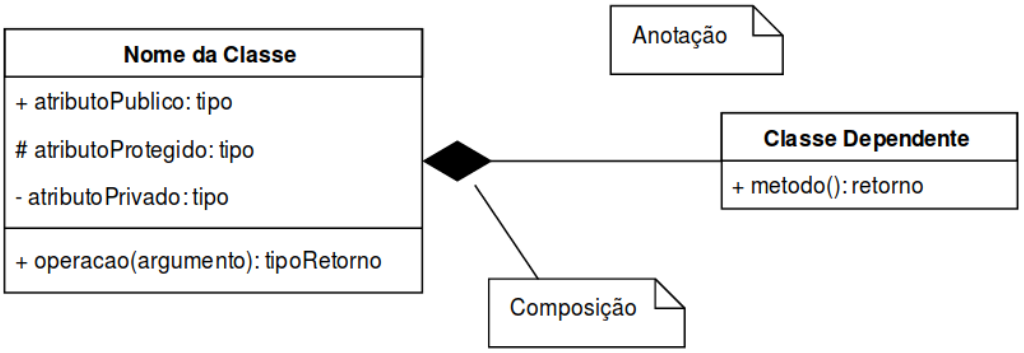
Herança: Na Herança, uma classe pode reutilizar a lógica de códigos de outra classe podendo herdar métodos e atributos em uma relação hierárquica. Criando assim uma dependência entre a classe pai e as classes que herdam dela, sendo considerado um ponto negativo. Um exemplo de herança pode ser a classe “carro” que herda métodos e atributos da classe “veículo”.

Polimorfismo: Permite que objetos adotem mais de uma forma dependendo do contexto de execução. Por exemplo, a classe veículo possui o método mover e é classe pai de carro e avião, como os veículos se movem de maneira diferente o

método “mover” também será diferente em cada classe filha, assim a execução do método “mover“ será diferente.

Diagrama de classes: Representação das relações entre as classes, usado para projetar o código de maneira organizada. Exemplo:

Figura 7: Diagrama de classes.



Fonte: https://pt.wikipedia.org/wiki/Diagrama_de_classes

6.2.1 Linguagem Java

Criada em 1995, Java é atualmente uma das linguagens de programação mais populares, sendo propriedade da Oracle é usada em diversas plataformas. O suporte para programação orientada a objetos em Java é similar ao de C++ e ao de C#.

Em Java, toda aplicação deve começar com uma classe que possui o mesmo nome do arquivo, no exemplo abaixo o nome do arquivo será “Main.java”:

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

Para a criação de classes, atributos e métodos é seguido a seguinte sintaxe no exemplo da criação da classe Veículo:

```
public class Veiculo{
        public String nome;                    // Visibilidade, Tipo, nome
        private int velocidademax;
```

```
        // Construtor
        public Veiculo(String nome, int velocidademax) {
                this.nome = nome;
                this.velocidademax = velocidademax;
        }
        // Métodos da classe
         public void mover(){
                System.out.println("movendo");
        }
}
```

Como na programação orientada a objeto a herança é uma parte importante, em Java ela acontece da seguinte forma: class Carro extends Veiculo . Dizendo que a classe carro herda as propriedades da classe veículo.

Para o teste das classes e a criação de objetos, é recomendado criar um novo arquivo com uma classe de Teste, como por exemplo a TestaVeiculo:

```
public class TestaVeiculo {
        public static void main(String[] args) {
                // Criar um objeto veiculo
                Veiculo v1 = new Veiculo ("Carro", 500);
                // Usar o método mover
                v1.mover();
        }
}
```

Existem diversas outras funcionalidades muito úteis em Java que permitem a criação de variados tipos de aplicação, um maior aprofundamento pode ser encontrado no material recomendado nas leituras complementares, visto que foi mantido um nível muito simples da linguagem neste tópico.

6.3. EXERCÍCIOS

1 - Crie por completo a classe veículo, adicione métodos e atributos que achar necessário e realize os conceitos de herança e polimorfismo.

2 - Java pode somente ser usado com o paradigma orientado a objetos?

3 - Crie um exemplo de código no paradigma procedural e tente reproduzi-lo em POO.

4 - Pesquise os tipos existentes de relacionamento entre classes.

Leitura complementar e links externos

- Curso da W3Schools sobre Java e POO: https://www.w3schools.com/java/
- Playlist de vídeos que comentam sobre Programação orientada a objeto: https://www.youtube.com/watch?v=KlIL63MeyMY&list=PLHz_AreHm4dkqe2aR0tQK74m8SFe-aGsY

7. PARADIGMA DECLARATIVO

O tópico anterior abordou o paradigma imperativo, cujas linguagens têm muita similaridade entre si por terem uma base em comum: a arquitetura de computadores de Von Neumann (SEBESTA, 2011). Sebesta (2011) comenta que as linguagens desse paradigma buscam usar essa arquitetura da forma mais eficiente possível, e que suas evoluções podem ser vistas como uma espécie de aperfeiçoamento do modelo básico, que era o Fortran I.

De forma geral, o programa consiste de um conjunto de dados que representam o seu estado e, após serem manipulados, tal estado é alterado e se chega na solução do problema (VIEIRA, 2015). Assim, o programador deve especificar todos os passos para se chegar em uma resolução, variando apenas os comandos e recursos disponíveis, o nível de abstração da linguagem e assim por diante. O foco está no “como” (MALAVASI, 2017).

Muitos programadores estão acostumados com a programação imperativa, tendo em vista que grande parte das linguagens mais populares atualmente se enquadram como tal, e raramente uma aplicação é feita sem utilizar esse paradigma (SEBESTA, 2011). Também segundo o autor, algumas pessoas consideram a dependência de arquitetura da programação imperativa desnecessária para a criação dos softwares, implicando no surgimento da programação declarativa.

A programação declarativa, foca no “que” deve ser computado ao invés de “como” deve ser computado (MALAVASI, 2017). Além disso, não há a ideia de estado de programa, o que implica que depois da declaração de algum comando e de suas interações seu resultado é imutável (VIEIRA, 2015).

Malavasi (2017) comenta que um bom exemplo da programação declarativa é a linguagem SQL: quando se faz uma consulta em um banco de dados com o comando “select”, pouco importa a forma como ele foi implementado, mas sim o resultado da consulta, os dados. Nota-se que o nível de abstração desse tipo de linguagem costuma ser comparativamente maior do que o paradigma imperativo.

Para esse estudo, vamos subdividir o paradigma declarativo em duas categorias: o paradigma lógico e o paradigma funcional, abordados a seguir.

7.1. PARADIGMA LÓGICO

O paradigma lógico, de acordo com Sebesta (2011), se baseia em uma notação lógica formal. Ele utiliza como base de dados, principalmente, fatos e regras. Um fato é uma sentença que é aceita como verdadeira, estabelecendo um relacionamento entre objetos. “Uruguai é um país”, por exemplo, é considerado um fato. Já as regras surgem de relacionamentos lógicos entre os fatos, ou seja, da composição deles. Por exemplo, a partir dos fatos “Lucas é pai de Pedro” e “Pedro é pai de Fernando”, podemos definir novas relações por meio da composição desses fatos, criando uma regra para avô e uma para neto. Concluindo, então, que “△Lucas é avô de Fernando “ e “△Fernando é neto de Lucas”. Um conjunto de fatos e regras é chamado de base de conhecimento. Tudo o que está na base de conhecimento é assumido como verdadeiro e a validade de novas proposições é verificada por meio dela (Sebesta, 2011).

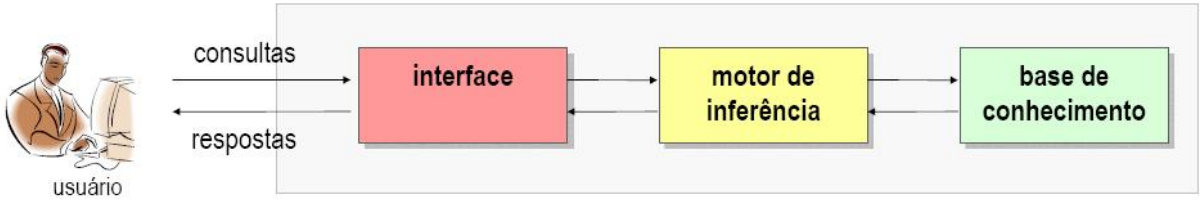
7.1.1 Linguagem Prolog

Segundo Sebesta (2011), a linguagem de programação lógica mais utilizada é Prolog, que significa *programming in logic*. Foi desenvolvida na década de 70 principalmente por Alain Colmerauer e Phillippe Roussel na Universidade de Aix-Marseille, com a ajuda de Robert Kowalski na Universidade de Edimburgo (Sebesta, 2011). Duas equipes começaram a trabalhar de forma independente a partir da especificação da linguagem, uma desenvolveu o interpretador e outra o compilador, sendo assim, Prolog é uma das primeiras linguagens criadas que possui versões interpretadas e compiladas desde o começo.

Com relação à sintaxe do Prolog, ela é bem rigorosa, especialmente com letras maiúsculas, minúsculas e o uso do ponto final, que deve ser colocado no fim de cada linha, sendo um fato, uma regra ou mesmo uma consulta. Isso ocorre por conta de que o Prolog considera letras minúsculas como um átomo e letras maiúsculas como variáveis (Sebesta, 2011). Segundo Sebesta (2011), um programa em Prolog é constituído por um conjunto de cláusulas, que são as declarações de fatos e regras. Essas cláusulas formam a base de conhecimento do programa. Então, com a base de conhecimento completa, o usuário realiza consultas por meio de uma interface. Em seguida, o Motor de Inferências é acessado pela interface e é o responsável por acessar a base de conhecimento, aplicando as regras aos fatos e

encerrando a execução quando encontra a solução ou quando não há regra a ser aplicada. Além disso, há casos em que a solução pode estar implícita. Por fim, a consulta é realizada e a resposta é retornada ao usuário (Sebesta, 2011).

Figura 8: Funcionamento da Linguagem Prolog



Fonte: https://docplayer.com.br/47248907-Programacao-logica-a-linguagem-prolog-paulo-henrique-ribeiro-gabriel-faculdade-de-computacao-universidade-federal-de-uberlandia.html

7.1.1.1 Fatos e Regras

"Uruguai é um país", como citado anteriormente, é um fato em linguagem natural. Esse fato em linguagem Prolog ficaria:

```
pais(uruguai).
```

Outros exemplos, como "Bill é um homem", "Fernanda é uma mulher", "Bill é pai de Fernanda", escritos em linguagem Prolog ficariam assim:

```
homem(bill).
mulher(fernanda).
pai(bill, fernanda).
```

No exemplo "pai(bill,fernanda)." "pai" é chamado de predicado e os seus parâmetros, ou seja, cada valor escrito em letras minúsculas que está entre parênteses é chamado de átomo. Neste caso, "bill" e "fernanda" são átomos. Lembrando que sempre deve-se utilizar o ponto final no fim de cada sentença. Segundo Sebesta (2011), uma função de predicado retorna verdadeiro ou falso, ou seja, o resultado da consulta será um valor booleano.

Já para a criação de regras, podemos utilizar variáveis. De acordo com Sebesta (2011), as variáveis são escritas iniciando com letra maiúscula e não são

vinculadas a tipos por declarações. Elas são vinculadas por um processo chamado de instanciação, que dura apenas até a consulta de uma proposição ser finalizada.

No exemplo abaixo temos uma regra, em que X e Y são as variáveis.

```
irmao(X, Y) :- pai(P, X), pai(P, Y), X\==Y, sexo(X, masculino).
```

A regra acima verifica se duas pessoas são irmãos. Segundo Sebesta (2011), o lado direito da regra, depois do ":-", é o antecedente e o lado esquerdo da regra, antes do ":-", é o consequente. Se o antecedente é verdadeiro, então o consequente também será. No exemplo, o antecedente dessa regra diz que se "P" é pai de "X", "P" é pai de "Y", "X" é diferente de "Y" e "X" é do sexo masculino, então o consequente será verdadeiro. Nesse caso, o consequente diz que "X" é irmão de "Y".

Conforme Sebesta (2011), o E lógico em Prolog é utilizado por meio de vírgulas, por exemplo:

```
pai(P, X), pai(P, Y)
```

Significa "P" é pai de "X" e "P" é pai de "Y". Caso no lugar da vírgula estivesse o ponto e vírgula (;) seria o OU lógico em Prolog, resultando em "P" é pai de "X" ou "P" é pai de "Y".

7.1.1.2 Exemplos de Consulta

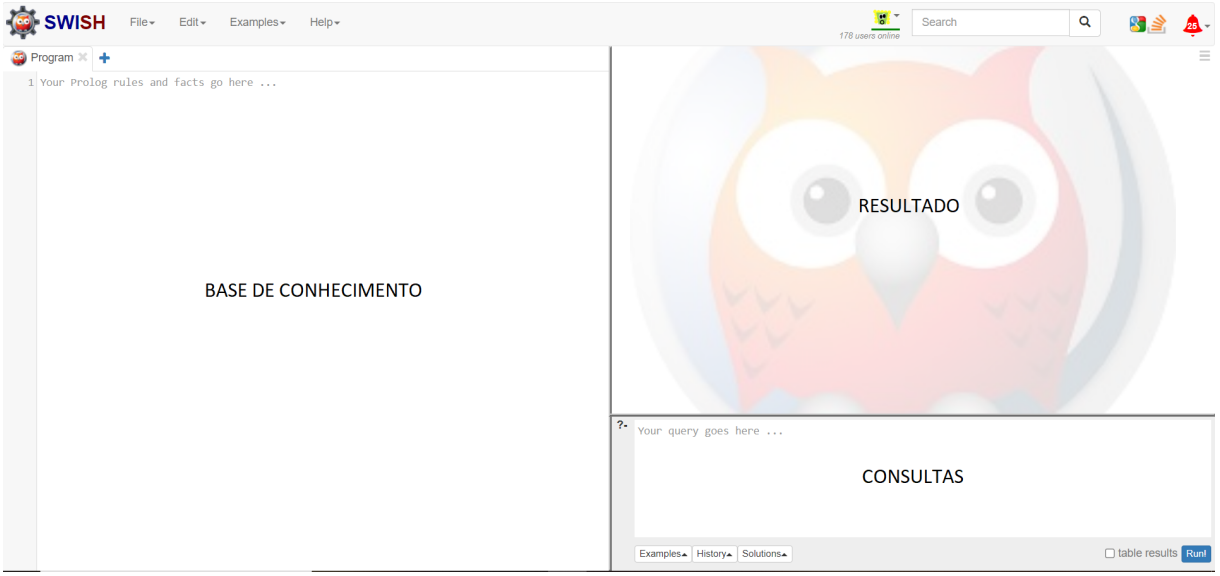
Para realizar uma consulta você pode optar pelo terminal ou por algum serviço online. Utilizando o terminal você deve ir até o diretório em que o arquivo com a extensão .pl esteja salvo. Em seguida, é necessário digitar a palavra "prolog" ou "swipl" para entrar no ambiente swipl. Então, basta carregar o arquivo desejado digitando entre colchetes o nome do arquivo e finalizar com o ponto final. Por exemplo, se seu arquivo se chama filmes.pl, você carregará ele da seguinte forma:

```
[filmes].
```

Caso escolha um serviço online para realizar a consulta basta acessar o link e começar a trabalhar. Vamos usar o site (https://swish.swi-prolog.org/) como exemplo. Ao entrar no site, você deve clicar em "Create a Program". Sua tela estará dividida

em 3 partes. A parte da base de conhecimentos, a área para fazer as consultas e por fim, o local onde será exibido o resultado das mesmas.

Figura 9: Site Swish, divisão da tela

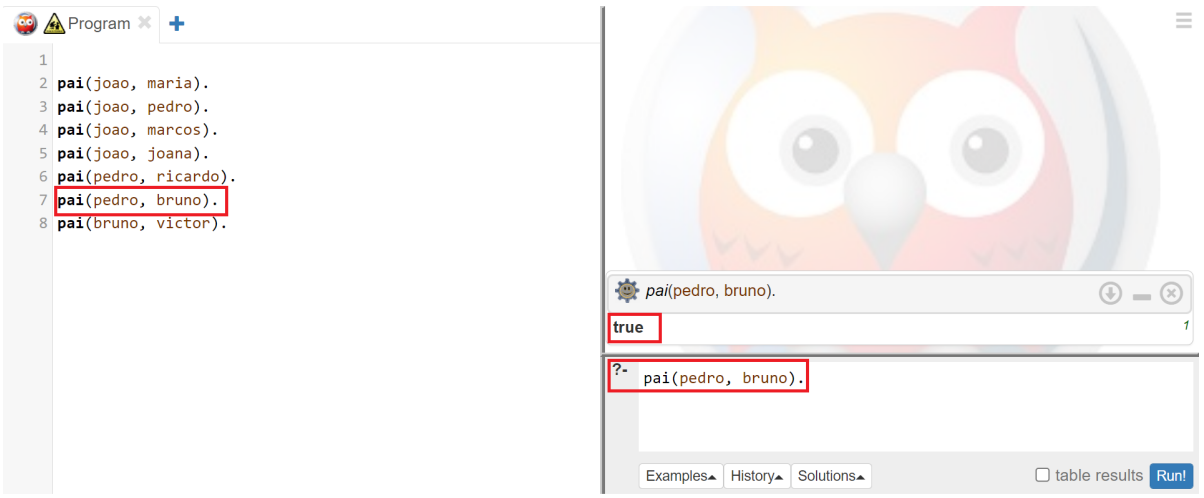


Fonte: Erick José Heiler

Na área da base de conhecimento ficará as regras e os fatos. A área de consultas serve para verificarmos se uma consulta é verdadeira a partir da base de conhecimentos. Por fim, o resultado será exibido na parte superior direita da tela.

Para realizar uma consulta é necessário iniciar com ?-, caso esteja utilizando o site do exemplo ele já estará incluso. Após digitar a consulta, é necessário pressionar Ctrl + enter para iniciar a pesquisa.

Figura 10: Prolog Exemplo 1 de Consulta

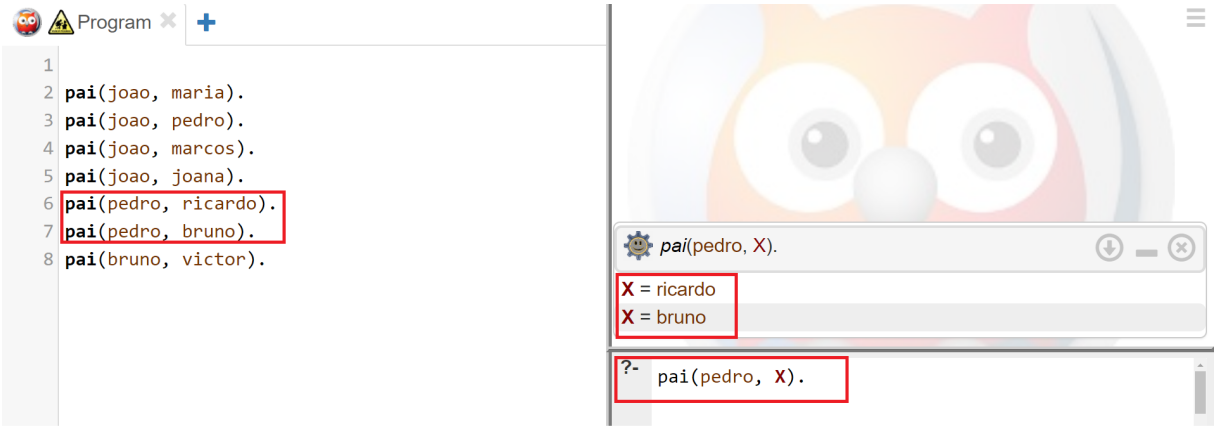


Fonte: Erick José Heiler

Na consulta acima ele está verificando se “pedro” é “pai” de “bruno”, retornando *true* na área dos resultados, pois essa informação está na nossa base de conhecimento.

Podemos também fazer uma consulta de variável, por exemplo, para saber de quem “pedro” é “pai”.

Figura 11: Prolog Exemplo 2 de Consulta



Fonte: Erick José Heiler

Na consulta acima será verificado se tem alguém de quem o “pedro” é “pai”. Se tiver, ele mostrará uma instância para qual “pai” é verdadeiro. No caso, a primeira instância é “X = ricardo”. Para pedir outras instâncias é necessário pressionar o ; (ponto e vírgula), caso esteja no terminal, ou clicando em *Next*, caso esteja no site do exemplo. Nessa consulta houve duas instâncias verdadeiras (“X = ricardo” e “X = bruno”).

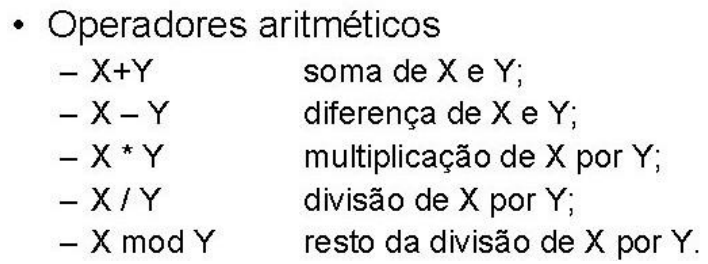
7.1.1.3 Operadores

Abaixo estão os operadores relacionais e aritméticos da linguagem Prolog.

Figura 12: Operadores Relacionais Prolog

Fonte: https://slidetodoc.com/introduo-linguagem-prolog-prof-fabrcio-enembreck-ppgia-programa/

Figura 13: Operadores Aritméticos Prolog



Fonte: https://slidetodoc.com/introduo-linguagem-prolog-prof-fabrcio-enembreck-ppgia-programa/

7.1.1.4 Desvantagens

Segundo Sebesta (2011), a linguagem Prolog possui algumas desvantagens, entre elas está o controle da ordem de resolução. No Prolog o usuário pode ordenar as sentenças da base de conhecimento para otimizar a execução, modificando o fluxo de controle. Isso pode acarretar em problemas na eficiência do programa, se não for feito de forma correta, já que algumas regras podem ser mais propensas a serem bem-sucedidas do que outras. Além disso, certas vezes, laços de repetição infinitos são gerados. Por conta disso, é muito importante que o usuário ordene as sentenças da base de conhecimento do programa de forma a não causar esse tipo de problema e otimizar a sua execução.

Outra desvantagem do Prolog, conforme Sebesta (2011), é a premissa do mundo fechado. Isso significa que o que não pode ser comprovado como verdadeiro

a partir da base de conhecimento do programa é assumido como falso. Caso a base de conhecimento esteja incompleta, resultados errados podem ser gerados.

Há também, segundo Sebesta (2011), o problema da negação. Pois, em Prolog, em determinadas regras é necessário especificar que o primeiro átomo deve ser diferente do segundo. Por exemplo na regra abaixo:

irmao(X, Y) :- pai(P, X), pai(P, Y), X\==Y, sexo(X, masculino).

É necessário especificar que “X” deve ser diferente de “Y”, caso contrário o programa irá considerar que “X” é irmão dele mesmo. Isso acaba tornando a base de conhecimento muito grande com o tempo, pois em determinados casos é necessário especificar muitos fatos dizendo que cada par de átomos não pode ser igual.

Por fim, Sebesta (2011) diz que um dos objetivos da programação lógica é fornecer uma programação não procedural, em que o programador especifica o que o programa deve fazer e não como isso deve ser feito. O problema é que isso pode acarretar em um processo muito lento, já que as vezes a forma como o programa lida com determinadas situações pode não ser a maneira mais eficiente de realizar tal procedimento.

7.1.1.5 Principais Usos do Paradigma Lógico

O paradigma lógico é muito utilizado, conforme aponta Sebesta (2011), em SGBDs (Sistemas de Gerenciamento de Banco de Dados) para consultas de dados na forma de tabelas, com o usuário descrevendo as características da resposta ao invés de como obter a resposta. Também é utilizado em sistemas especialistas, com o uso do sistema APES, que possui um recurso para obter informações do usuário para a construção de um sistema especialista. Há também alguns tipos de processamento de linguagem natural que são feitos com programação lógica, como por exemplo base de dados inteligentes. Além disso, é muito utilizado para aplicações de inteligência artificial, como a robótica. Por fim, também é utilizado na demonstração de teoremas e na construção de compiladores.

7.1.1.6 Exercícios

1) Por que a base de conhecimento de um programa em Prolog deve ser o mais completa possível?

2) Qual é a diferença de antecedente e consequente em Prolog?

3) “Leandro é irmão de Rafael” é um fato em linguagem natural, escreva-o em linguagem Prolog.

4) prima(laura, bruna). é um fato em linguagem Prolog, escreva-o em linguagem natural.

5) Explique o funcionamento da regra a seguir e indique o antecedente e o consequente da mesma:

```
neta(X, Y) :- pai(Y, Z), pai(Z, X), sexo(X, feminino)
```

6) Cite 3 aplicações do paradigma lógico.

7) Cite 2 desvantagens do paradigma lógico. Explique porque é considerado uma desvantagem.

7.2. PARADIGMA FUNCIONAL

A programação funcional é um paradigma de programação que descreve uma computação como uma expressão a ser avaliada. Também é um paradigma declarativo. É baseado em funções matemáticas e originário do cálculo Lambda.

De acordo com SEBESTA (2011) às linguagens de programação puramente funcionais não usam variáveis nem sentenças de atribuição. Dessa forma o programador não precisa se preocupar com a alocação de memória, já que essa operação ocorre automaticamente. Não é possível construir uma operação de iteração sem variáveis , já que estas são controladas por elas. Sem as variáveis a execução de um programa funcional não tem estado no sentido de semântica operacional e denotacional. Os programas são definidos por funções e especificações de aplicações de funções. As funções sempre retornam os mesmos resultados se forem fornecidos os mesmos parâmetros. Se f(x,y) = 10 na primeira execução da função, então o resultado será sempre 10. Este recurso é chamado de

Transparência referencial, tornando a semântica das linguagens puramente funcionais muito mais simples que as imperativas.

Segundo SEBESTA (2011), a programação funcional é utilizada no ensino de matemática, sistemas financeiros, bancários e sistemas embarcados. Alguns exemplos de linguagens funcionais são LISP (1ª), Haskell, ML, Miranda, Clojure, F#, Scheme e Elixir. Atualmente muitas linguagens suportam recursos de programação funcional, como JavaScript, Python, Ruby e C#.

7.2.1 Linguagem Haskell

De acordo com DU BOIS (2022) a linguagem de programação Haskell foi criada em 1990. É uma linguagem fortemente tipificada de maneira estática e suas funções podem ser polimórficas, além disso são usadas semânticas não estritas em Haskell. Infere tipos quando estes não são informados. Possui tipos e funções recursivas. A ideia principal da linguagem é baseada na avaliação de expressões. A implementação da linguagem avalia (simplifica) a expressão passada pelo programador até sua forma normal.

7.2.2 Operadores

- Relacionais: <, >, <=, >=, ==, /=
- Matemáticos: /, +, *, -, **, ^, sqrt
- Lógicos: &&, ||, not. Exemplos:

```
2 > 1 && 1 < 2
1 > 30 || 30 > 29
not(mod 4 3 == 0)
```

7.2.3 Sistema de Tipos

Alguns dos tipos suportados por Haskell são:

- Bool (True, False)

- Char

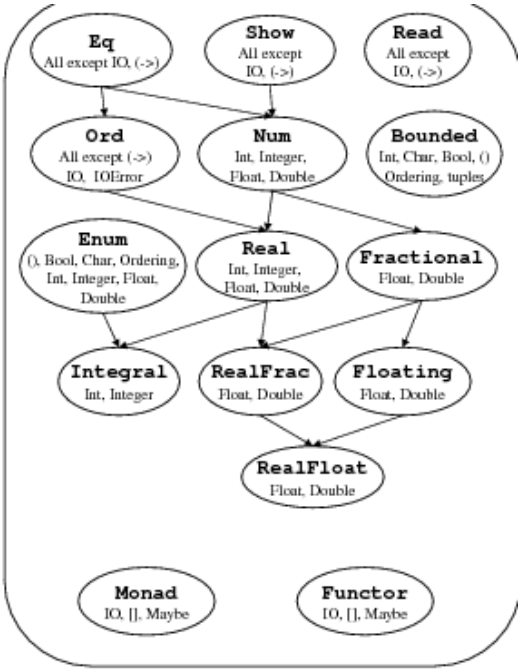
- Int

- Integer

- Double

- Função (sim, em Haskell uma função pode ser um tipo abstrato)

Figura 14: Hierarquia de Classes Haskell



Fonte: https://drive.google.com/drive/folders/1Mkcc87m6QgrzGNdcU1KrdQreYT9xEkhI

7.2.4 Funções

Segundo DU BOIS (2022), grande parte das funções em Haskell serão recursivas. Principalmente as que necessitam de algum tipo de repetição. Um clássico exemplo seria o fatorial de um número inteiro positivo.

Figura 15: Exemplo fatorial de um número

O fatorial de um número inteiro positivo pode ser dividido em dois casos:

- O fatorial de 0 será sempre 1;
- E o fatorial de um número n>0, será 1 * 2 *...* (n-1) * n

Então:

```
fatorial :: Int -> Int
fatorial 0      = 1                          (regra 1)
fatorial n      = n * fatorial (n-1)         (regra 2)
```

Exemplo de avaliação:

```
fatorial 3
= 3 * (fatorial 2)                           (2)
= 3 * 2 * (fatorial 1)                       (2)
= 3 * 2 * 1 * (fatorial 0)                   (2)
= 3 * 2 * 1 * 1                        (1)
= 6                                    Multiplicação
```

Fonte:https://www.inf.ufpr.br/andrey/ci062/ProgramacaoHaskell.pdf

Podem receber outras funções como parâmetro, podendo também retornar uma função. Funções como essas são chamadas de funções de alta ordem ou superiores.

Quando uma função possui parâmetros e resultados que não são funções ela é dita de primeira ordem.

7.2.5 Criando uma Função

A função abaixo soma três número inteiros,

```
somaTres :: Int -> Int -> Int -> Int -- esta linha não é obrigatória
-- somaTres :: Integer -> Integer -> Integer -> Integer
somaTres x y z = x + y + z
```

- Salve o arquivo como exemplo1.hs

- A partir do diretório em que exemplo1.hs foi salvo, abra o terminal e execute o ghci

- Digite:

> :l exemplo1.hs

-- O comando acima irá carregar o código do arquivo exemplo1.hs. Se estiver tudo OK, execute a função!

> somaTres 1 3 4

--O resultado será 8

Fonte: Exemplo retirado do Material Didático do Professor Ricardo Ladeira

7.2.6 Agregação de vários componentes (DU BOIS, s.d.)

Agregação de vários componentes ou uma sequência de elementos que podem ser de tipos variados.É possível em Haskell agrupar ou agregar elementos de tipos diferentes, conforme o exemplo abaixo.

```
(5, 8)
(3, 6* 3)
(1777988, "Julio")
(3 > 3, 5 + 3, True, 70.0) --
```

7.2.7 Avaliação Preguiçosa (lazy evaluation)

De acordo com DU BOIS (2022) este recurso está presente nas linguagens de programação funcionais puras. Significa dizer que os argumentos das funções são avaliados somente quando necessário. Por exemplo:

```
f(x) = 20
f(-2+21^2/20-5/100) = 20
```

Como essa função sempre retorna o mesmo resultado, não é necessário avaliá-la. A avaliação preguiçosa auxilia no trabalho com estruturas infinitas.Uma das principais estruturas infinitas são as listas. Segue exemplo:

Figura 16: Exemplo lista infinita

Um exemplo simples de lista infinita seria

```
uns = 1 : uns
```

Se passarmos esta estrutura para um interpretador:

```
Haskell > uns
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ^C {Interrupted}
```

Fonte:https://www.inf.ufpr.br/andrey/ci062/ProgramacaoHaskell.pdf

Esta função levaria a uma lista infinita até que uma tecla de interrupção fosse usada.

7.2.8 Condicionais

De acordo com DU BOIS (2022) a linguagem de programação Haskell permite utilizar a estrutura condicional com if e guardas:

```
if ExpressãoBool then Expressao1 else Expressao2
guardas
        | ExpressaoBool1 = Expressao1
        | ExpressaoBool2 = Expressao2
        | ExpressaoBool3 = Expressao3
        | ...
        | ExpressaoBooln = Expressaon -- esta última pode ser substituída por otherwise (caso contrário)
```

Segue exemplo abaixo desenvolvido pelo professor Ricardo Ladeira com base em DU BOIS.

- Em um arquivo .hs, acrescente a definição da função "sinal":

```
sinal x = if x < 0 then -1 else if x > 0 then 1 else 0
```

- Abra o terminal no mesmo diretório em que o arquivo está e abra o GHCi.

- Carregue o arquivo com :l nomedoarquivo.hs e execute a função das três formas possíveis:

```
> sinal 9
```

> sinal 0

> sinal (-9) -- o valor precisa estar entre parênteses

- Outra implementação válida para a mesma função, mas utilizando guardas:

```
sinal2 x

    | x > 0 = 1
    | x < 0 = -1
    | otherwise = 0
```

Fonte: Exemplo retirado do Material Didático do Professor Ricardo Ladeira

7.2.9 Let

De acordo com DU BOIS (2022), let (seja) é uma palavra reservada na linguagem Haskell - e em outras linguagens funcionais - para definir declarações aninhadas. A expressão tem escopo definido à direita da declaração, isto é, o valor do parâmetro será utilizado somente nas expressões ali definidas.

Exemplos:

```
> let x = 4 in x + 9
> let x = (-8) in sinal x
> let a = 5; b = 171 in mdc a b
> let n = 5 in fatorial1 n
```

Fonte: Exemplo retirado do Material Didático do Professor Ricardo Ladeira

7.2.10 Listas

De acordo com DU BOIS (2022), em Haskell é possível trabalhar com listas de vários tipos diferentes. Uma lista t, com elementos do tipo t.

Exemplos:

```
> let x = 4 in x + 9
> let x = (-8) in sinal x
> let a = 5; b = 171 in mdc a b
> let n = 5 in fatorial1 n
```

Fonte: Exemplo retirado do Material Didático do Professor Ricardo Ladeira

Pode-se trabalhar também com listas de listas, listas de tuplas e listas de funções (desde que as funções tenham o mesmo tipo)

Figura 17: Exemplo listas dentro de lista

```
[[1,2,3], [2,3], [3,4,5,6]]        :: [[Int]]
[(1,'a'), (2, 'b') , (3, 'c')]     :: [(Int, Char)]
[(/), (**), logBase]               :: [Float -> Float -> Float]
```

Fonte:https://www.inf.ufpr.br/andrey/ci062/ProgramacaoHaskell.pdf

Um outro caso são as listas vazias, [], que não possuem elementos e podem ser de qualquer tipo:

Figura 18 : Exemplo listas dentro de lista

```
[]        :: [Bool]
[]        :: [Float]
[]        :: [Int -> Int]
```

Fonte:https://www.inf.ufpr.br/andrey/ci062/ProgramacaoHaskell.pdf

A ordem e o número de ocorrência dos elementos é significante. Uma lista [3,4] é diferente de uma lista [4,3], e uma lista [1] é diferente de uma lista [1,1].

Uma lista pode ser criada também por intervalos.Exemplo:

```
>[-10..10]
>[2,4..20]
>[1,3..20]
>[100,10..]
```

Fonte: Exemplo retirado do Material Didático do Professor Ricardo Ladeira

Segundo DU BOIS (2022) a Compreensão de Lista é uma maneira de se descrever uma lista inspirada na notação de conjuntos. Por exemplo, se a lista list é [1, 7, 3], pode-se duplicar o valor dos elementos desta lista da seguinte maneira:

Figura 19: Exemplo listas dentro de lista

[2 * a | a < - list]

que terá valor:

[2, 14, 6]

ou

Haskell > [2* a | a<- [1, 7, 3]]
[2, 14, 6]

Fonte:https://www.inf.ufpr.br/andrey/ci062/ProgramacaoHaskell.pdf

7.2.11 Ferramentas

GHC, GHCi, Hugs, repl.it (https://repl.it/languages/haskell), www.tryhaskell.org e outras.

7.2.12 Desvantagens

O custo da recursão, ainda há poucos programadores de haskell e é grande a dificuldade em prever os custos de tempo e memória em programas com avaliação preguiçosa.

7.2.13 Exercícios

POSCOMP 2012 - Exercício 32

32 **Em linguagens de programação declarativas, em especial aquelas que seguem o paradigma funcional, a lista é uma estrutura de dados fundamental. Uma lista representa coleções de objetos de um único tipo, sendo composta por dois elementos: a cabeça (*head*) e o corpo (*tail*), exceto quando está vazia. A cabeça é sempre o primeiro elemento e o corpo é uma lista com os elementos da lista original, excetuando-se o primeiro elemento. O programa Haskell, a seguir, apresenta uma função que utiliza essa estrutura de dados.**

```
poscomp :: [Int] -> [Int]
poscomp [] = []
poscomp [x] = [x]
poscomp (a:b:c) | a > b = b : (a : poscomp c)
                | otherwise = a : (b : poscomp c)
```

Uma chamada a esta função através da consulta

```
poscomp [5,3,4,5,2,1,2,3,4]
```

produzirá o resultado:

a) [1,2,2,3,3,4,4,5,5]

b) [3,5,4,5,1,2,2,3,4]

c) [5,3,4,5,2,1,2,3,4]

d) [5,4,3,2,1]

e) [5,3,4,2,1]

REFERÊNCIAS

CASAVELLA, Eduardo. **Tipos de dados em C**. Intellectuale, 2019. Disponível em: <http://linguagemc.com.br/tipos-de-dados-em-c/>. Acesso em: 19 jan. 2022.

DEEPSOURCE. **Procedural Programming.** Disponível em: <https://deepsource.io/glossary/procedural-programming/>. Acesso em: 18 jan. 2022.

DIAS, Artur Miguel. **Tipos**. 2019. Material teórico. Disponível em: <http://ctp.di.fct.unl.pt/~amd/lap/teoricas/25.html>. Acesso em: 18 jan. 2022.

DOYLE, Kerry. **A breakdown of object-oriented programming concepts** Disponível em: <https://searchapparchitecture.techtarget.com/tip/A-breakdown-of-object-oriented-programming-concepts>. Acesso em: 19 jan. 2022.

DU BOIS, ANDRÉ RAUBER. **PROGRAMAÇÃO FUNCIONAL COM A LINGUAGEM HASKELL**. SD. Material didático. Disponível em: <https://www.inf.ufpr.br/andrey/ci062/ProgramacaoHaskell.pdf>. Acesso em: 19 jan. 2022.

FONSECA, Elton. **Quais as diferenças entre tipagens**: estática ou dinâmica e forte ou fraca. TreinaWeb. 2019. Disponível em: <https://www.treinaweb.com.br/blog/quais-as-diferencas-entre-tipagens-estatica-ou-dinamica-e-forte-ou-fraca>. Acesso em: 13 jan. 2022.

FRONTIER, Paul J.; MICHEL, Howard E. **Computer Systems Performance Evaluation and Prediction**. Digital Press, 2003.

HACKR.IO. **Procedural Programming [Definition].** Disponível em: <https://hackr.io/blog/procedural-programming>. Acesso em: 18 jan. 2022.

ISAACCOMPUTERSCIENCE. **The procedural paradigm.** Disponível em: <https://isaaccomputerscience.org/concepts/prog_pas_paradigm?examBoard=all&stage=all>. Acesso em: 18 jan. 2022.

MARTINS, Luiz Gustavo Almeida. **Apostila de Linguagem C (Conceitos Básicos).** Disponível em: <http://www.facom.ufu.br/~gustavo/ED1/Apostila_Linguagem_C>. Acesso em: 18 jan. 2022.

LADEIRA, Ricardo de La Rocha. **Linguagens de programação**. 2021. Material didático. Disponível em: <https://drive.google.com/drive/folders/1orLQscuv3RkWioj914S6bghjc8YfQtvB>. Acesso em: 18 jan. 2022.

LADEIRA, Ricardo de La Rocha. **Linguagens de programação**. 2021. Material didático. Disponível em: <https://drive.google.com/drive/folders/1bs-9gmpLCT7ilrTKHWTFrH-GcCtdkuDf>. Acesso em: 18 jan. 2022.

LADEIRA, Ricardo de La Rocha. **Paradigma Lógico**. 2021. Material didático. Disponível em: <https://drive.google.com/drive/folders/1ltKVfeEBYVNsIaDB-ysRMsh6TbFrXe0S>. Acesso em: 17 jan. 2022.

LADEIRA, Ricardo de La Rocha. **Sistemas de Tipos**. 2021. Material didático. Disponível em: <https://drive.google.com/drive/folders/1HVxaCNRzG2PHAIg-JdnyPU55mql8ORUN>. Acesso em: 10 jan. 2022.

LADEIRA, Ricardo de La Rocha. **Paradigma funcional**. 2021. Material didático. Disponível em: <https://drive.google.com/drive/folders/1Mkcc87m6QgrzGNdcU1KrdQreYT9xEkhI>. Acesso em: 19 jan. 2022.

LIERET, Kilian. **Programming Paradigms**. Disponível em: <https://indico.cern.ch/event/853710/contributions/3708306/attachments/1985126/3307454/programming_paradigms_280920_handout.pdf>. Acesso em 19 jan. 2022.

SOCIEDADE BRASILEIRA DE COMPUTAÇÃO. **Exame POSCOMP 2012**. 2012. Disponível em: <https://www.cops.uel.br/v2/download.php?Acesso=YzlmNzU2YTBiMWIzYTM4MDZiM2RmN2FiYWEzZDdkMWFhNjBhM2M3ZTZiMDc2ZDQ1MzZmYjgxOTM4YTAxNGRl

MWMyMTBiYjNiZDRjZGFlMThkZjg5NzkyODE3ZDIzNWNiY2IwYmIzNjMxYTczMmZkMzM0MTg5YThhY2RlNTY4MjdlMjc5ZDIzMjM3NTY1MjQ0ZWQ4YTQyYWQxMDJmODM0ZQ==>. Acesso em: 19 jan. 2022.

MALAVASI, Alexandre. **Descomplicando**: programação imperativa, declarativa e reativa. programação imperativa, declarativa e reativa. 2017. Disponível em: <https://medium.com/@alexandre.malavasi/descomplicando-programa%C3%A7%C3%A3o-imperativa-declarativa-e-reativa-a481baa87742>. Acesso em: 18 jan. 2022.

MEDINA, Marco; FERTING, Cristina. **Algoritmos e programação: teoria e prática**. Novatec Editora, 2006.

MOURA, Hermano Perreli de. **Sistemas de Tipo**. 2003. Material didático. Disponível em: <https://www.cin.ufpe.br/~if686/aulas/tipos.ppt>. Acesso em: 10 jan. 2022.

NETO, Alberto Costa. **Sistemas de Tipos**. 2014. Material didático. Disponível em: <http://albertocn.sytes.net/2014-1/plp/slides/05-SistemasDeTipos.pdf>. Acesso em: 12 jan. 2022.

OLIVETE JÚNIOR, Celso. **Linguagens de Programação**: aula 2. Color. Disponível em: <http://www2.fct.unesp.br/docentes/dmec/olivete/lp/arquivos/Aula2.pdf>. Acesso em: 4 jan. 2022.

PIERCE, Benjamin C. **Types and Programming Languages.** The MIT Press, 2002.

ROY, Peter Van. **Programming Paradigms for Dummies**: What Every Programmer Should Know. 2010. Disponível em: <http://www.sm.luth.se/csee/courses/timber/reading/VanRoy.pdf>. Acesso em: 16 Jan. 2022.

SAMPAIO, Augusto; MARANHÃO, Antônio. **Conceitos e Paradigmas de Programação via Projeto de Interpretadores**. 2008. Disponível em: <https://www.cin.ufpe.br/~in1007/transparencias/jai/Jai2008Augusto.pdf>. Acesso em: 16 Jan. 2022.

SEBESTA, Robert W. **Conceitos de Linguagens de Programação**. Bookman Editora, 2011.

SOCIEDADE BRASILEIRA DE COMPUTAÇÃO. **Exame POSCOMP 2016**. 2016. Disponível em: <http://publicacoes.fundatec.com.br/portal/concursos/394/0394.001.pdf?idpub=472644>. Acesso em: 17 jan. 2022.

SOCIEDADE BRASILEIRA DE COMPUTAÇÃO. **Exame POSCOMP 2018**. 2018. Disponível em: <https://www.sbc.org.br/documentos-da-sbc/send/202-2018/1203-prova-2018>. Acesso em: 17 jan. 2022.

SOCIEDADE BRASILEIRA DE COMPUTAÇÃO. **Exame POSCOMP 2019**. 2019. Disponível em: <https://www.sbc.org.br/documentos-da-sbc/send/212-2019/1246-prova-2019>. Acesso em: 17 jan. 2022.

SOUZA, Alessandro J. de Souza. **Variáveis e Tipos de Dados.** 2012. Material didático. Disponível em: <https://docente.ifrn.edu.br/alessandrosouza/disciplinas/algoritmo/downloads/aula-3-variaveis-e-tipo-de-dados>. Acesso em: 19 jan. 2022.

STERLING, Thomas; ANDERSON, Matthew; BRODOWICZ, Maciej. **High Performance Computing**. Morgan Kaufmann Publishers, 2018.

STRINGFIXER. **Sistema de Tipo**. Disponível em: <https://stringfixer.com/pt/Dynamic_typing>. Acesso em: 12 jan. 2022.

TANENBAUM, Andrew S.; ZUCCHI, Wagner Luiz. **Organização estruturada de computadores**. Pearson Prentice Hall, 2009.

VIEIRA, Leandro Fernandes. **PARADIGMAS DE PROGRAMAÇÃO**: uma abordagem comparativa. Uma Abordagem Comparativa. 2015. Disponível em: <https://leandromoh.gitbooks.io/tcc-paradigmas-de-programacao/content/2_paradigmas_imperativo_e_declarativo/22_paradigma_declarativo.html>. Acesso em: 16 jan. 2022.

W3Schools. **Java Tutorial**. Disponível em: <https://www.w3schools.com/java/> . Acesso em: 18 jan. 2022.