

Paradigma Funcional

Universidade Vila Velha
Linguagens de Programação - 2024/1

O que é o Paradigma Funcional?

O que é o Paradigma Funcional?

Tópicos:

1. Definição e conceitos
2. Funções
 - a. Composição de funções
 - b. Funções puras
 - c. Funções de primeira classe
 - d. Funções de alta ordem
3. Imutabilidade
4. Transparência referencial
5. Recursão
6. Lazy Evaluation
7. Paradigma funcional puro
8. Linguagens
 - a. Família Lisp
 - b. Outras

Definição e conceitos

Podemos definir o paradigma funcional como o modo de pensar em computação que tem como interesse a obtenção de respostas de perguntas computáveis **por meio de funções**.

O programa não está interessado em gerenciar o estado do sistema para entregar soluções.

O programa está interessado em demonstrar respostas corretas pela avaliação de funções.

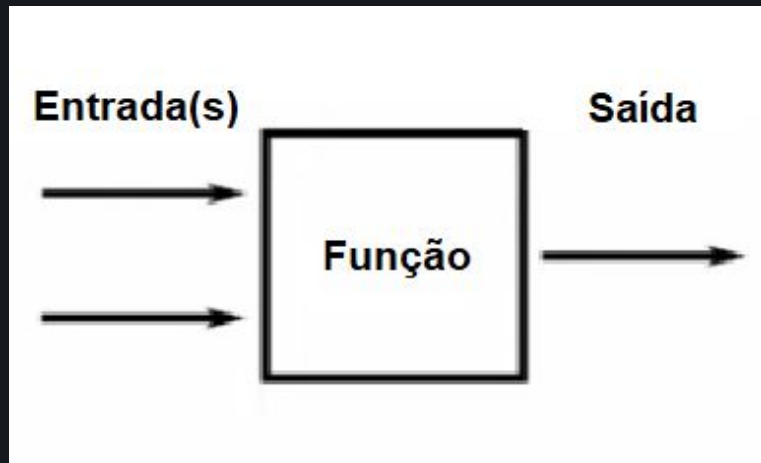


Diagrama de uma função,
a base do paradigma funcional.

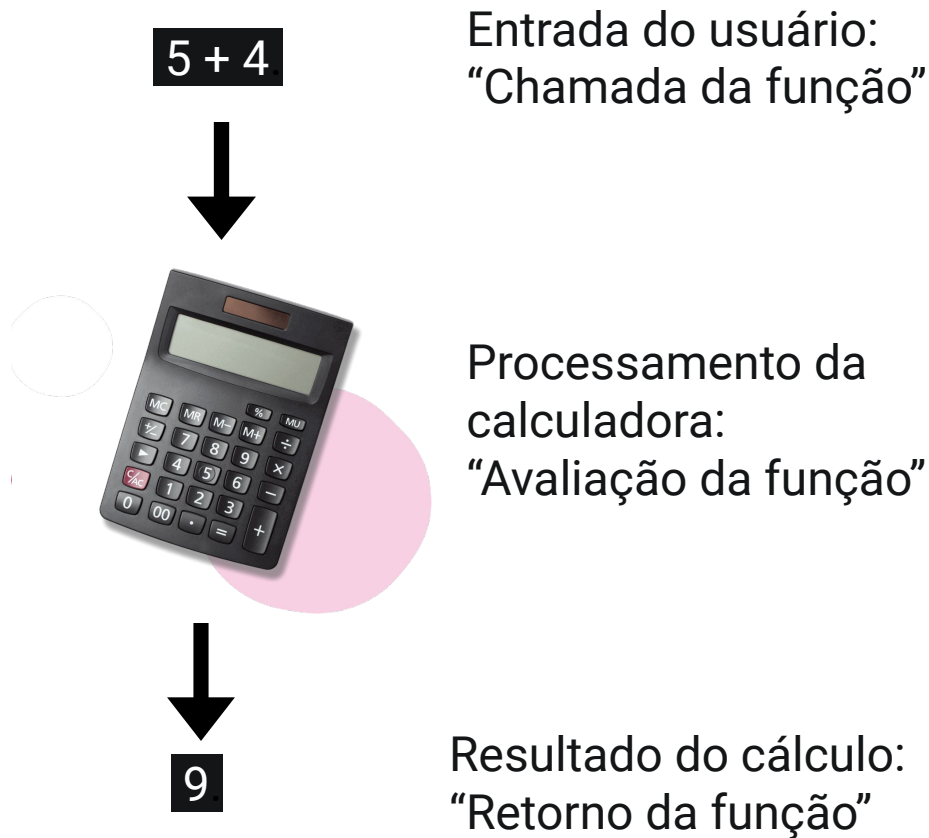
“A programação funcional considera tudo em termos de funções. Não existe uma lista de instruções a serem executados pelo computador, mas sim uma sequência de funções matemáticas que juntas resolverão um problema.”

Pensamento no Paradigma Funcional

Essencialmente, o uso de um programa funcional pode ser visto como um ciclo de entradas e saídas.

De forma análoga, podemos visualizar programas funcionais como uma calculadora.

O usuário faz a entrada de uma expressão, a calculadora processa a expressão, e então exibe o resultado.



A close-up photograph of a person's hand holding a yellow pencil, writing the quadratic formula on a white piece of paper. The formula is written in several steps: first, the expression $\left(x + \frac{b}{2a}\right)^2 = -\frac{c}{a} + \frac{b^2}{4a^2}$ is written; then, it simplifies to $x + \frac{b}{2a} = \pm \frac{\sqrt{b^2 - 4ac}}{2a}$; and finally, the solution is given as $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$. The pencil is positioned at the end of the final formula.
$$\left(x + \frac{b}{2a}\right)^2 = -\frac{c}{a} + \frac{b^2}{4a^2}$$
$$x + \frac{b}{2a} = \pm \frac{\sqrt{b^2 - 4ac}}{2a}$$
$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

A programação funcional é excepcional para usos onde diversos cálculos matemáticos devem ser feitos para demonstrar resultados.

O pensamento no paradigma funcional é voltado à declarações do resultado que desejamos, sem se importar em como será feito pelo computador.

Casos como "qual as raízes dessa função de segundo grau?", "o número x é primo?" ou "qual o resultado de $\log(x)$?" são bons exemplos de como interações com o programa ocorrem neste paradigma.

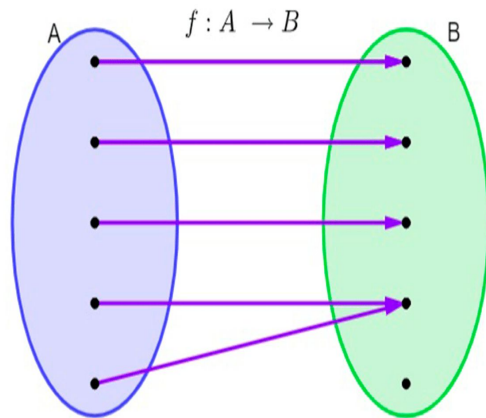
Conclusão

O sistema no paradigma funcional deve **receber entradas** e **demonstrar resultados corretamente** de forma consistente.

As mesmas entradas devem resultar na mesma saída, o resultado não deve se alterar com base no estado do sistema e funções não devem ter efeitos colaterais.

A forma como o programa funcional alcança isso é pela definição de funções determinísticas e de variáveis imutáveis.

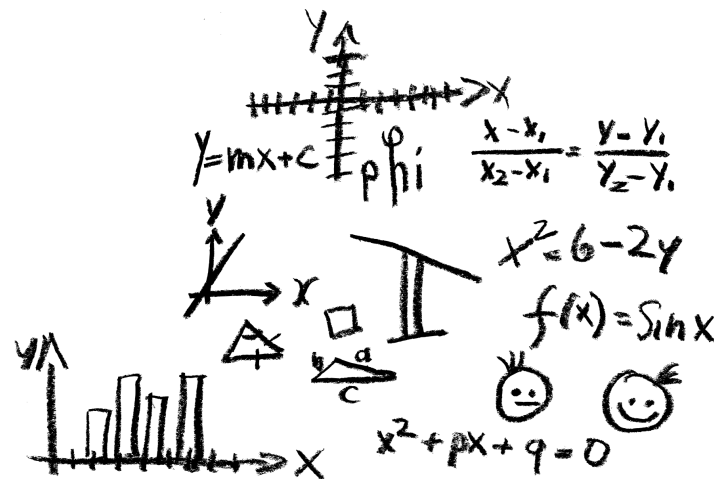
Com tudo isso em mente, é fácil perceber a importância que as funções têm dentro deste paradigma.



Funções

As funções são essenciais para a programação funcional, sendo que a forma de como as funções são avaliadas se difere um pouco dos outros paradigmas.

Além disso, as **funções se assemelham a funções matemáticas** em diversos aspectos, tendendo a ser mais determinísticas e puras.



Composição de funções

A composição de funções é a **prática de “encadear” funções**, sendo muito semelhante ao conceito de funções compostas na matemática em que se pratica o aninhamento de funções.

A composição de funções **não tem limite real**, ou seja, **pode-se aninhar quantas funções quiser**.

Dessa forma, é comum ver na programação funcional grandes “encadeamentos” de funções.

$$(X + (Y * Z))$$

Note que X será somado ao resultado de $Y * Z$



$$(4 + (2 * 3))$$

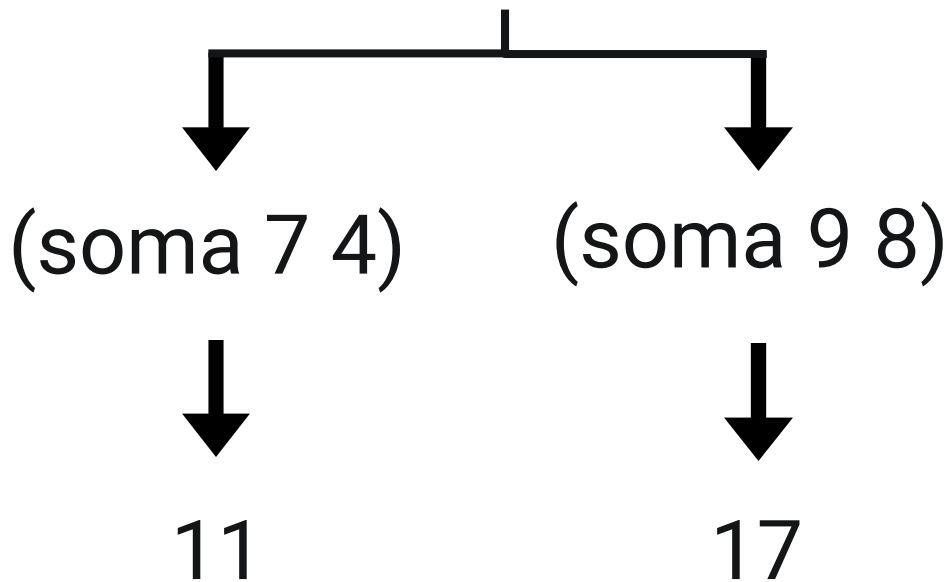


$$(4 + 6)$$



$$10$$

(defun soma (a b)
 (+ a b))



Funções puras

Funções puras formam a base da programação funcional, suas características são:

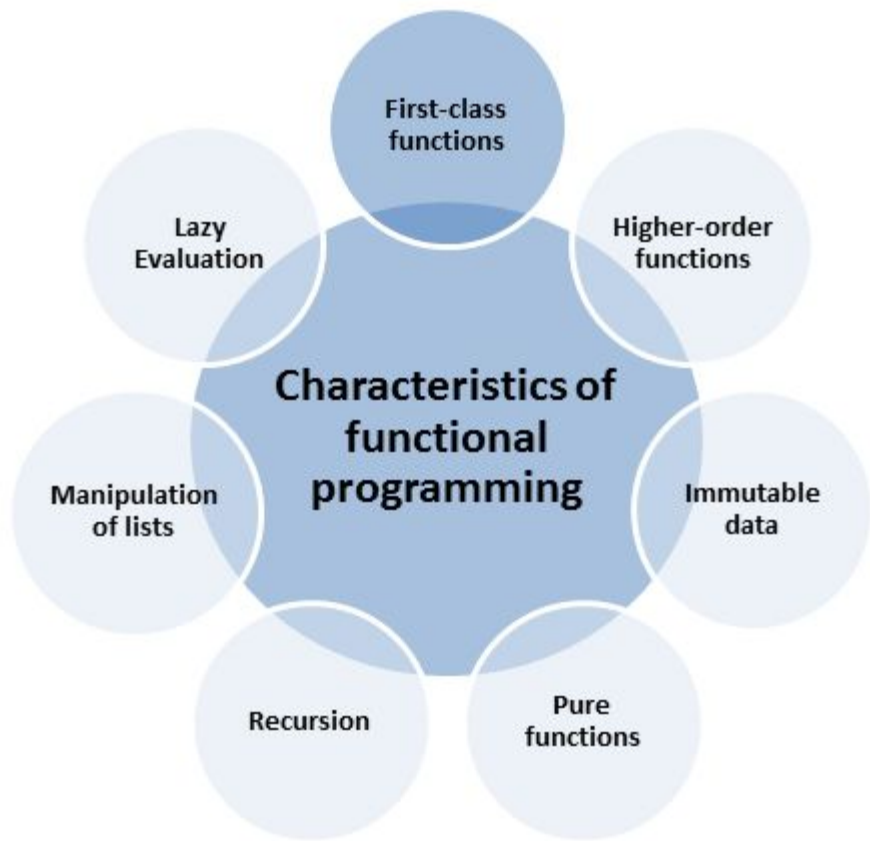
- Elas produzem a mesma saída se a entrada fornecida for a mesma.
- Elas não possuem efeitos colaterais.

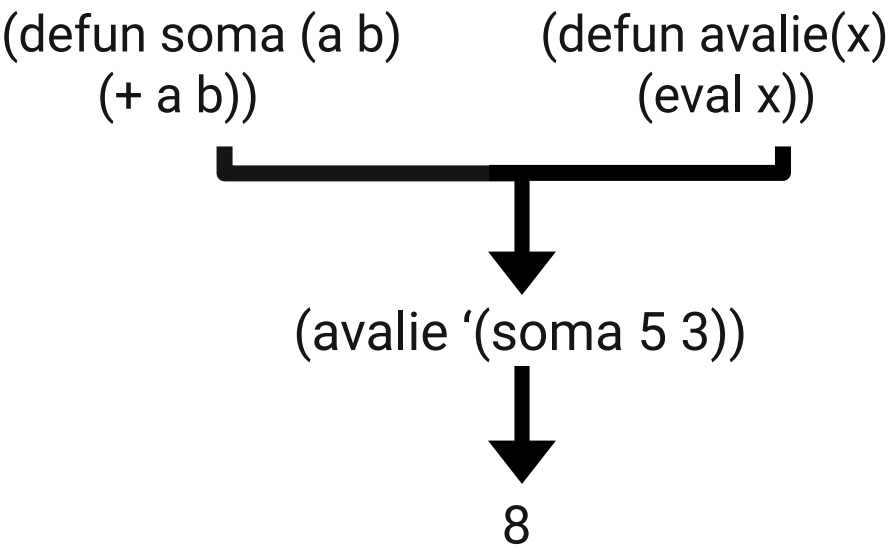
Essas funções funcionam bem com valores imutáveis, pois descrevem como as entradas e saídas irão se relacionarem.

Funções de primeira classe

Em programação funcional, as funções são de primeira classe. Isso significa que podem ser **utilizadas de forma anônima, retornadas por funções, passadas para outras funções como argumentos** ou **armazenadas em estruturas de dados** (agregado ou não).

Vale notar que funções anônimas em Lisp são chamadas de “funções Lambda”.





```
CL-USER> (defun alta-ordem (x)
            (eval x))
ALTA-ORDEM
CL-USER> (alta-ordem '(+ 1 2))
3
```

Funções de alta ordem

Funções no paradigma funcional também podem ser usadas como funções de alta ordem. **Uma função que aceita outras funções como parâmetros ou retornam funções como saídas é chamada de função de alta ordem.**

Essa característica é uma mera consequência do fato que funções são cidadãos de primeira classe no paradigma funcional.

Imutabilidade

Na programação funcional, **não podemos modificar uma variável após ela ser criada.**

A razão para isso é que gostaríamos de manter o estado do programa consistente durante todo o tempo de execução.

Isso significa que quando criamos variáveis, podemos executar o programa com segurança, sabendo que o valor das variáveis permanecerá constante e nunca poderá mudar - isso garante uma previsibilidade maior do nosso código.

É como se as variáveis no paradigma funcional fossem equivalentes às constantes do paradigma imperativo.

↓

x	=	10	✓
y	=	20	✓
x	=	30	✗



Transparência referencial

Consequente da imutabilidade e uso de funções puras, o paradigma funcional possui o que chamamos de “transparência referencial”.

A transparência referencial se refere ao fato que **funções podem ser consideradas iguais ao seu retorno equivalente**.

Assim, cada parte do programa funcional **sempre tem o mesmo resultado**, independentemente do contexto em que ele se encontra.

```
CL-USER> (defun soma (x y)
            (+ x y))
SOMA
CL-USER> (soma 5 4)
9
```

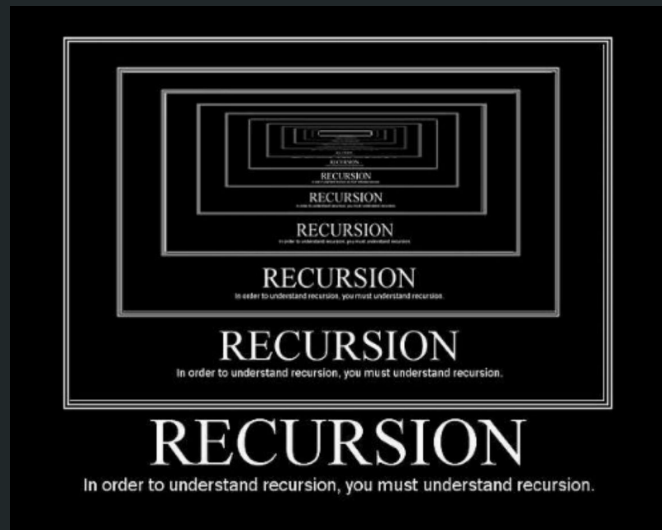
A função (soma 5 4) pode ser considerada 9 em qualquer local do código, independente do contexto.

Recursão

Na programação funcional não é **desejável fazer uso de loops “while” ou “for”**. Os programas funcionais evitam expressões que criam resultados diferentes em cada execução.

O uso de funções recursivas é mais comum.

Funções recursivas chamam a si mesmas repetidamente até atingirem o estado ou solução desejada, conhecida como caso base.



Exemplo de recursão: Função fatorial

```
(defun fat (x)
  (if (<= x 1) 1
      (* x (fat (- x 1)))))
```

Neste caso, $x \leq 1$ é o nosso caso base. A função vai continuar chamando a si mesma, diminuindo o valor de x por 1 em cada chamada, até chegar nesse caso base, finalizando assim seu retorno.

```
(fat 3) ; Chamada inicial
(* 3 (fat 2))
(* 3 (* 2 (fat 1)))
(* 3 (* 2 (* 1 1)))
(* 3 (* 2 1))
(* 3 2)
6 ; Retorno final
```

Recursão em cauda

Uma função é dita recursiva de cauda se a chamada recursiva for a última coisa feita na função antes de fazer seu retorno. Normalmente, o caso base é tratado no **final da função**.

O exemplo da função fatorial apresentado também se aplica aqui. A função fatorial é uma função recursiva em cauda.

```
(defun factorial (n acc)
  (if (= n 1)
      acc
      (recur (- n 1) (* acc n))))
```



A chamada recursiva acontece
no final da função.

Lazy Evaluation

Também chamada de "Avaliação preguiçosa", a Lazy Evaluation é uma estratégia de avaliação de expressões amplamente suportada em linguagens funcionais.

Essa estratégia consiste no **atraso da avaliação de expressões até que o valor dessa avaliação seja necessário**, também sendo usada para **evitar avaliações desnecessárias em cláusulas lógicas**.

A lazy evaluation é suportada pela maioria das linguagens funcionais, e tem diversos usos. Não vamos entrar em detalhes nestes usos, mas vale notar de forma geral que a lazy evaluation pode ser usada para controlar o fluxo de avaliação de expressões e criar estruturas de dados complexas.

Lazy Evaluation - exemplo 1

Considere o seguinte pseudocódigo:

```
1. x = 10 + 5  
2. print("ola mundo!")  
3. print(x)
```

Ao final da linha 1, $x = 15$

Ao final da linha 2, $x = 15$

Ao final da linha 3, $x = 15$

Sem lazy evaluation, desde a linha 1 o valor de x será 15.

Isso acontece pois a avaliação da expressão $10 + 5$ acontece na hora que a alocação à variável acontece.

A avaliação ocorre sempre de forma imediata sem a lazy evaluation.

Lazy Evaluation - exemplo 1

Com lazy evaluation, a expressão $10 + 5$ não será avaliada já no momento da alocação da variável. A avaliação vai ser “deixada para depois.”

A avaliação apenas vai ocorrer quando o valor da variável for requisitado.

```
1. x = 10 + 5  
2. print("ola mundo!")  
3. print(x)
```

Ao final da linha 1, $x = 10 + 5$

Ao final da linha 2, $x = 10 + 5$

Ao final da linha 3, $x = 15$



Valor de x é requisitado: Agora não tem jeito! vou ter que avaliar.

Lazy Evaluation - exemplo 2

Considere também este pseudocódigo:

```
termoT = TRUE  
termoF = FALSE  
  
if (termoF AND termoT)
```

Com lazy evaluation, a avaliação da expressão terminará no termoF. O termoT nunca será avaliado, pois a expressão imediatamente retorna falso.

Isso acontece pois o programa entende que, tendo um único termo falso no AND, o AND já será falso independente dos outros termos. Assim, ele nem termina de avaliar os termos restantes do AND e apenas retorna falso.

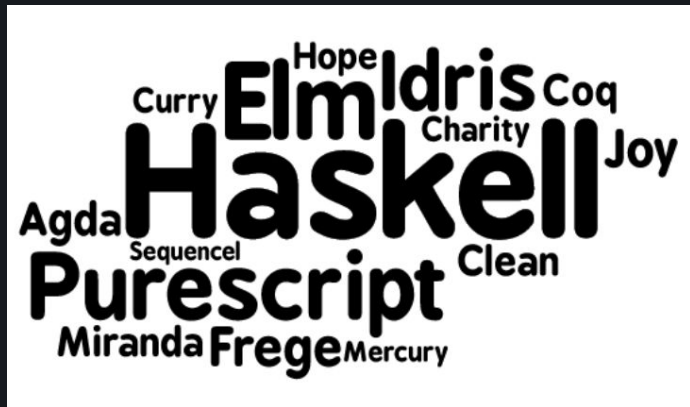
Essa lógica também pode ser aplicada ao OR, onde se um único termo for verdadeiro, o programa já cessa a avaliação e retorna verdadeiro.

Paradigma funcional puro

A definição do "paradigma funcional puro" não é muito certa, havendo variações de acordo com a literatura consultada. Em geral, surge na ideia de seguir as ideias teóricas do paradigma funcional de forma **estrita**.

Common Lisp é "**impuro**". Embora sejam indesejados e fortemente rejeitados, **efeitos colaterais ainda podem ser feitos, variáveis podem ser modificadas e loops estruturais podem ser usados**.

A linguagem Haskell é considerada "**pura**", ou seja, **não permite a criação de efeitos colaterais, a alteração de variáveis e o uso de loops estruturais nativamente**.



[Linguagens funcionais puras](#)

Paradigma funcional impuro é paradigma funcional?

Entenda: só porque Common Lisp permite a alteração de variáveis, criação de efeitos colaterais e uso de loops estruturais **não** significa que a linguagem não aplica o paradigma funcional.

Todas essas coisas que o Common Lisp permite fazer não são boa prática, e existem apenas como ferramentas para “caso seja necessário”.

Em um programa normal, e em casos ideais, você **não** deverá usar essas ferramentas.

Elas realmente só estão lá para aumentar a flexibilidade da linguagem em casos específicos, onde por algum motivo você precisou de, por exemplo, modificar uma variável.

Puro X Impuro

No final das contas, se a linguagem funcional será pura ou não depende do projeto dela.

Linguagens puras tendem a incentivar códigos mais funcionais, porém podem ser problemáticas quando uma funcionalidade não suportada deve ser usada.

Imagina ter um sistema de décadas em Haskell, que, por uma mudança nas regras de negócio, agora precisa causar um único efeito colateral?

Apenas um efeito colateral talvez não justifique trocar todo o paradigma a ser usado em sua solução, e por isso você quer manter o sistema em Haskell.

O problema é que agora você vai ter que gastar tempo e recursos fazendo gambiarras para permitir um efeito colateral no seu código Haskell.

Por isso algumas linguagens preferem se manter impuras - elas não QUEREM as impurezas, mas aceitam que em soluções reais pode acontecer de serem necessárias.

Linguagens - Lisp

Até agora tratamos dos conceitos do paradigma funcional na teoria. Brevemente, vamos agora apontar algumas aplicações reais deste paradigma.

O maior e mais famoso exemplo da aplicação do paradigma funcional é a família de linguagens **Lisp**.

Muitas das linguagens funcionais conhecidas são dialetos Lisp, e a própria história do paradigma é intrinsecamente conectada à história do Lisp.



Scheme, Common Lisp, Racket e Clojure são exemplos de dialetos lisp.

Linguagens - Outras

Outras linguagens funcionais notáveis, mas não parte da família de linguagens Lisp:

Haskell - Linguagem que tenta aplicar o conceito da "programação funcional pura". É uma das maiores linguagens funcionais.

Elixir - Linguagem brasileira especialista em escalabilidade e eficiência.

Rust - Linguagem multi paradigma que visa a criação de código seguro.



História

Paradigma Funcional: A Origem



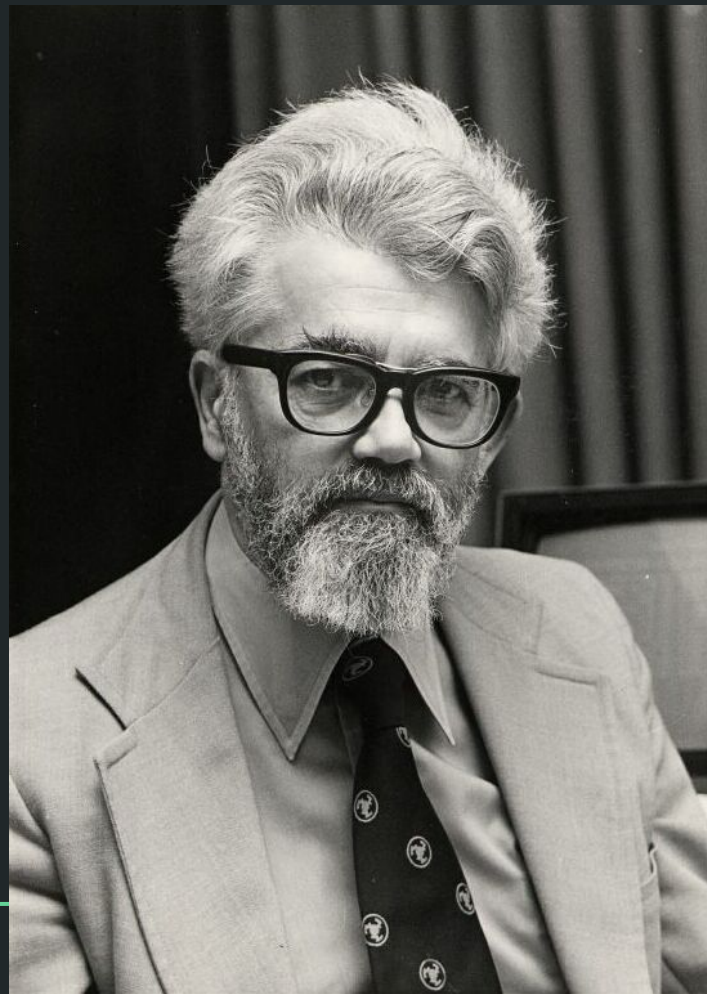
O paradigma funcional emergiu como uma abordagem de programação **influenciada** pela matemática, em particular pelo **cálculo lambda** de Alonzo Church, desenvolvido nos anos 1930. Church introduziu o cálculo lambda como um formalismo para descrever computações por meio de funções matemáticas. Isso levou ao conceito fundamental de que as funções podem ser tratadas como cidadãos de primeira classe,

Outra influência importante foi o trabalho de Haskell Curry, que desenvolveu a **teoria da combinação funcional**, que também contribuiu para os fundamentos do paradigma funcional.

O surgimento do paradigma funcional foi justamente com a linguagem de programação Lisp desempenhando um papel significativo ao introduzir conceitos como **funções de ordem superior e recursão**. Dito isso, vamos desenvolver mais o Lisp nos slides a seguir.

Sobre o criador do Lisp

John McCarthy foi o criador da linguagem Lisp e ficou conhecido tanto por isso quanto pelo seu **vasto conhecimento sobre inteligência artificial**, vale citar também que ele foi **estudante do Church**. Ele foi um **cientista da computação** e trabalhou em instituições como a **Universidade de Stanford** e o **Instituto de Tecnologia de Massachusetts**. McCarthy foi agraciado com o **Prêmio Turing** em 1971 e com a **Medalha Nacional de Ciências dos Estados Unidos** durante a sua vida.



O Começo : IPL



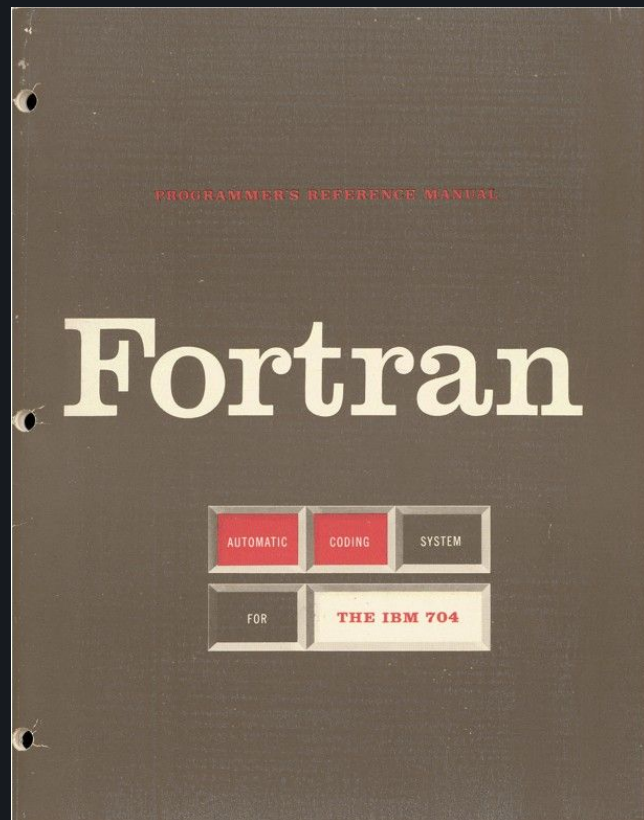
A origem do Lisp **começou** a ocorrer aproximadamente em **1956**, quando, durante uma **reunião de verão** no Dartmouth College sobre inteligência artificial, **John McCarthy** aprendeu sobre a técnica chamada "**processamento de lista**". Algo relativamente novo para a época, tomando como partido que a maioria da programação era feita através do assembly, diretamente no circuito da máquina. Esse "**processamento de lista**", criado por Allen Newell, J. C. Shaw e Herbert Simon, era **chamado de IPL (Information Processing Language)**, baseando sua manipulação através de listas e símbolos. Embora ainda possuísse uma sintaxe similar ao assembly, a manipulação desses dados era de **suma importância para a programação de inteligência artificial**.

Todavia, ainda por volta da década de **1950**, **uma nova linguagem surgiu chamada Fortran**. Ela permitia que o programador saísse do assembly e pudesse codificar em termos matemáticos, sendo mais fácil expressar suas ideias e deixando para o computador a tarefa de converter isso para assembly e executar na máquina. Essa inovação radical fez com que McCarthy desejasse criar uma linguagem para símbolos igualmente poderosos.

Pontos positivos que fez ele ficar popular:

- Fortran permite que você use variáveis simbólicas, o que significa que você **pode usar a mesma variável em uma expressão após defini-la anteriormente**. Exemplo : “a + a” antes de definir a.
- Fortran suporta expressões algébricas, permitindo que você escreva fórmulas matemáticas diretamente no código.
- Fortran possui uma instrução chamada **Arithmetic IF** que permite tomar decisões com base no resultado de uma expressão aritmética, especificamente se ela é negativa, zero ou positiva.
- Fortran suporta a composição de **sub-rotinas**, permitindo que você divida seu código em **partes menores** e mais gerenciáveis para facilitar a leitura e a manutenção.

Um sinal de vida: Fortran



Cada Vez mais Perto: FLPL



A primeira sugestão foi criar uma **linguagem baseada no Fortran**, desenvolvendo assim um conjunto de sub-rotinas especiais para a manipulação das listas. Essa ideia foi implementada por Herbert Gelerntner e Carl Gerberich, que criaram a **FLPL (FORTRAN List Processing Language)** para a máquina IBM.

Principais Pontos:

- Replicar conceitos do **IPL**
- Estender o Fortran com listas
- Mas tudo tinha que ser um **número inteiro** (ou seja, nenhum domínio simbólico)

Existiam Problemas:

- Difícil de visualizar listas em Fortran
- Valor de uma função dependendo do estado do registrador
- IF Aritmético com passagem por referência: **ambas as ramificações precisam ser avaliadas**
- Incapacidade de estender funções padrão

LISP

Todavia, McCarthy, devido à sua experiência em diversos locais, decidiu criar sua própria linguagem nova, baseando-se tanto na IPL quanto na FORTRAN, no FLPL, e nos problemas presentes. Assim, surgiu a primeira versão do LISP (List Processor), também para a máquina IBM 704.

Contudo, foi somente em **1962, com o lançamento do Lisp 1.5 Programmer's Manual**, que ele começou a ser amplamente utilizado. Nessa época, o Lisp já era empregado em vários computadores, tornando-se uma das primeiras linguagens interativas e levando até ao desenvolvimento de computadores projetados com foco em seu desempenho.

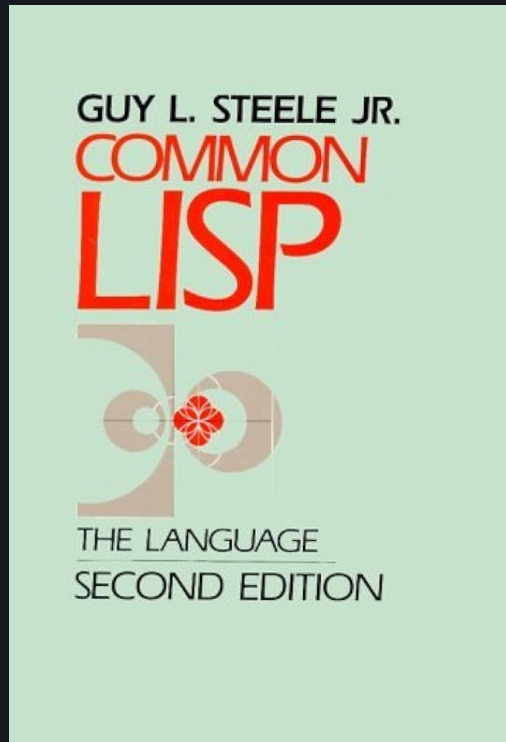


Muito Depois da Criação



A partir de meados da década de 1960, começou a haver uma forte divergência na comunidade, o que resultou em uma divisão a partir do dialeto original do Lisp 1.5. Isso gerou caminhos incontáveis e improváveis para o rumo do dialeto Lisp, com várias pessoas implementando seus próprios dialetos baseados na versão 1.5 e em seus próprios conceitos e ideias. Um exemplo é o **Stanford Lisp** 1.6, uma versão mais nova do MacLisp, que posteriormente se tornou o UCI Lisp. Em meio a isso, Guy Steele e Gerald Sussman decidiram criar um novo tipo de Lisp, chamado Scheme. Ele tentava combinar de forma elegante as ideias da família dos algoritmos com a poderosa sintaxe e estrutura de dados do Lisp. Assim, o **Scheme** teve seu desenvolvimento e evolução paralelos ao desenvolvimento do Lisp.

Common Lisp



Em meados da década de 1980, por volta de 1984, já existiam diversas versões e implementações obsoletas e incompatíveis do Lisp. Um dos criadores do Scheme, junto com Scott Fahlman, Daniel Weinreb, David Moon e Richard Gabriel, criou o Common Lisp. Essa nova linguagem reuniu as melhores funções dos diferentes dialetos de Lisp existentes no mercado. Após algum tempo, foi lançada uma versão revisada, que foi amplamente aclamada e rapidamente acolhida pela comunidade. A existência do Common Lisp acabou suprimindo certos dialetos, exceto o Scheme, que continuou seu desenvolvimento e popularidade.

Vantagens

Vantagens

Variáveis previsíveis: Elas são imutáveis, portanto nunca terá dúvidas sobre seu conteúdo.

Funções Puras: Funções puras definem saídas determinísticas e não causam efeitos colaterais, sendo assim, previsíveis e de menor complexidade.

Programação Paralela: Pela previsibilidade do código, é possível dividir uma tarefa em várias partes e executar todas simultaneamente com muita mais facilidade.



Vantagens

Depuração: Como o código é previsível e marcado pela imutabilidade, fica fácil e simples de achar bugs e erros de código.

Modularização e escalabilidade: A composição feita pelo uso de funções reutilizáveis para realizar as operações torna programas extremamente modularizado e com intuitiva escalabilidade.

Abstração: Uma vez feita a estrutura da função, não é necessário saber exatamente todas as lógicas dentro dela, basta chama ela.

Lazy evaluation: A lazy evaluation permite controle melhor do fluxo de avaliações de expressões, além do desenvolvimento de técnicas avançadas de manipulação de estruturas de dados. Ela também promove melhor desempenho em algumas situações.

Vantagens

Flexibilidade das funções: Por serem cidadãos de primeira classe, funções podem ser manipuladas de diversas formas, aumentando a gama de possibilidades dentro do código.

Sinergia com matemática: Soluções que precisam seguir conceitos matemáticos mais estritamente se beneficiam de utilizar o paradigma funcional.

Fácil legibilidade: Código funcional pode ser mais facilmente lido por possuir variáveis e funções previsíveis. Os resultados das avaliações são mais claros.

Desvantagens

Desvantagens



Curva de aprendizado: Aprender a programar de forma funcional pode ser um desafio para desenvolvedores acostumados com paradigmas como a programação imperativa e orientada a objetos. Conceitos como funções puras, recursão e imutabilidade podem exigir um novo modo de pensar e resolver problemas.

Imutabilidade: Se a sua solução precisa de dados sendo constantemente alterados, atualizados e gerenciados, o paradigma funcional pode não ser adequado para sua solução.

Dificuldade com certos tipos de problemas: Nenhum paradigma é a solução perfeita, e por isso a programação funcional nem sempre é a melhor escolha para todos os tipos de problemas. Por exemplo, ela pode ser menos adequada para tarefas que exigem muita interação com hardware ou modificação de estado persistente.

Desvantagens

Falta de suporte em algumas linguagens: Nem todas as linguagens de programação que oferecem suporte ao paradigma funcional fazem isso de forma completa. Linguagens como Java ou C++, por exemplo, possuem recursos funcionais limitados, o que pode exigir soluções alternativas ou adaptações.

Integração com código de outros paradigmas: Integrar código funcional com sistemas existentes escritos em paradigmas diferentes pode ser complexo. A interoperabilidade entre paradigmas pode exigir adaptações significativas, aumentando a complexidade do desenvolvimento e da manutenção.

Stack Traces Complexos: Erros em programas funcionais podem resultar em stack traces longos que são difíceis de interpretar, especialmente em casos de composições imensas de funções. Desenvolvedores devem se familiarizar a forma que Stack Traces devem ser lidos.

Desvantagens

Recursão em vez de loops estruturais: A programação funcional geralmente utiliza recursão em vez de loops tradicionais. Em linguagens que não otimizam adequadamente chamadas recursivas (como otimização de cauda), isso pode levar a problemas de desempenho e estouro de pilha.

Gerenciamento de efeitos colaterais: Embora a programação funcional busque minimizar efeitos colaterais, em aplicações reais lidar com I/O, interações com banco de dados e outros efeitos colaterais podem ser necessários. A integração disso em programas funcionais exige técnicas avançadas, como monads em Haskell. Se seu programa *precisa* de efeitos colaterais, seria melhor não usar o paradigma funcional.



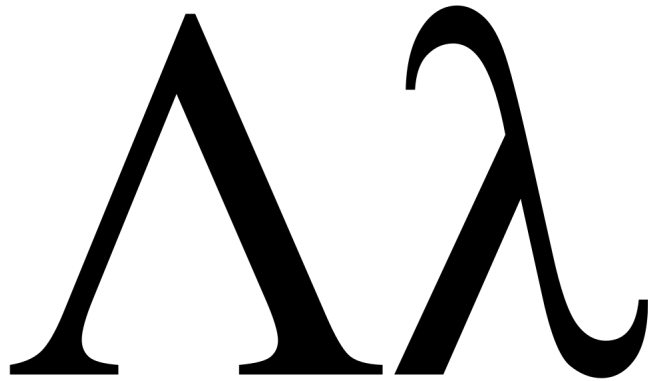
Desvantagens

Performance: Em alguns casos, a ênfase na imutabilidade gera a criação de novas cópias de estruturas de dados, e o uso de recursão em excesso pode causar problemas de alocação no stack, fatores que podem resultar em overhead de desempenho e maior uso de memória, especialmente se o gerenciamento de memória não for otimizado.

Ferramentas e bibliotecas limitadas: Embora linguagens funcionais populares como Haskell e Lisp tenham um ecossistema crescente, elas ainda podem ter menos suporte de bibliotecas e ferramentas em comparação com linguagens imperativas amplamente usadas como JavaScript ou Python.

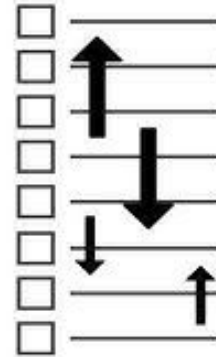
Comparativos com outros Paradigmas

Paradigma Funcional x Paradigma Imperativo



x

Imperative



Paradigma Funcional x Paradigma Imperativo

Abordagem

Na programação funcional, o foco está na avaliação de expressões e na aplicação de funções matemáticas. O programa é composto por uma série de funções puras, que não têm efeitos colaterais e retornam um valor com base em seus argumentos.

Na programação imperativa, o foco está na execução de uma sequência de instruções que modificam o estado do programa. O programa é composto por comandos que alteram variáveis e controlam o fluxo de execução.

Estado e mutabilidade

O paradigma funcional enfatiza a imutabilidade dos dados. As funções não alteram o estado dos objetos, mas produzem novos objetos como resultado. Isso facilita a escrita de código livre de efeitos colaterais, tornando o programa mais fácil de entender e testar.

No paradigma imperativo, o estado é mutável. Variáveis podem ser alteradas repetidamente ao longo do programa, e a modificação do estado é uma parte central do controle de fluxo.

Paradigma Funcional x Paradigma Imperativo

Controle de Fluxo

Na programação funcional, o controle de fluxo é realizado por meio de funções de ordem superior e recursão, eliminando a necessidade de loops explícitos.

Na programação imperativa, o controle de fluxo é explícito, usando estruturas como loops (for, while) e condicionais (if, switch) para determinar a sequência de execução.

Tratamento de Efeitos Colaterais

A programação funcional evita efeitos colaterais, favorecendo funções puras que não alteram o estado global. Isso resulta em um código mais previsível e fácil de testar.

A programação imperativa aceita e frequentemente depende de efeitos colaterais, onde funções e procedimentos podem alterar o estado do programa ou de variáveis globais, o que pode dificultar a compreensão e a depuração do código.

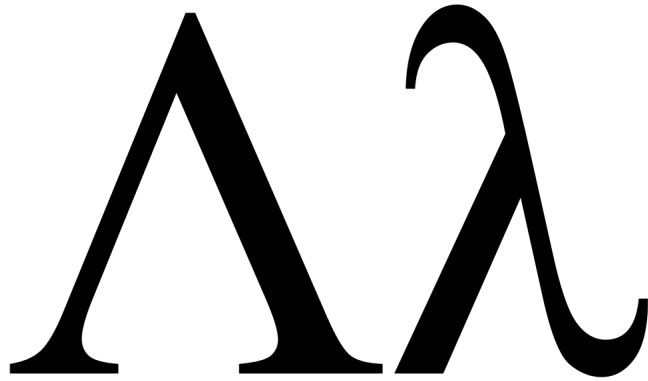
Paradigma Funcional x Paradigma Imperativo

Paralelismo

A imutabilidade e ausência de efeitos colaterais na programação funcional facilitam o paralelismo, pois diferentes partes do código podem ser executadas simultaneamente sem risco de interferência.

O paradigma imperativo, com seu estado mutável, pode apresentar dificuldades para realizar o paralelismo, exigindo técnicas adicionais como locks e sincronização para evitar condições de corrida.

Paradigma Funcional x Paradigma Orientado a Objetos



x



Paradigma Funcional x Paradigma Orientado a Objetos

Abordagem

Na programação funcional, o foco está na avaliação de expressões e na aplicação de funções matemáticas. O programa é composto por uma série de funções puras, que não têm efeitos colaterais e retornam um valor com base em seus argumentos.

Na programação orientada a objetos (POO), o foco está na modelagem de objetos que encapsulam dados e comportamentos relacionados. Programas são compostos por classes e objetos, onde os objetos são instâncias das classes e podem interagir entre si por meio de mensagens.

Estado e mutabilidade

O paradigma funcional enfatiza a imutabilidade dos dados. As funções não alteram o estado dos objetos, mas produzem novos objetos como resultado. Isso facilita a escrita de código livre de efeitos colaterais, tornando o programa mais fácil de entender e testar.

Por outro lado, a POO permite a mutabilidade dos objetos. Os objetos podem alterar seu estado interno e, portanto, podem ter efeitos colaterais. Isso facilita a modelagem de certos domínios de problema e a implementação de interações complexas entre objetos.

Paradigma Funcional x Paradigma Orientado a Objetos

Herança e polimorfismo

A programação funcional utiliza conceitos como funções de ordem superior e composição de funções para permitir a reutilização de código e a manipulação de abstrações.

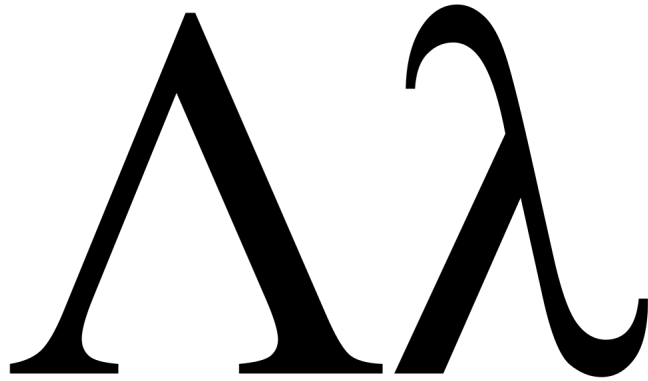
A programação orientada a objetos suporta herança e polimorfismo. A herança permite que as classes herdem atributos e comportamentos de classes pai, enquanto o polimorfismo permite tratar objetos de diferentes classes de maneira uniforme quando compartilham uma interface comum.

Estrutura do programa

Com maior foco em funções e na composição de expressões, os programas funcionais são frequentemente escritos com funções de ordem superior, recursão e técnicas como mapeamento e redução.

Já os programas orientados a objetos são escritos em termos de mensagens enviadas entre objetos e manipulação de estado, com ênfase na organização dos dados e comportamentos em classes e objetos.

Paradigma Funcional x Paradigma Lógico



x



Paradigma Funcional x Paradigma Lógico

Abordagem

Na programação funcional, o foco está na avaliação de expressões e na aplicação de funções matemáticas. O programa é composto por uma série de funções puras, que não têm efeitos colaterais e retornam um valor com base em seus argumentos.

Na programação lógica, o foco está na definição de relações e na utilização de regras lógicas para inferir novos fatos a partir de fatos existentes. Programas são conjuntos de declarações lógicas e consultas.

Forma de Resolução

A programação funcional executa diretamente funções e expressões, aplicando funções a dados de maneira previsível e determinística.

A programação lógica utiliza um motor de inferência para resolver consultas, explorando diferentes combinações de fatos e regras até encontrar soluções que satisfazem as condições especificadas.

Paradigma Funcional x Paradigma Lógico

Dados e Conhecimento

No paradigma funcional, os dados são passados entre funções, e o conhecimento é encapsulado dentro das funções.

Na programação lógica, os dados e o conhecimento são representados como fatos e regras. O programa descreve o que é verdadeiro e o motor de inferência descobre como alcançar as soluções.

Aplicabilidade

A programação funcional é bem adequada para tarefas que envolvem transformação de dados, processamento paralelo e manipulação de listas.

A programação lógica é ideal para problemas que envolvem raciocínio simbólico, como sistemas especialistas, prova automática de teoremas e consulta de bases de dados complexas.

Paradigma Funcional x Paradigma Lógico

Execução e Desempenho

A execução na programação funcional é geralmente mais direta e eficiente, com avaliações de expressões acontecendo de forma previsível.

A programação lógica pode ser menos eficiente, especialmente em problemas complexos, devido à necessidade de explorar múltiplos caminhos de inferência e realizar backtracking.

Como instalar LISP?

Instalando o Portacle

Caso queira baixar o LISP o mais rápido possível, o recomendado é usar o Portacle.

Acessando o site do Common Lisp, na área de downloads, basta clicar no botão de download do Portacle.

Download and Install

If you are a newbie or you want to get started as fast as possible, then **Portacle** is probably your best option. Portacle is a multiplatform, complete IDE for Common Lisp. It includes Emacs, SBCL, Git, Quicklisp, all configured and ready to use.

 [Download Portacle - All-In-One Common Lisp](#)



Instalando o Portacle

Ao clicar no botão, você será redirecionado para o site do Portacle.

Clicando no botão de download para windows, você será transferido para área de download.



Instalando o Portacle

Para fazer o download do instalador, você precisa clicar em "latest release".

Após isso, selecione o local do seu computador onde deseja salvar o instalador do programa.



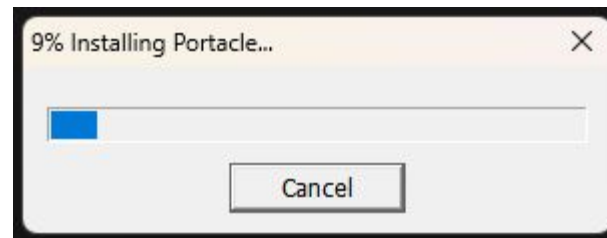
Windows

Download the **latest release** and run it. It will ask you where to install it to, defaulting to your home folder. Note that you do not need to append `portacle` to the end of the path. After extraction, you can launch it by double-clicking the `portacle.exe`.

Note that `portacle.exe` is tied to the `portacle` directory and needs everything within it to function properly. You can however create a shortcut to the exe to reach it more easily from your desktop.

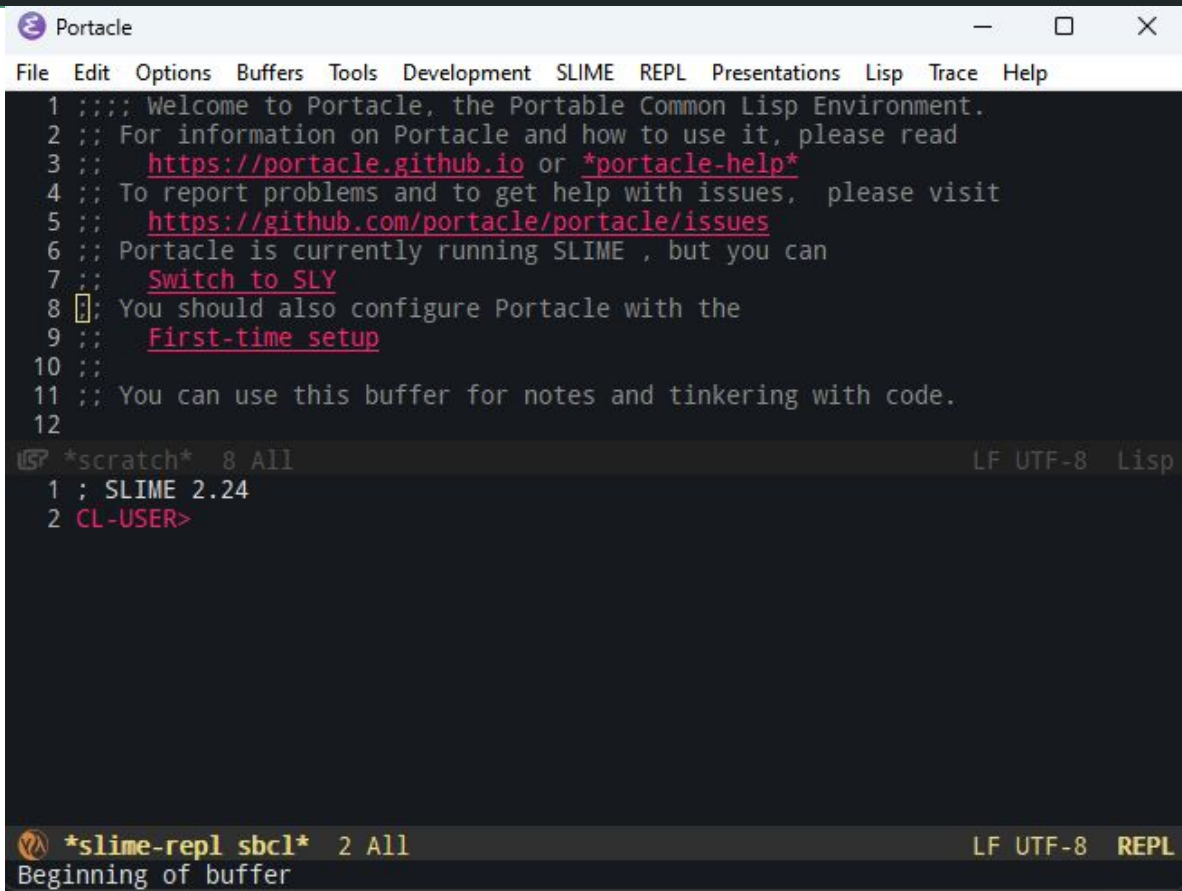
Instalando o Portacle

Executando o instalador, selecione em qual pasta será salvo o programa.



Instalando o Portacle

Abrindo a pasta, clique para executar o arquivo "portacle.exe" que o programa irá iniciar, estando pronto para ser utilizado.



The screenshot shows the Portacle application window. The title bar says "Portacle". The menu bar includes File, Edit, Options, Buffers, Tools, Development, SLIME, REPL, Presentations, Lisp, Trace, and Help. The main text area displays a welcome message with several links and instructions. At the bottom, there are two buffers: "*scratch*" and "*slime-repl sbcl*".

```
Portacle
File Edit Options Buffers Tools Development SLIME REPL Presentations Lisp Trace Help

1 ::; Welcome to Portacle, the Portable Common Lisp Environment.
2 :: For information on Portacle and how to use it, please read
3 :: https://portacle.github.io or \*portacle-help\*
4 :: To report problems and to get help with issues, please visit
5 :: https://github.com/portacle/portacle/issues
6 :: Portacle is currently running SLIME , but you can
7 :: Switch to SLY
8 :: You should also configure Portacle with the
9 :: First-time setup
10 ::
11 :: You can use this buffer for notes and tinkering with code.
12

*scratch* 8 All LF UTF-8 Lisp
1 ; SLIME 2.24
2 CL-USER>

*slime-repl sbcl* 2 All LF UTF-8 REPL
Beginning of buffer
```

Instalando no VS Code

Caso queira utilizar o LISP no Visual Studio Code, terá que clicar em "Download", na área do Steel Bank Common Lisp (SBCL).

Download and Install

If you are a newbie or you want to get started as fast as possible, then [Portacle](#) is probably your best choice for Common Lisp. It includes Emacs, SBCL, Git, Quicklisp, all configured and ready to use.

 [Download Portacle - All-In-One Common Lisp](#)


Otherwise, Common Lisp comes in many different [flavors, or implementations](#).

Two popular *open source* ones are Steel Bank Common Lisp (SBCL) and Clozure Common Lisp (CCL):

Steel Bank Common Lisp (SBCL)

- [Main website](#)
- [Download](#)  
- [Install](#)

Clozure Common Lisp (CCL)

- [Main website](#)
- [Download/Install](#) 

Look [here](#) for other Common Lisp compilers, including powerful *commercial* implementations.

You can also try Common Lisp online:

 [Try Lisp Online](#)

Instalando no VS Code

Ao clicar, será redirecionado para o site do SBCL.

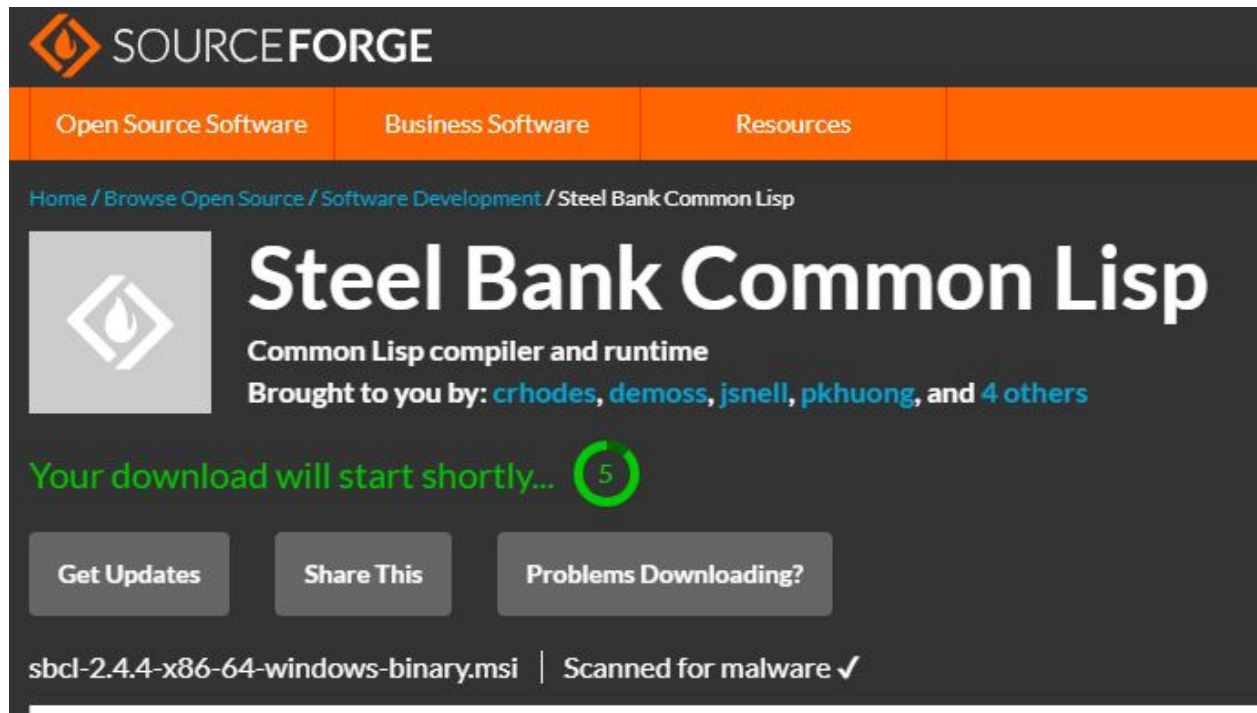
Clique no quadrado verde da versão SBCL que deseja, no nosso caso, Windows AMD64.

[illegible]

Instalando no VS Code

Você será direcionado para o site do courseforge, e o download do instalador irá ser iniciado após 5 segundos.

Caso o download não inicie, basta clicar no botão de download após alguns segundos de espera.




The screenshot shows the SourceForge project page for Steel Bank Common Lisp. The page has a dark theme with orange accents. At the top, the SourceForge logo is on the left, and navigation links for 'Open Source Software', 'Business Software', and 'Resources' are on the right. Below the navigation bar, a breadcrumb trail reads 'Home / Browse Open Source / Software Development / Steel Bank Common Lisp'. The main content area features a large icon of a flame inside a diamond shape, followed by the project title 'Steel Bank Common Lisp' in large white text. Below the title, it says 'Common Lisp compiler and runtime' and 'Brought to you by: crhodes, demoss, jsnell, pkhuong, and 4 others'. A green circular progress indicator with the number '5' inside is shown next to the text 'Your download will start shortly...'. At the bottom of the main content area, there are three buttons: 'Get Updates', 'Share This', and 'Problems Downloading?'. Below these buttons, the filename 'sbcl-2.4.4-x86-64-windows-binary.msi' is displayed, followed by a checkmark and the text 'Scanned for malware'.

SOURCEFORGE

Open Source Software Business Software Resources

Home / Browse Open Source / Software Development / Steel Bank Common Lisp

 **Steel Bank Common Lisp**

Common Lisp compiler and runtime

Brought to you by: [crhodes](#), [demoss](#), [jsnell](#), [pkhuong](#), and [4 others](#)

Your download will start shortly... 5

Get Updates Share This Problems Downloading?


sbcl-2.4.4-x86-64-windows-binary.msi | Scanned for malware ✓

Instalando no VS Code

Você será direcionado para o site do courseforge, e o download do instalador irá ser iniciado após 5 segundos.

Caso o download não inicie, basta clicar no botão de download após alguns segundos de espera.

[Home](#) / [Browse Open Source](#) / [Software Development](#) / [Steel Bank Common Lisp](#)





Steel Bank Common Lisp

Common Lisp compiler and runtime
Brought to you by: [crhodes](#), [demoss](#), [jsnell](#), [pkhuong](#), and [4 others](#)

★★★★★ 12 Reviews

Downloads: 1,460 This Week

 **Download** 

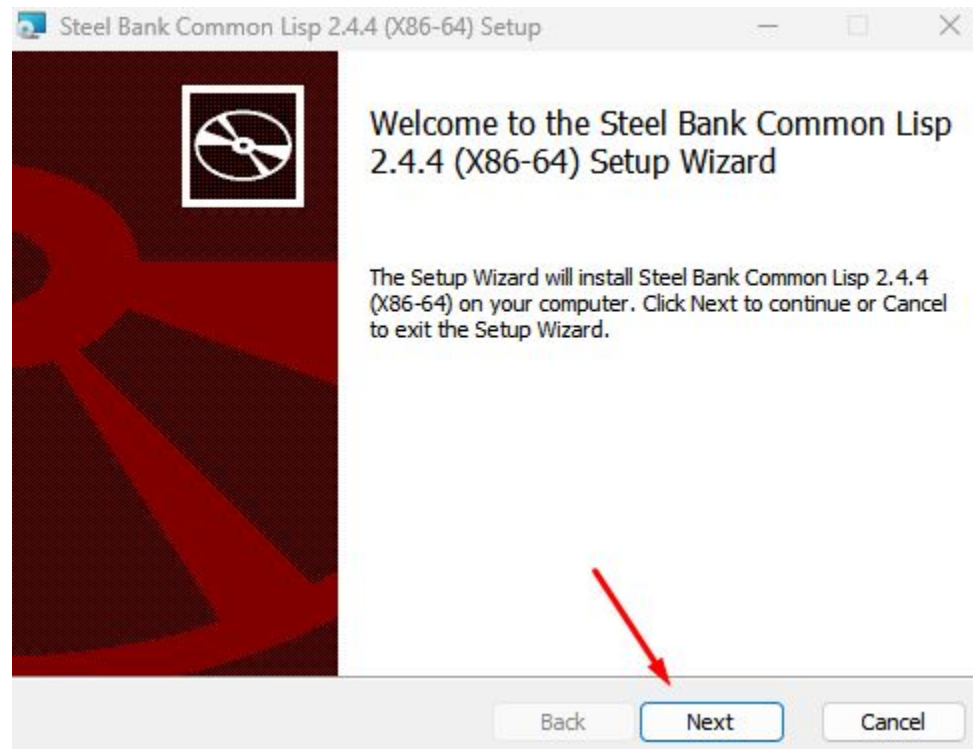
[Get Updates](#) [Share This](#)

[Linux](#) | [Mac](#) | [Windows](#)

Instalando no VS Code

Agora que você tem o instalador, inicie a instalação do SBCL.

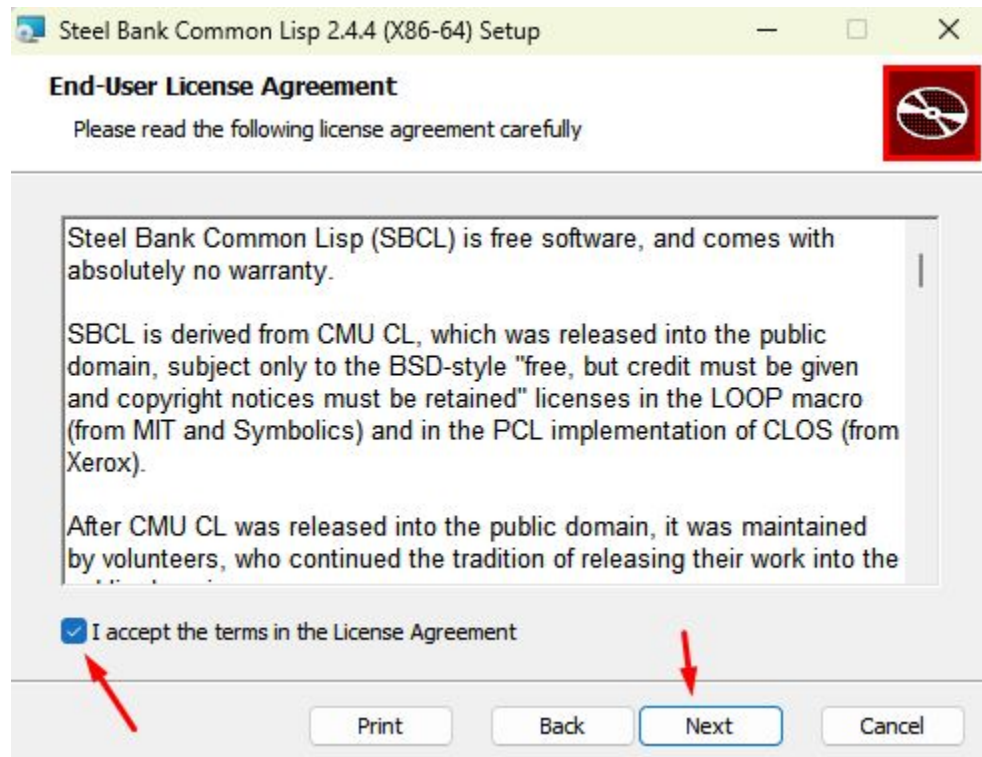
Siga o passo a passo a seguir:



Instalando no VS Code

Agora que você tem o instalador, inicie a instalação do SBCL.

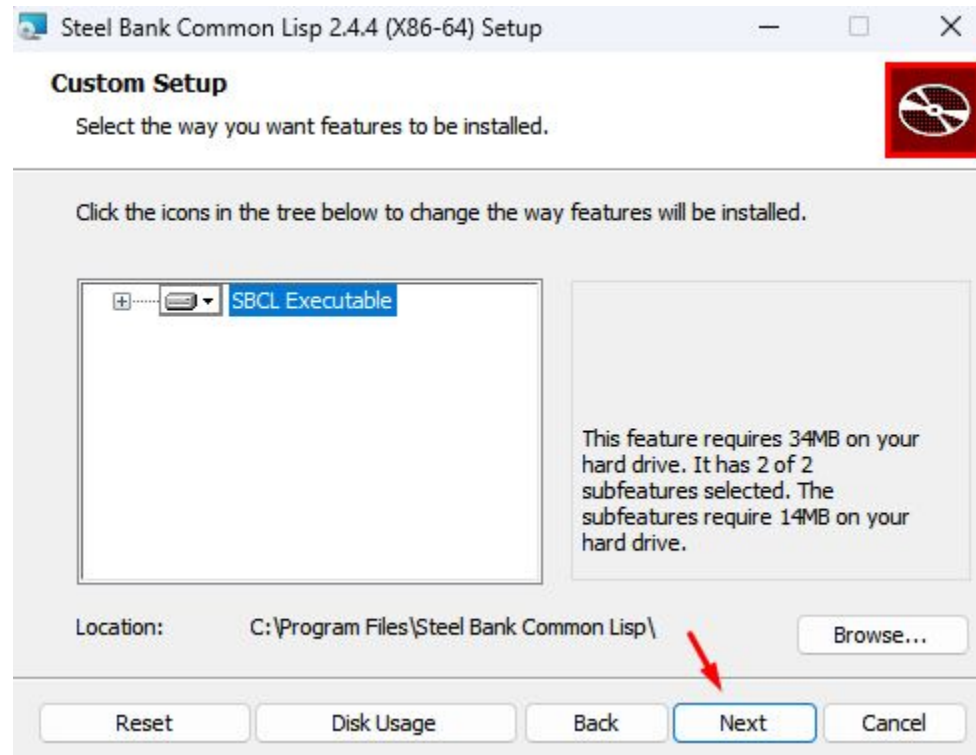
Siga o passo a passo a seguir:



Instalando no VS Code

Agora que você tem o instalador, inicie a instalação do SBCL.

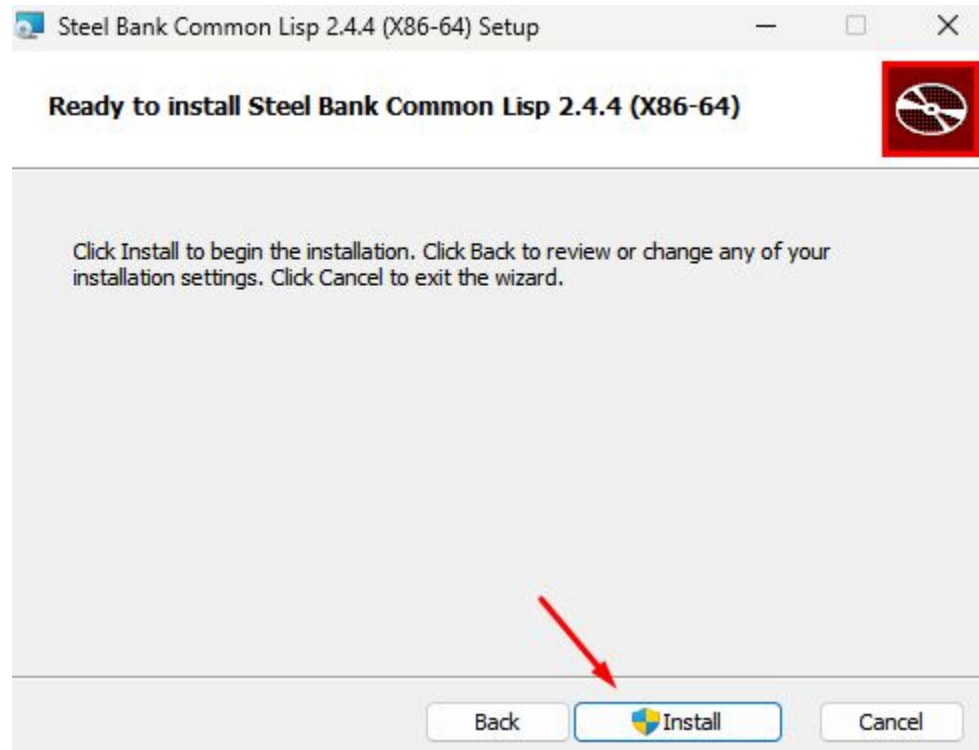
Siga o passo a passo a seguir:



Instalando no VS Code

Agora que você tem o instalador, inicie a instalação do SBCL.

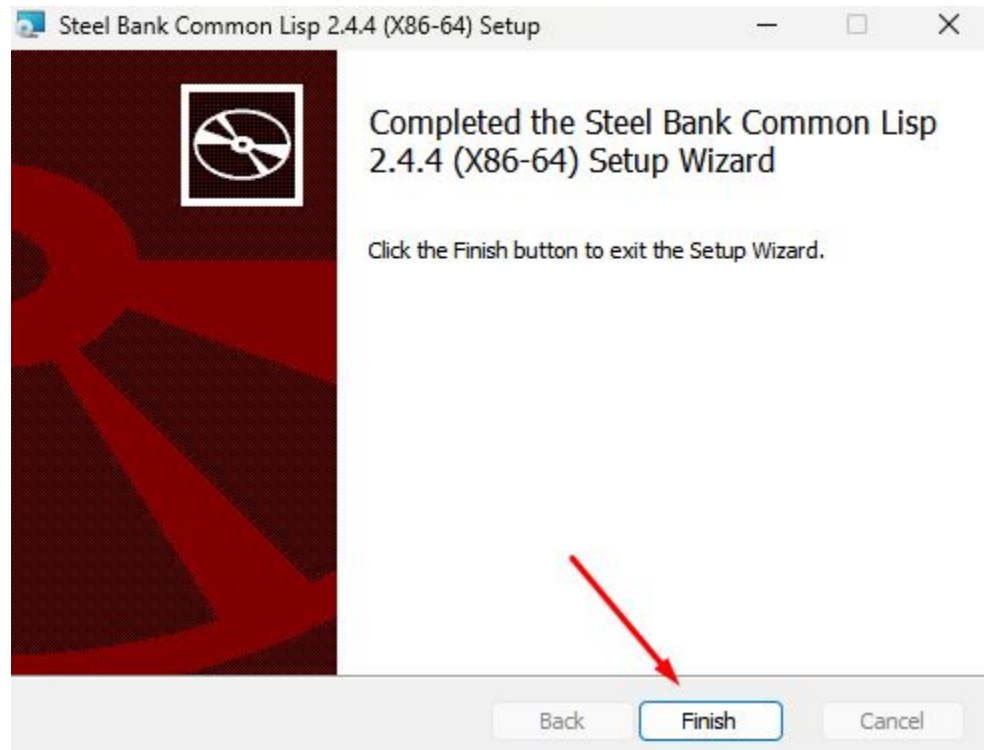
Siga o passo a passo a seguir:



Instalando no VS Code

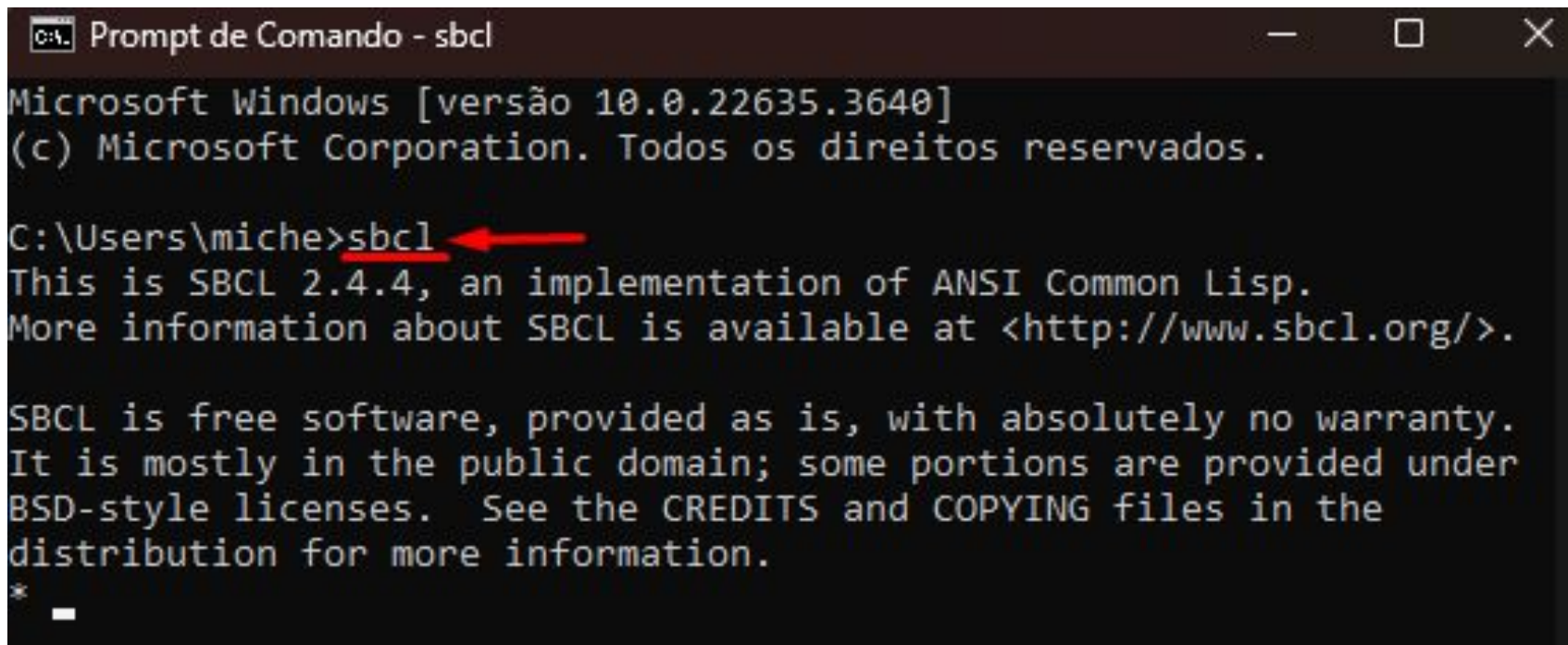
Agora que você tem o instalador, inicie a instalação do SBCL.

Siga o passo a passo a seguir:



Instalando no VS Code

Após a instalação, você pode verificar se o SBCL foi instalado com sucesso no prompt de comando, basta digitar “sbcl” e apertar a tecla enter, deve aparecer a seguinte mensagem.



```
Prompt de Comando - sbcl
Microsoft Windows [versão 10.0.22635.3640]
(c) Microsoft Corporation. Todos os direitos reservados.

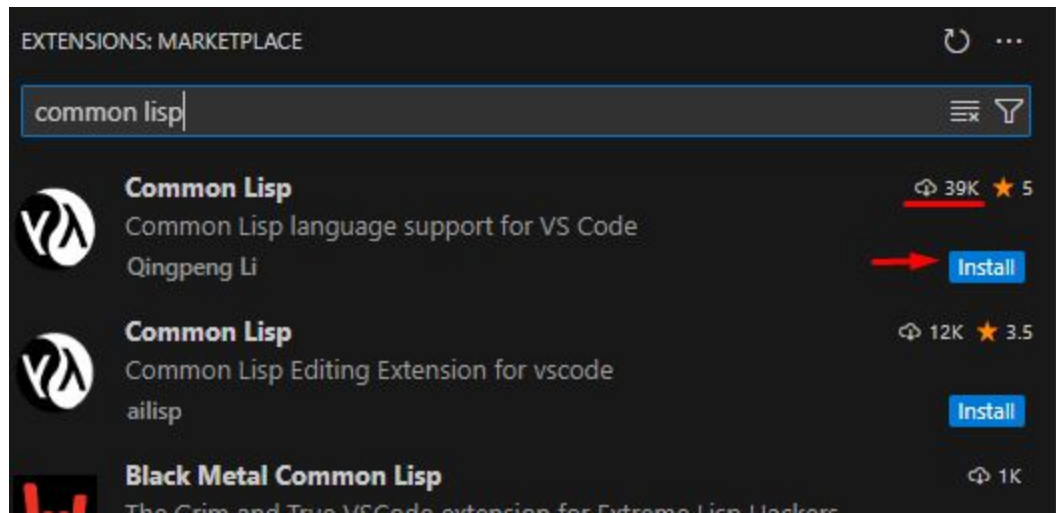
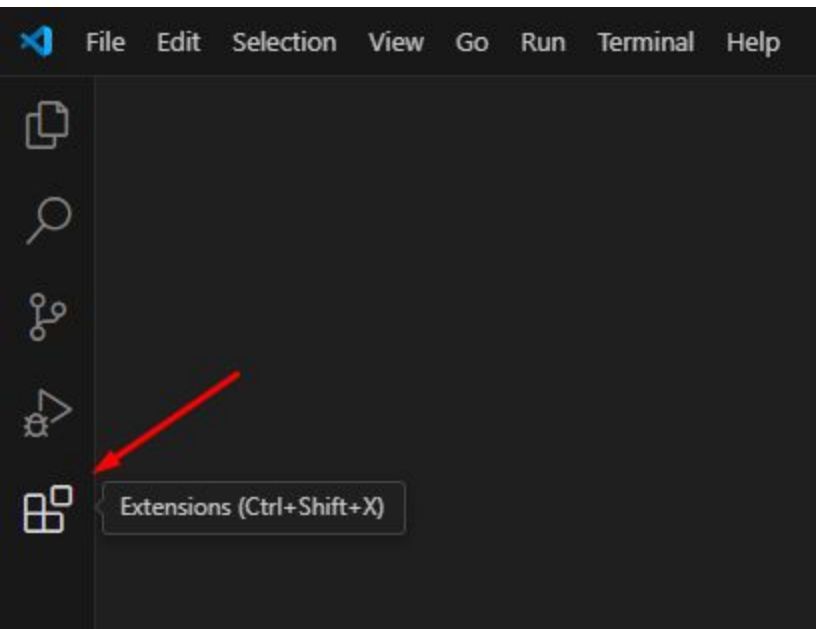
C:\Users\miche>sbcl
This is SBCL 2.4.4, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses. See the CREDITS and COPYING files in the
distribution for more information.
*
```

Instalando no VS Code

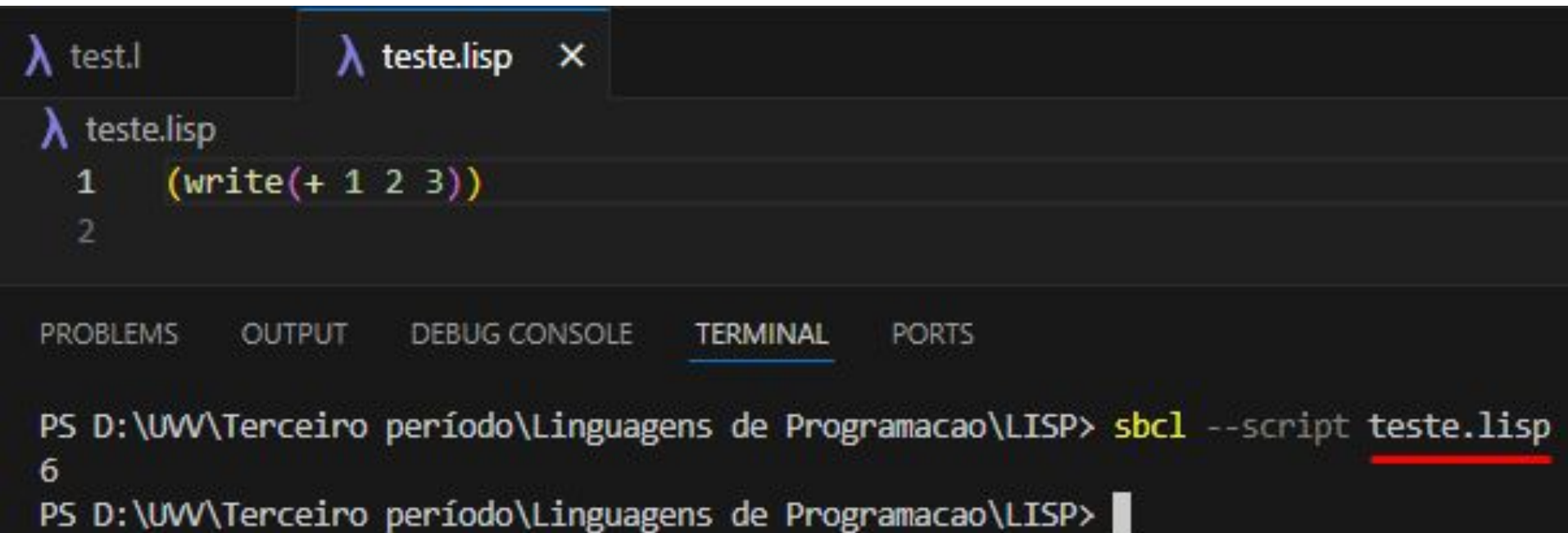
Agora abra o VS Code e clique na área de extensões ou aperte (Ctrl + Shift + X).

Procure pela extensão chamada "Common Lisp" criada por Qingpeng Li e adicione ao seu VS Code.



Instalando no VS Code

Após adicionar a extensão, basta criar um arquivo lisp, a extensão do arquivo pode terminar em ".l" ou ".lisp".



The image shows a screenshot of the Visual Studio Code (VS Code) interface. At the top, there are two tabs: 'test.l' and 'teste.lisp'. The 'teste.lisp' tab is active, showing a single line of Lisp code: `(write(+ 1 2 3))`. Below the editor, there is a panel with four tabs: 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', and 'TERMINAL'. The 'TERMINAL' tab is selected, displaying a command prompt window. The prompt shows the command `sbcl --script teste.lisp` being executed, with the output `6` displayed on the next line. The prompt is currently at the start of a new line, ready for another command.

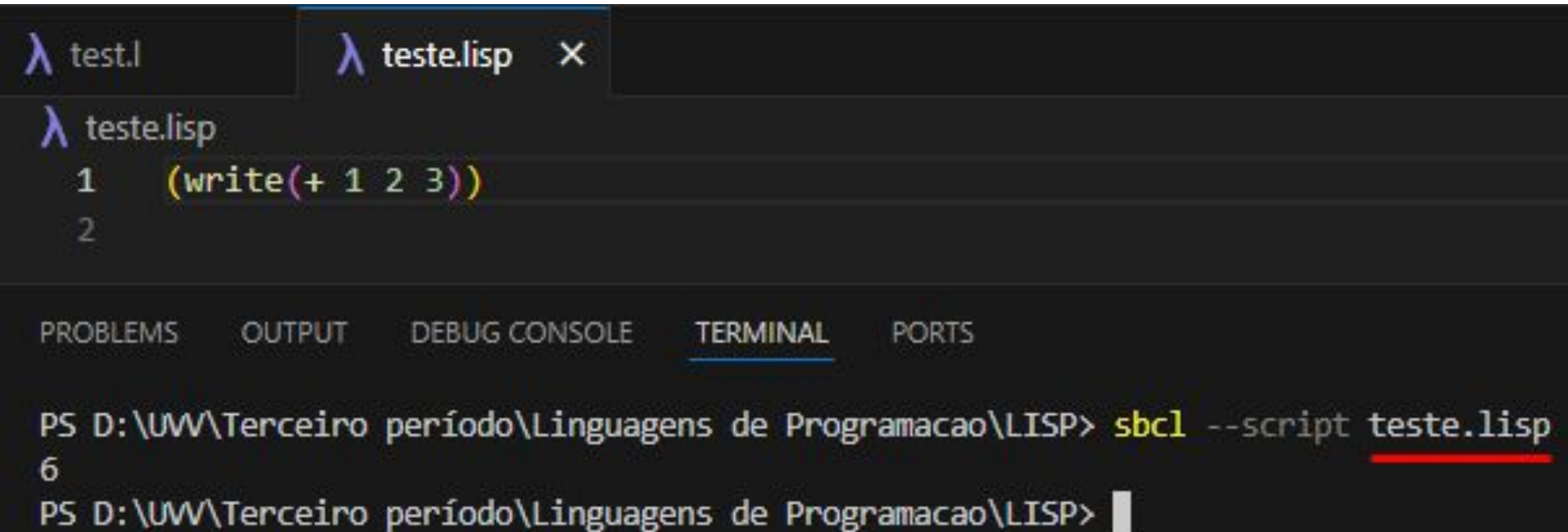
```
λ test.l
λ teste.lisp ×
λ teste.lisp
1  (write(+ 1 2 3))
2

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS D:\UW\Terceiro período\Linguagens de Programacao\LISP> sbcl --script teste.lisp
6
PS D:\UW\Terceiro período\Linguagens de Programacao\LISP> 
```

Instalando no VS Code

Para rodar o programa criado, é necessário abrir o terminal do Visual Studio e inserir o seguinte comando “sbcl --script NomedoArquivo.extensão”. Ex: sbcl --script teste.lisp



The image shows a Visual Studio Code interface with a dark theme. At the top, there are two tabs: 'test.l' and 'teste.lisp'. The 'teste.lisp' tab is active and shows the following code:

```
λ teste.lisp
1  (write(+ 1 2 3))
2
```

Below the editor, the 'TERMINAL' panel is open, showing the command prompt. The command 'sbcl --script teste.lisp' has been entered, and the output '6' is displayed. The file 'teste.lisp' is underlined in red in the command.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS D:\UW\Terceiro período\Linguagens de Programacao\LISP> sbcl --script teste.lisp
6
PS D:\UW\Terceiro período\Linguagens de Programacao\LISP> |
```

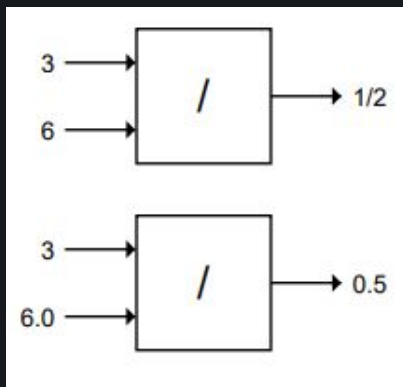
Exemplos de código e

Introdução ao Common Lisp

Introdução aos Tipos de Dados

Números

O tipo mais comum de data em lisp, um número, ele pode ser representado com Inteiros, float (ponto flutuante) e Razão.



Símbolos

Símbolos é o segundo tipo de dados comum em Common Lisp. Um símbolo pode ser praticamente qualquer caractere ou combinação de caracteres

```
X          ZORCH
BANANAS    R2D2
COMPUTER   WINDOW-WASHER
LORETTA    WARP-ENGINES
ABS        GARBANZO-BEANS
YEAR-TO-DATE  BEEBOP

          and even

          ANTIDISESTABLISHMENTARIANISM
```

Símbolos Especiais

NIL = Falso. É também o mesmo que uma lista vazia ().

T = Verdadeiro. Tudo que não é NIL é considerado T em Common Lisp.

Números

X

Símbolos

Uma sequência de dígitos “0” a “9”, opcionalmente precedido por um sinal de mais ou menos.

123 e 30.5 são exemplos de números.

Qualquer sequência de letras, dígitos e números permitidos.

BANANA e 123.456-78 são exemplos de símbolos

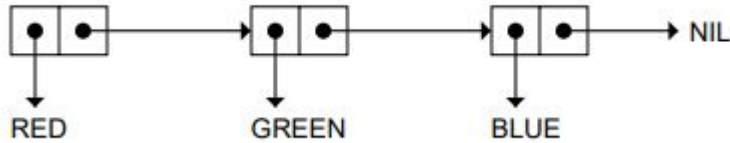
Quando queremos escrever um código em LISP, devemos declarar o operador que queremos utilizar, juntamente com os parâmetros que serão utilizados na operação.

```
2 CL-USER> (+ 1 1)
3 2
4 CL-USER> (- 2 1 1)
5 0
6 CL-USER> (* 2 2)
7 4
8 CL-USER> (/ 2 2)
9 1
```

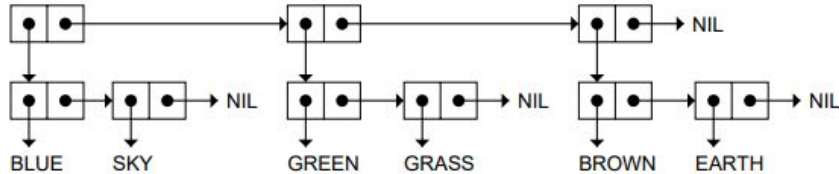
Repare que, quanto mais valores colocarmos, mais operações serão feitas, sempre da esquerda para a direita, por exemplo, o (- 2 1 1), que em “Linguagem Humana” resultaria em:

2 - 1 - 1, que é o mesmo que 2 - 2, resultando em 0.

Estrutura de Listas



(RED GREEN BLUE)



((BLUE SKY) (GREEN GRASS) (BROWN EARTH))

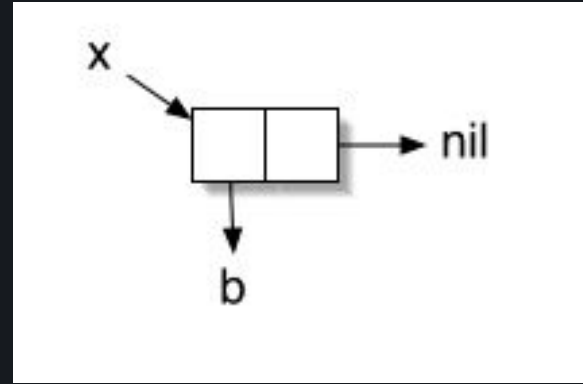
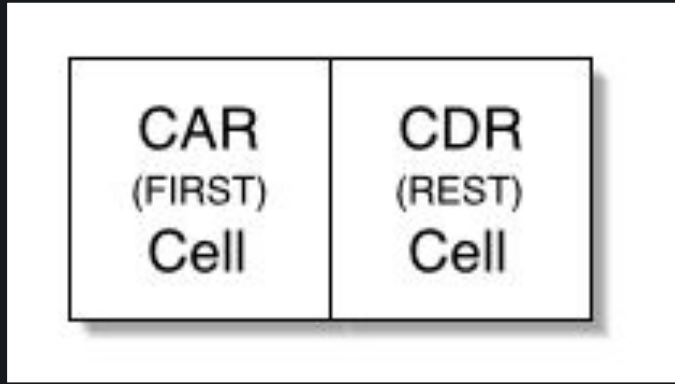
Uma lista em Lisp existe através da organização e criação de Cons Cells, uma estrutura de dados com dois ponteiros.

o primeiro ponteiro aponta para o dado armazenado naquela Cons Cell.

O segundo ponteiro aponta para a próxima cons cell da lista.

O segundo ponteiro da última cons cell de uma lista normalmente aponta para NIL, indicando o fim da lista.

Estrutura de uma Cons Cell



Cada metade de uma Cons Cell tem um nome específico:

- CAR: O ponteiro que aponta para o primeiro elemento da cons cell.
- CDR: O ponteiro que aponta para o “resto” da lista.

Agora para criarmos uma lista de elementos utilizaremos a função `cons` ou `list`, dependendo de como desejamos:

- **CONS:** Coloca um elemento na frente de uma lista existente pela anexação de uma nova Cons cell no início da lista.
- **LIST:** Cria uma lista de elementos ou de listas pela criação de uma ou mais Cons cells.

```
10 CL-USER> (cons 'um '(dois tres))
11 (UM DOIS TRES)
12 CL-USER> (list 'um 'dois 'tres)
13 (UM DOIS TRES)
```

Funções CAR e CDR

Existem duas funções no Lisp chamadas CAR e CDR, não a serem confundidas com o CAR e o CDR de uma cons cell.

Elas são chamadas assim pois seu retorno é semelhante ao que o CAR e o CDR de uma cons cell representam:

- A função CAR sempre retorna o primeiro elemento de uma lista.
- A função CDR sempre retorna o resto da lista, excluindo o primeiro elemento.

Vale notar que o retorno de CDR sempre será uma lista, nem que seja uma lista vazia (NIL).

```
14 CL-USER> (car '(abobora melancia banana manga))
15 ABOBORA
16 CL-USER> (cdr '(abobora melancia banana manga))
17 (MELANCIA BANANA MANGA)
```

Também podemos concatenar CAR e CDR, sempre lendo da direita para esquerda, por exemplo, CADR é um CDR e depois um CAR, enquanto CDAR é um CAR e depois um CDR.

```
40 CL-USER> (cadr '((abobora melancia) (banana manga)))  
41 (BANANA MANGA)  
42 CL-USER> (car (cdr '((abobora melancia) (banana manga))))  
43 (BANANA MANGA)  
44 CL-USER> (cdar '((abobora melancia) (banana manga)))  
45 (MELANCIA)  
46 CL-USER> (cdr (car '((abobora melancia) (banana manga))))  
47 (MELANCIA)
```

Abaixo faremos uma função pura de segundo grau simples, conforme a fórmula padrão comumente conhecida:

$$x = \frac{-b \pm \sqrt{\Delta}}{2a}; \quad \Delta = b^2 - 4ac$$

Caso fizéssemos em C, resultaria no seguinte código:

```
#include <math.h>
#include <stdio.h>

void func(float a, float b, float c, float *x1, float *x2);

int main(void) {
    float a = 1, b = -1, c = -12;
    float x1, x2;
    func(a, b, c, &x1, &x2);
    printf("O valor de x1: %.2f\n", x1);
    printf("O valor de x1: %.2f\n", x2);
    return 0;
}

void func(float a, float b, float c, float *x1, float *x2) {
    float delta = pow(b, 2) - 4 * a * c;
    *x1 = (-b + sqrt(delta)) / 2 * a;
    *x2 = (-b - sqrt(delta)) / 2 * a;
}
```

Enquanto em Lisp, resultaria em uma expressão mais simplificada

```
(defun calcularDelta (a b c)
  (let* (
    (delta (- (expt b 2) (* 4 a c)))
    (x1 (/ (+ (- b) (sqrt delta)) (* 2 a)))
    (x2 (/ (- (- b) (sqrt delta)) (* 2 a)))
  )
  (values x1 x2)
)
```

```
CL-USER> (calculardelta 1 2 -15)
3.0
-5.0
```


Metaprogramação e Capacidade de IA

Metaprogramação

Utilizando metaprogramação o código pode se auto examinar e modificar sua própria estrutura de forma dinâmica, em vez de depender apenas da estrutura fixa definida no momento da compilação.

Um exemplo de uso de metaprogramação em Lisp seria a criação de um novo tipo de dado, como uma lista circular.

É possível utilizar metaprogramação para criar uma sintaxe especial para trabalhar com essa estrutura de dados de forma mais conveniente.

Por exemplo, ao invés de escrever (inserir-elemento-na-lista-circular elemento lista) toda vez que se deseja inserir um elemento, é possível criar uma macro que permita escrever (push elemento lista).

Isso pode ser feito mais facilmente devido ao suporte do Lisp em relação a metaprogramação.

```
(defstruct circular-list
  (head nil)
  (tail nil))
```

```
(defmacro push (element list)
  `(setf (circular-list-head ,list) (cons ,element
(circular-list-head ,list)))
  `(when (null (circular-list-tail ,list))
    (setf (circular-list-tail ,list) (circular-list-head ,list))))
```

```
(defmacro pop (list)
  (let ((element (car (circular-list-head ,list))))
    `(setf (circular-list-head ,list) (cdr (circular-list-head ,list)))
    `(when (null (circular-list-head ,list))
      (setf (circular-list-tail ,list) nil))
    element))
```

O que ele faz de verdade?

```
(defparameter minha-lista (make-circular-list))
```

```
(push 1 minha-lista)
```

```
(push 2 minha-lista)
```

```
(push 3 minha-lista)
```

```
(pop minha-lista) ; retorna 3
```

```
(pop minha-lista) ; retorna 2
```

```
(pop minha-lista) ; retorna 1
```

```
(pop minha-lista) ; retorna nil (lista vazia)
```

IA em Lisp

Utilizando metaprogramação e outras funcionalidades, o Lisp se torna uma das mais poderosas linguagens para se usar para IA.

O código é complexo demais para um nível introdutório, então apenas fica *documentado* o exemplo para que seja possível entender que Lisp é poderoso para implementar IAs.

“Neste exemplo, a função `train-neural-network` define uma rede neural simples com dois nós ocultos e um nó de saída e a treina usando um algoritmo de retropropagação. A função recebe como entrada um conjunto de dados de treinamento, resultados alvo, uma taxa de aprendizado e várias iterações de treinamento. A saída da função são os pesos da rede treinada.”

```
(defun sigmoid (x)
  (/ 1 (+ 1 (exp (- x)))))

(defun train-neural-network (inputs targets learning-rate iterations)
  (let* ((num-inputs (length (first inputs)))
        (num-hidden 2)
        (num-outputs 1)
        (hidden-layer (make-array num-hidden :initial-element 0.0))
        (output-layer (make-array num-outputs :initial-element 0.0))
        (hidden-weights (make-array (list num-inputs num-hidden) :initial-element 0.0))
        (output-weights (make-array (list num-hidden num-outputs) :initial-element 0.0)))
    ;; initialize weights randomly
    (dotimes (i (array-total-size hidden-weights))
      (setf (aref hidden-weights i) (random 1.0)))
    (dotimes (i (array-total-size output-weights))
      (setf (aref output-weights i) (random 1.0)))

    ;; train the network
    (dotimes (i iterations)
      (dotimes (j (length inputs))
        (let* ((input (aref inputs j))
              (target (aref targets j))
              (hidden-inputs (dot-product input hidden-weights))
              (hidden-outputs (map 'vector #'sigmoid hidden-inputs))
              (output-input (dot-product hidden-outputs output-weights))
              (output (sigmoid output-input))
              (output-error (- target output))
              (output-gradient (* output-error output (* (- 1 output))))
              (hidden-error (dot-product output-gradient (transpose output-weights)))
              (hidden-gradient (map 'vector #'* hidden-outputs (- 1 hidden-outputs) hidden-error)))
          (dotimes (k num-outputs)
            (setf (aref output-layer k) output))
          (dotimes (k num-hidden)
            (setf (aref hidden-layer k) (aref hidden-outputs k)))
          (dotimes (k num-outputs)
            (dotimes (l num-hidden)
              (let ((delta (* output-gradient (aref hidden-outputs l))))
                (setf (aref output-weights l k) (+ (aref output-weights l k) (* learning-rate delta))))))
            (dotimes (k num-hidden)
              (dotimes (l num-inputs)
                (let ((delta (* (aref hidden-gradient k) (aref input l))))
                  (setf (aref hidden-weights l k) (+ (aref hidden-weights l k) (* learning-rate delta))))))
              (list hidden-weights output-weights)))))
```

Curiosidades

Site que mostra onde se é usado
Common Lisp hoje em dia.

Table of contents
» North America
» United States
» Canada
» Mexico
» Europe
» Belgium
» Denmark
» England
» Finland
» France
» Germany
» Netherlands
» Norway
» Poland
» Portugal
» Spain
» Sweden
» Switzerland
» Africa
» South Africa
» Asia
» India
» Japan
» Australia

Companies using Lisp

North America

United States

- [Acceleration](#)
- Acceleration is a leading provider of business class IT solutions: high-speed internet access services, network design and managed support, website design, hosting, dedicated and virtual server hosting. They have a [Github account](#).
- [Alphacet, LLC](#) 🤖
- software tools for analysts, traders, and portfolio managers. Stamford, CT. Includes post-sales work. Clients [expand proprietary trading](#) libraries in Lisp.
- *Disappeared as of January 2018.*
- Boeing
- uses the Allegro NFS Server in the onboard network system of Boeing 747 and 777 aircraft (source: [2015](#)).
- [Clozure Associates](#)
- software development consulting firm specializing in Common Lisp development. Somerville, MA. S and developers of OpenMCL.
- *Active as of January 2018.*
- [Eaton Vance](#)
- financial services. Boston, MA. Bruce Lewis created [BRL](#), a dialect of Scheme called "a database-oriented language to embed in HTML and other markup."
- *Active as of January 2018*
- [Franz Inc.](#)
- offers AllegroCL, a Common Lisp compiler and programming environment; [AllegroGraph](#), a graph database triple store written in Common Lisp; and consultancy services. Oakland, CA.
- *Active as of September 2018*
- [Genworks](#)
- provides General-purpose Declarative Language (GDL), a Generative Application Development system for creating web-centric Knowledge Based Engineering and Business applications, based on the free software [Gendl](#). Genworks uses AllegroCL, its Head of Product Development is the founder of the Common Lisp foundation.
- [GrammarTech](#)
- Source code analysis tools for C, C++, Ada. Ithaca, NY. [Hiring software engineers](#) and interns with experience programming in Scheme, compilers, and static analysis.
- *Active as of January 2018*
- [GraphMetricx](#)
- automation of document extraction and publishing for construction, property and logistics.

Questões

Teóricas e de Código



Questão 1

Leia o texto abaixo e assinale a alternativa correta.

"No paradigma funcional, as funções puras são essenciais. Funções puras são aquelas que não têm efeitos colaterais e dependem exclusivamente de seus argumentos de entrada para produzir um resultado."

- a) Funções puras no paradigma funcional podem alterar o estado global do programa.
- b) Funções puras no paradigma funcional podem retornar resultados diferentes para os mesmos argumentos de entrada.
- c) Funções puras no paradigma funcional não recebem entradas e são dependentes do estado global do programa.
- d) Funções puras no paradigma funcional não afetam o estado global do programa e seu resultado é previsível.

Questão 1

Leia o texto abaixo e assinale a alternativa correta.

"No paradigma funcional, as funções puras são essenciais. Funções puras são aquelas que não têm efeitos colaterais e dependem exclusivamente de seus argumentos de entrada para produzir um resultado."

- a) Funções puras no paradigma funcional podem alterar o estado global do programa.
- b) Funções puras no paradigma funcional podem retornar resultados diferentes para os mesmos argumentos de entrada.
- c) Funções puras no paradigma funcional não recebem entradas e são dependentes do estado global do programa.
- d) Funções puras no paradigma funcional não afetam o estado global do programa e seu resultado é previsível.

Questão 2

Qual das seguintes afirmações sobre funções puras está incorreta?

- a) Funções puras sempre retornam o mesmo resultado para os mesmos argumentos
- b) Funções puras podem ter efeitos colaterais
- c) Funções puras não dependem de nenhum estado externo
- d) Funções puras não modificam variáveis externas

Questão 2

Qual das seguintes afirmações sobre funções puras está incorreta?

- a) Funções puras sempre retornam o mesmo resultado para os mesmos argumentos
- b) Funções puras podem ter efeitos colaterais**
- c) Funções puras não dependem de nenhum estado externo
- d) Funções puras não modificam variáveis externas

Questão 3

Qual a principal diferença entre programação funcional e programação imperativa?

- a) Programação funcional utiliza mutabilidade, enquanto a imperativa utiliza imutabilidade
- b) Programação funcional foca em funções e seus retornos, enquanto a imperativa foca em comandos e mudanças de estado
- c) Programação funcional utiliza apenas laços estruturais, enquanto a imperativa utiliza apenas recursão
- d) Programação funcional é orientada a objetos, enquanto a imperativa não é

Questão 3

Qual a principal diferença entre programação funcional e programação imperativa?

- a) Programação funcional utiliza mutabilidade, enquanto a imperativa utiliza imutabilidade
- b) Programação funcional foca em funções e seus retornos, enquanto a imperativa foca em comandos e mudanças de estado**
- c) Programação funcional utiliza apenas laços estruturais, enquanto a imperativa utiliza apenas recursão
- d) Programação funcional é orientada a objetos, enquanto a imperativa não é

Questão 4

Considerando o código em Common Lisp, qual o retorno das seguintes expressões? E quais causam um erro?

```
; ITENS:  
(+ 3 5)  
(3 + 5)  
(+ 3 (5 6))  
(+ 3 (* 5 6))  
'(manha tarde noite)  
('manha 'tarde 'noite)  
(list 'manha 'tarde 'noite)  
(car nil)  
(+ 3 banana)  
(+ 3 'banana)
```

Questão 4

Considerando o código em Common Lisp, qual o retorno das seguintes expressões? E quais causam um erro?

```
; ITENS:  
(+ 3 5)  
(3 + 5)  
(+ 3 (5 6))  
(+ 3 (* 5 6))  
'(manha tarde noite)  
( 'manha 'tarde 'noite)  
(list 'manha 'tarde 'noite)  
(car nil)  
(+ 3 banana)  
(+ 3 'banana)
```

```
RESPOSTAS:  
; 8  
; ERRO  
; ERRO  
; 33  
; (manha tarde noite)  
; ERRO  
; (manha tarde noite)  
; NIL  
; ERRO  
; ERRO
```

Questão 5

O paradigma de programação funcional tem como principal conceito de programação a abordagem das estruturas de dados do programa como funções matemáticas. A respeito do paradigma funcional, assinale a alternativa correta.

- a) Funções podem ser passadas como argumento e retornadas como resultado, mas não podem ser guardadas como valores em variáveis, tampouco armazenadas como componentes de estruturas de dados maiores.
- b) Ter funções como cidadãos de primeira classe em uma linguagem funcional implica não ser possível especificar um valor funcional sem dar um nome a ela.
- c) Uma função anônima (lambda) é uma expressão funcional que não especifica a relação entre entrada e saída.
- d) Uma diferença do paradigma funcional em relação à imperativa é o que costuma ser chamado de transparência referencial: cada parte do programa funcional sempre tem o mesmo resultado, independentemente do contexto em que ele se encontra.

Questão 5

O paradigma de programação funcional tem como principal conceito de programação a abordagem das estruturas de dados do programa como funções matemáticas. A respeito do paradigma funcional, assinale a alternativa correta.

- a) Funções podem ser passadas como argumento e retornadas como resultado, mas não podem ser guardadas como valores em variáveis, tampouco armazenadas como componentes de estruturas de dados maiores.
- b) Ter funções como cidadãos de primeira classe em uma linguagem funcional implica não ser possível especificar um valor funcional sem dar um nome a ela.
- c) Uma função anônima (lambda) é uma expressão funcional que não especifica a relação entre entrada e saída.
- d) Uma diferença do paradigma funcional em relação à imperativa é o que costuma ser chamado de **transparência referencial**: cada parte do programa funcional sempre tem o mesmo resultado, independentemente do contexto em que ele se encontra.

Questão 6

Considerando o código em Common Lisp, qual o retorno e qual seria um nome apropriado para a função?

```
(defun CODIGOSECRETO (n)
  (if (<= n 1)
      1
      (* n (CODIGOSECRETO (- n 1)))))
(CODIGOSECRETO 4)
```

Questão 6

Considerando o código em Common Lisp, qual o retorno e qual seria um nome apropriado para a função?

```
(defun CODIGOSECRETO (n)
  (if (<= n 1)
      1
      (* n (CODIGOSECRETO (- n 1)))))
(CODIGOSECRETO 4)
```

O retorno seria 24 e o nome apropriado seria 'fatorial'

Questão 7

Em programação funcional, o que significa "composição de funções"?

- a) Combinar várias listas em uma única lista através de uma função
- b) Definir uma função dentro de outra função
- c) Aplicar uma função sobre o resultado de outra função
- d) Modificar variáveis diretamente através de funções

Questão 7

Em programação funcional, o que significa "composição de funções"?

- a) Combinar várias listas em uma única lista através de uma função
- b) Definir uma função dentro de outra função
- c) Aplicar uma função sobre o resultado de outra função**
- d) Modificar variáveis diretamente através de funções

Questão 8

Sobre funções de ordem superior, quais das seguintes afirmativas são corretas?

I. Funções de ordem superior podem aceitar outras funções como argumentos.

II. Funções de ordem superior não podem retornar outras funções.

III. Funções de ordem superior são comuns em programação funcional.

a) I e II

b) I e III

c) II e III

d) Todas estão corretas

Questão 8

Sobre funções de ordem superior, quais das seguintes afirmativas são corretas?

I. Funções de ordem superior podem aceitar outras funções como argumentos.

II. Funções de ordem superior não podem retornar outras funções.

III. Funções de ordem superior são comuns em programação funcional.

a) I e II

b) I e III

c) II e III

d) Todas estão corretas

Questão 9

O que é uma função lambda em programação funcional?

- a) Uma função que modifica variáveis globais
- b) Uma função definida sem um nome
- c) Uma função que sempre retorna o mesmo valor
- d) Uma função que encapsula estado

Questão 9

O que é uma função lambda em programação funcional?

- a) Uma função que modifica variáveis globais
- b) Uma função definida sem um nome**
- c) Uma função que sempre retorna o mesmo valor
- d) Uma função que encapsula estado

Questão 10

Considerando o código em Common Lisp, o que essa função faz?

```
(defun FUNCAO-DESCONHECIDA (n)
  (if (> n 0) (+ n 1) (- n 1)))
```

Questão 10

Considerando o código em Common Lisp, o que essa função faz?

```
(defun FUNCAO-DESCONHECIDA (n)
  (if (> n 0) (+ n 1) (- n 1)))
```

Ela afasta a entrada de zero.

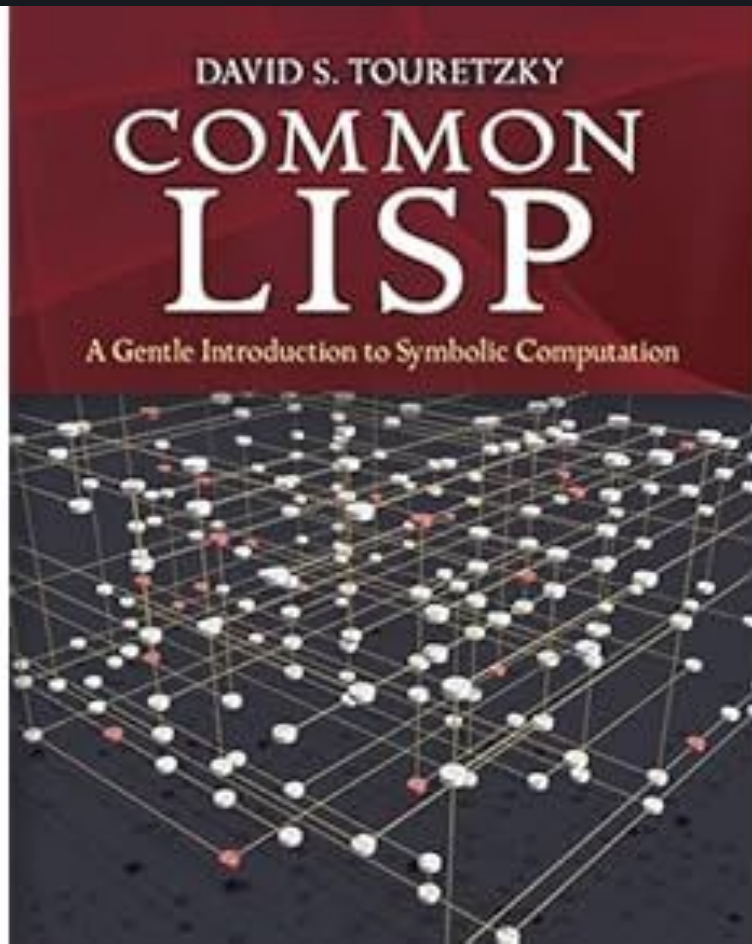
Se a entrada for positiva, soma 1.

Se a entrada for negativa, subtrai 1.

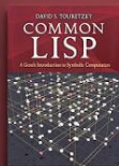
A função considera que se o valor for 0, ela vai subtrair 1, se distanciando para os negativos.

Bibliografia

Livros



Vídeos Recomendados:



Aprenda LISP 0 Introdução

Estudo do livro "Common LISP: A Gentle Introduction to Symbolic...









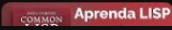
Computação Raiz

56 vídeos 7.204 visualizações Última atualização em 2...



▶ Reproduzir tu...

🔀 Ordem aleató...

- **1**
Funções e Dados
8:28
Funções e Dados: Introdução (Common Lisp: A Gentle Introduction to Symbolic Computation)
Computação Raiz • 1,3 mil visualizações • há 2 anos
- **2**
Funções Numéricas
8:06
Funções Numéricas (Common Lisp: A Gentle Introduction to Symbolic Computation)
Computação Raiz • 637 visualizações • há 2 anos
- **3**
Tipos Numéricos
10:40
Três Tipos de Números (Common Lisp: A Gentle Introduction to Symbolic Computation)
Computação Raiz • 514 visualizações • há 2 anos
- **4**
Ordem dos Inputs
4:26
A Ordem dos Inputs é Importante (Common Lisp: A Gentle Introduction to Symbolic Computation)
Computação Raiz • 377 visualizações • há 2 anos
- **5**
Exercícios
9:26
Exercício 1.1. Caixas de Funções com Inputs e Outputs (Common Lisp: A Gentle Introduction to...)
Computação Raiz • 371 visualizações • há 2 anos
- **6**
Símbolos
11:30
Símbolos (Common Lisp: A Gentle Introduction to Symbolic Computation)
Computação Raiz • 392 visualizações • há 2 anos
- **7**
Exercícios
4:03
Exercício 1.2. Símbolo ou Número? (Common Lisp: A Gentle Introduction to Symbolic Computation)
Computação Raiz • 277 visualizações • há 2 anos
- **8**
T e NIL
3:57
Os Símbolos Especiais: T e NIL (Common Lisp: A Gentle Introduction to Symbolic Computation)
Computação Raiz • 312 visualizações • há 2 anos
- **9**
Alguns Predicados Simples (Common Lisp: A Gentle Introduction to Symbolic Computation)

Sites

<https://www.locaweb.com.br/blog/temas/codigo-aberto/programacao-funcional-e-poo-veja-as-diferencas/>

[https://community.revelo.com.br/as-quatro-linguagens-maes-lisp/#:~](https://community.revelo.com.br/as-quatro-linguagens-maes-lisp/#:~:text=funcional,programa%C3%A7%C3%A3o,funcional,parte,1,fa3bff0d6d2d)

<https://sergiocosta.medium.com/paradigma-funcional-3194924a8d20>

<https://www.ic.unicamp.br/~meidanis/courses/mc346/2015s2/funcional/apostila-lisp.pdf>

<https://blog.nubank.com.br/programacao-funcional-o-que-e-relacao-nubank/>

<https://building.nubank.com.br/functional-programming-technology-at-nubank/>

<https://blog.geekhunter.com.br/quais-sao-os-paradigmas-de-programacao/>

<https://sergiocosta.medium.com/princ%C3%ADpios-e-padr%C3%B5es-de-programa%C3%A7%C3%A3o-funcional-parte-1-fa3bff0d6d2d>

<https://medium.com/@marcelomg21/programa%C3%A7%C3%A3o-funcional-teoria-e-conceitos-975375cfb010>

<https://www.treinaweb.com.br/blog/linguagens-e-paradigmas-de-programacao>

<https://tripleten.com.br/blog/paradigmas-de-programacao-o-que-sao-e-quais-os-principais/>

<https://www.alura.com.br/artigos/programacao-funcional-o-que-e>

<https://www.turing.com/kb/introduction-to-functional-programming>

https://pt.wikipedia.org/wiki/John_McCarthy

[https://www.baeldung.com/cs/referential-transparency#:~](https://www.baeldung.com/cs/referential-transparency#:~:text=funcional,programa%C3%A7%C3%A3o,funcional,parte,1,fa3bff0d6d2d)

<https://www.geeksforgeeks.org/tail-vs-non-tail-recursion/>

Obrigado pela atenção

João Victor da Silva Cunha
Lorenzo Simonassi Moura
Lucas Bonato Soares
Marcelo Cunha Lima Nogueira Dessaune
Mateus Melo Fernandes
Matheus Endlich Silveira
Michel Gonçalves Salles
Nicolas Oliveira Goldner
Pedro Henrique Novelli Soares
Pedro Henrique Pimentel da Silva
Pedro Henrique Semensato Denadai

