



Universidade Vila Velha

Paradigma Funcional

Linguagens de Programação - 2024/1

João Victor da Silva Cunha - 202306489

Lorenzo Simonassi Moura - 202306210

Lucas Bonato Soares - 202307653

Marcelo Cunha Lima Nogueira Dessaune - 202305767

Mateus Melo Fernandes - 202306480

Matheus Endlich Silveira - 202305392

Michel Gonçalves Salles - 202305768

Nicolas Oliveira Goldner - 202305844

Pedro Henrique Novelli Soares - 202306364

Pedro Henrique Pimentel da Silva - 202305389

Pedro Henrique Semensato Denadai - 202308371

Professor Abrantes Araújo Silva Filho ¹

¹ Github Do Professor

Sumário

Sumário	2
1	Introdução 4
1.1	A apostila 4
1.2	Como instalar? 4
1.2.1	Portacle 4
1.2.2	VS Code (windows) 4
2	O que é o Paradigma Funcional? 5
2.1	Conceitos 5
2.2	Funções 6
2.2.1	Composição de funções 6
2.2.2	Funções puras 6
2.2.3	Funções de primeira classe 7
2.2.4	Funções de alta ordem 7
2.3	Imutabilidade 7
2.4	Transparência referencial 7
2.5	Recursão 8
2.5.1	Recursão de cauda 9
2.6	Lazy Evaluation 9
2.7	Aplicação 10
2.8	Paradigma funcional puro 11
3	Historia do Paradigma 11
4	Vantagens E Desvantagens 13
4.1	Vantagens 13
4.2	Desvantagens 14
5	Exemplos de Código 15
5.0.1	Introdução aos Tipos de Dados 15
5.0.2	Expressões em Lisp 16
5.0.3	Listas em Lisp 16
5.0.4	Cons cells 17
5.0.5	Funções de manipulação de listas 17
5.0.6	Exemplo de código - Bhaskara 18
5.0.7	Teorema de Pitagoras 19
5.0.8	Meta programação 19
5.0.9	Inteligência Artificial 21
6	Comparativo com Outros paradigmas 22
6.1	Comparativo com imperativo 22

6.1.1	Abordagem	22
6.1.2	Estados do sistema e efeitos colaterais	22
6.1.3	Loops	22
6.1.4	Paralelismo	22
6.2	Comparativo com orientada a objeto	23
6.2.1	Abordagem	23
6.2.2	Estados do sistema e efeitos colaterais	23
6.2.3	Herança e polimorfismo	23
6.2.4	Estrutura do programa	23
6.3	Comparativo com logica	24
6.3.1	Abordagem	24
6.3.2	Forma de Resolução	24
6.3.3	Dados e conhecimento	24
6.3.4	Aplicabilidade	24
7	Questões Produzidas	25
7.1	Questões Teóricas	25
7.2	Questões de código	29

1. INTRODUÇÃO

1.1. A APOSTILA

Esta apostila documenta o que é o paradigma funcional, sua história, vantagens, desvantagens, comparativo com outros paradigmas e algumas questões para praticar. Através de recomendações de colegas e do próprio professor Abrantes, escolhemos usar o Common Lisp como linguagem de referência no material. A apostila tem foco acadêmico e o objetivo é compartilhar conhecimento com a turma sobre esse paradigma.

O próximo tópico tem como foco auxiliar na instalação da linguagem Common Lisp, utilizada nesta apostila.

1.2. COMO INSTALAR?

1.2.1. 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. 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. Para fazer o download do instalador, você precisa clicar em "latest release". Após isso, selecione o local do seu computador aonde deseja salvar o instalador do programa. Executando o instalador, selecione em qual pasta será salvo o programa. Abrindo a pasta, clique para executar o arquivo "portacle.exe" que o programa irá iniciar, estando pronto para ser utilizado.

1.2.2. VS Code (windows)

Caso queira utilizar o LISP no Visual Studio, ainda utilizando o site do Common Lisp, terá que clicar em "Download", na área do Steel Bank Common Lisp (SBCL). Ao clicar, será redirecionado para o site do SBCL. Clique no quadrado verde da versão SBCL que deseja, no nosso caso, Windows AMD64. 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. Agora que você tem o instalador, inicie a instalação do SBCL. 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 versão do SBCL que você está utilizando. Agora abra o VS Code e clique na área de extensões ou aperte (Ctrl + Shift + X). Procure pela extensão chamada "Common Lisp" criada ppt Qingpeng Li e adicione ao seu VS Code. Após adicionar a extensão, basta criar um arquivo lisp, a extensão do arquivo pode terminar em

".lisp" ou ".lisp". 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

2. O QUE É O PARADIGMA FUNCIONAL?

2.1. CONCEITOS

O paradigma funcional pode ser definido como o modo de pensar em soluções computacionais em que a obtenção de respostas pela avaliação de funções é o foco. Isso significa que pensamos em como fazer composições de funções a fim de chegar no resultado que queremos. Não estamos interessados na elaboração de sequências de declarações imperativas ou no estado do sistema, e sim na obtenção de resultados concisos pela chamada de funções.

Essencialmente, o paradigma funcional não pensa em programas contínuos que entregam soluções pela mudança do estado do sistema em tempo de execução. Aqui, destacamos a avaliação de funções. Por analogia, podemos visualizar programas funcionais como calculadoras, onde o usuário faz a entrada de uma expressão, e a calculadora retorna o resultado do processamento dela.

Entenda a entrada do usuário na calculadora como a "chamada de uma função", o processamento que a calculadora realiza como a "avaliação da função", e o resultado exibido pela calculadora como o "retorno da função". Também podemos chamar o próprio retorno de "avaliação da função", pois o retorno é o próprio resultado do processamento.

Mantendo essa linha de raciocínio, é fácil entender que 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 o resultado da função $f(x)$ ao quadrado?", "o número x é primo?" ou "qual o resultado de $\log(x)$?" são bons exemplos de como a interação com o programa ocorre neste paradigma.

Dessa forma, o paradigma funcional se preocupa com a definição de funções determinísticas e de variáveis imutáveis para gerar um ciclo de entradas e saídas confiável e estável. O sistema 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.

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

2.2. FUNÇÕES

Funções são essenciais para a programação funcional, e, por isso, possuem definições específicas dentro do paradigma.

A forma como as funções são tratadas e avaliadas na programação funcional difere um pouco dos outros paradigmas. No paradigma funcional funções possuem características que as aproximam das definições matemáticas, tendendo a serem mais determinísticas e puras. Além disso, funções tem manipulação flexível dentro de programas funcionais.

2.2.1. Composição de funções

A composição de funções se refere a prática de "encadear" funções dentro de um programa. Semelhante ao conceito de *funções compostas* da matemática, a composição de funções é a prática de fazer funções "aninhadas", ou seja, aplicar uma função sobre o retorno de outra função.

De forma simples, podemos visualizar isso pelos conceitos básicos da matemática:

$$(x + (y * z))$$

Observe: X será somado ao *resultado* ou *retorno* da expressão $(y * z)$. Podemos entender esse caso como a composição de duas funções.

A composição de funções não tem limite real: Pode-se aninhar quantas funções quiser. Por isso é comum, dentro do paradigma funcional, a criação de grandes "cadeias" de funções sendo aplicadas uma sobre a outra. É uma das ferramentas mais importantes para tornar a programação dentro do paradigma funcional tão flexível, poderosa e precisa.

2.2.2. Funções puras

Funções puras formam a base da programação funcional e possuem duas propriedades principais:

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

Funções puras funcionam bem com valores imutáveis, pois descrevem como as entradas se relacionam com as saídas precisamente. As funções puras são reutilizáveis, fáceis de organizar, simples de depurar e tornam os programas flexíveis e adaptáveis

às mudanças.

Vale notar que embora efeitos colaterais não existam em funções puras, programadores nem sempre conseguem desenvolver programas ideais. Ou seja, programas funcionais podem ter funções impuras que causem efeitos colaterais por alguma necessidade da solução. Assim, é mais lúcido entender efeitos colaterais não como algo inexistente no paradigma funcional, mas algo indesejável, que deve ser evitado sempre que possível.

2.2.3. 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).

2.2.4. 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 retorna funções como saídas é chamada de função de alta ordem. Essa característica é uma mera consequência do fato que funções são cidadãos da primeira classe no paradigma funcional.

2.3. 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. É boa prática programar cada função para produzir o mesmo resultado, independentemente do estado do programa. Isso significa que quando criamos uma variável e atribuímos um valor, podemos executar o programa com segurança, sabendo que o valor das variáveis permanecerá constante e nunca poderá mudar.

2.4. TRANSPARÊNCIA REFERENCIAL

A transparência referencial é uma propriedade de uma função que permite que ela seja substituída por sua saída equivalente. Em termos mais simples, se você chamar a função uma segunda vez com os mesmos argumentos, terá a garantia de obter o mesmo valor de retorno.

Vamos considerar o seguinte código em Common Lisp:

```
(defun soma (x y)
  (+ x y))
```

O código é simples: Ele recebe dois valores e retorna a soma deles. Agora, considere o seguinte uso da função:

```
(soma 5 4) ; 9
```

A função retorna 9, e, por isso, a função (soma 5 4) pode ser considerada equivalente ao valor 9.

Assim:

```
(/ 90 (soma 5 4)) ; 10
(soma 20 (soma 5 4)) ; 29
(list 'numero (soma 5 4)) ; (numero 9)
```

Podemos utilizar a função (soma 5 4) como se fosse o valor 9. Em qualquer local do código, independente do contexto, podemos considerar (soma 5 4) equivalente ao valor 9, pois sabemos que dados as mesmas entradas, a função sempre retornará a mesma saída. Essa característica é, justamente, a transparência referencial.

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

O exemplo mais clássico e simples de recursão é o da função *fatorial*. Não nativa ao *Common Lisp*, a função fatorial deve ser definida pelo programador. Vamos utilizar recursão para isso:

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

Com isso, a função *fat* retorna o fatorial de um número x qualquer. Sabendo como um fatorial funciona, devemos multiplicar o valor de x por todos os números inteiros menores que ele até o 1. Para isso, multiplicamos o valor de x pelo valor de $x - 1$, sucessivamente, até que a subtração resulte em 1.

A recursão ocorre pela forma como isso é feito em código: No caso onde $x > 1$, a função multiplica x pelo retorno de uma chamada da própria função, tendo como entrada $x = x - 1$.

O caso onde $x \leq 1$ é justamente o *caso base* deste exemplo. Ao alcançar o caso base, a função cessa a recursão e retorna um valor constante, que neste caso é 1.

Considerando a chamada inicial "(fat 3)", podemos visualizar a recursão da seguinte forma:

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

2.5.1. Recursão de 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.

A recursão final de recursividade em cauda pode ser otimizada por compiladores/interpretadores para usar uma quantidade constante de espaço da *Stack*, evitando problemas de memória.

A própria função recursiva fatorial é um exemplo de recursividade de cauda, sendo que ao ser feita em cauda tem uma estrutura um pouco diferente da apresentada. Perceba: A última coisa que a função faz é a chamada recursiva.

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

É perceptível que a sintaxe fica um pouco mais avançada, mas entenda que na prática a recursão ainda acontece, porém pela recursão de cauda.

2.6. 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.

Além disso, lazy evaluation também é conhecida por simplificar o processamento de algumas operações lógicas.

De forma simples, podemos entender isso com o seguinte código Lisp:

```
(setf x (+ 5 3)) ; linha 1
(+ 3 2)          ; linha 2
(+ 2 x)          ; linha 3
```

Sem lazy evaluation, esse código seria entendido como: Faça a avaliação da expressão $(+ 5 3)$ e armazene o resultado na variável de nome x . Nesse caso, o valor 8 seria armazenado na variável de nome x . Com isso, a avaliação da expressão ocorre na **linha 1**.

Com lazy evaluation, esse código seria entendido como: A variável x agora possui como valor a expressão $(+ 5 3)$, que não será avaliada ainda. Nesse caso, a expressão $(+ 5 3)$ seria armazenada na variável de nome x . Apenas quando o valor da variável x for necessário, na linha 3, que a avaliação será feita. Com isso, a avaliação da expressão ocorre na **linha 3**.

Além disso, a lazy evaluation diminui o gasto de processamento em algumas expressões lógicas. Isso pode ser visto na seguinte expressão:

```
; considere termoT = T, e termoF = NIL
(and termoF termoT)
(or termoT termoF)
```

Existem dois termos em ambas expressões - então a avaliação do AND e do OR deveria ter que avaliar dois termos para verificar se é verdadeiro ou falso. Mas isso não acontece.

Na realidade, o programa entende que se um único termo da operação AND for NIL, a operação AND já será NIL, independente das outras entradas. A mesma lógica acontece na avaliação do OR, onde a presença de um único termo T significa que o OR retornará T.

Dessa forma, o programa evita a avaliação de termos desnecessários. Em ambos os exemplos, apenas o primeiro termo do AND e OR são avaliados. Isso significa que sem a lazy evaluation teríamos que avaliar 2 termos para cada operação, enquanto com lazy evaluation avaliamos apenas 1 em cada operação.

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.

2.7. APLICAÇÃO

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

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

Outras linguagens notáveis que não são da família Lisp mas aplicam o paradigma funcional são: **Haskell**, uma linguagem que tenta aplicar o conceito da ”*programação funcional pura*”, **Elixir**, uma linguagem brasileira especialista em escalabilidade e eficiência, e **Rust**, uma linguagem multi paradigma que visa a criação de código seguro.

Como citado anteriormente, estaremos utilizando o Common Lisp como base prática para nossa apostila.

2.8. 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.

Como citado anteriormente, nem sempre é possível programar de forma a não haver efeitos colaterais, tratando-os não como algo impossível mas como algo indesejável. Linguagens funcionais puras, como *Haskell*, já não permitem esse pensamento, impedindo a criação de funções impuras como um todo.

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.

3. HISTORIA DO PARADIGMA

O paradigma funcional, influenciado pela matemática e pelo cálculo lambda de Alonzo Church, surgiu como uma abordagem revolucionária na programação, onde as funções são tratadas como cidadãos de primeira classe. O trabalho de Haskell Curry, especialmente sua teoria da combinação funcional, também desempenhou um papel crucial na fundamentação desse paradigma.

Esse novo modelo de programação foi posteriormente solidificado com o advento do Lisp em 1956, quando John McCarthy introduziu conceitos inovadores, como funções de ordem superior e recursão. Posto isso, a origem do Lisp começou a acontecer 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.

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. Todavia, McCarthy, devido à sua experiência em diversos locais, decidiu criar sua própria linguagem nova, baseando-se tanto na IPL quanto na FORTRAN e na FLPL. 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.

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.

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.

Várias ideias importantes surgiram a partir da conexão com o Lisp. Exemplos incluem a maximização do nível de interpretação e compilação, funções responsivas, depuração, entre outros. Atualmente, o Lisp é conhecido principalmente por sua sofisticada pesquisa em programação funcional, orientação a objetos e estilo de programação paralela.

4. VANTAGENS E DESVANTAGENS

Na área a seguir, descreveremos as vantagens e desvantagens do paradigma funcional, considerando as aplicação comum que esse paradigma possui em suas linguagens mais utilizadas:

4.1. 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.
- **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.
- **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.

4.2. 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.
- **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.
- **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.
- **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.

5. EXEMPLOS DE CÓDIGO

Nessa seção, vamos apresentar um pouco mais sobre a linguagem Common Lisp e mostrar exemplos de sua aplicação na prática em código. Vamos também apenas citar algumas funcionalidades notáveis do Lisp que são complexas demais para serem elaboradas em uma apostila introdutória.

5.0.1. Introdução aos Tipos de Dados

Em Lisp, existem dois tipos principais de dados primitivos: Números e Símbolos. Todo o processo de criar código depende desses dois primitivos, e sem eles o Lisp simplesmente não funciona.

Números: Números em Lisp funcionam como o esperado, representando valores numéricos e podendo serem utilizados em cálculos aritméticos, e eles podem

ser representados como *inteiros*, *decimais* e **razões**. Alguns exemplos de números em Lisp são 10, 543, 123.5, 0.35, 12/3, 15/7.

Símbolos: Símbolo é um fundamental tipo de dado no Lisp, pois representa praticamente qualquer caractere ou combinação de caracteres no código. Nomes de variáveis e de funções são símbolos e símbolos podem ser usados de forma literal, como se fossem Strings. Alguns exemplos de símbolos são BANANA, 123-456-789, EU-GOSTO-DE-LISP.

Em Lisp, existem dois símbolos especiais: T e NIL. Eles são símbolos que avaliam para si mesmos, e representam os valores lógicos TRUE e FALSE. T representa TRUE e NIL representa FALSE.

Vale notar também que todo valor que não seja NIL é considerada T pelo Lisp. Isso significa que 'banana seria considerada T em uma expressão booliana. Além disso, o símbolo NIL ao mesmo tempo que representa o valor falso, também representa uma lista vazia ().

5.0.2. Expressões em Lisp

Expressões em Lisp não são escritas na ordem usual. Ao invés de escrevermos (1 + 2), escrevemos (+ 1 2). Primeiro, deve-se inserir o nome da função que queremos chamar. Em seguida, escreve-se os argumentos se necessário.

(+ 1 1)	; Resulta em 2
(- 2 1 1)	; Resulta em 0
(* 2 2)	; Resulta em 4
(/ 2 2)	; Resulta em 1

Também 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.

5.0.3. Listas em Lisp

Listas são estruturas de dados que comportam vários elementos. Listas permitem que vários tipos de dados possam ser armazenados nela: Pode-se fazer listas misturando números, símbolos e outras listas aninhadas.

A sintaxe em código de listas é o uso de parênteses. Uma lista contendo os números 1, 2 e 3 seria, por exemplo: (1 2 3).

Na memória, entretanto, listas são armazenadas pelo uso de estruturas de dados chamadas de CONS CELLS.

5.0.4. Cons cells

a Cons cell é uma estrutura de dados formada por dois ponteiros. O primeiro ponteiro, chamado de CAR, aponta para o conteúdo da cons cell, enquanto o segundo ponteiro, chamado de CDR, aponta para o próximo elemento da lista.

Podemos encadear cons cells ao fazer com que o CDR de uma cons cell aponte para o CAR de outra. Assim são formadas as listas em Lisp. Quando o CDR da última cons cell da lista aponta para NIL, significa que a lista terminou.

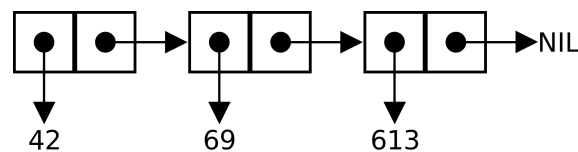


Figura 1 – Lista formada por 3 cons cells. Em código, seria a lista (42 69 613).

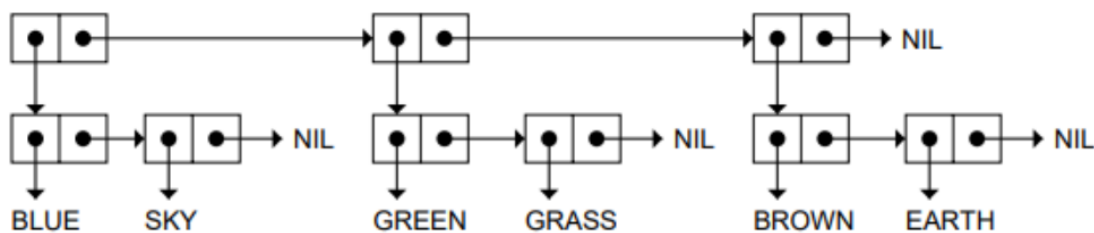


Figura 2 – Listas aninhadas. Em código, seria a lista ((BLUE SKY) (GREEN GRASS) (BROWN EARTH))

5.0.5. Funções de manipulação de listas

As funções CONS e LIST são usadas para criar listas em Lisp.

CONS recebe dois argumentos: Um elemento e uma lista. Ele então adiciona o elemento na frente da lista por meio da criação de uma nova cons cell que é adicionada no início da estrutura da lista.

LIST recebe um ou mais argumentos, e cria uma lista nova onde cada cons cell aponta para cada elemento passado nos argumentos.

Vejamos o exemplo:

```
(cons 'um '(dois tres))      ; Resulta em (um dois tres)
(list 'um 'dois 'tres)       ; Resulta em (um dois tres)
```

Outras funções importantes são as funções CAR e CDR, não a serem confundidas com o CAR e o CDR de uma cons cell. Essas funções são utilizadas para capturar elementos das listas.

A função CAR retorna o primeiro elemento da lista - ou seja - o CAR da primeira cons cell.

A função CDR retorna o resto da lista, excluindo o primeiro elemento - ou seja- o CDR da primeira cons cell. CDR sempre retorna uma lista, nem que seja uma lista vazia (NIL).

Exemplo em código das funções CAR e CDR:

```
(car '(abobora melancia banana manga)) ; Resulta em abobora  
(cdr '(abobora melancia banana manga)) ; Resulta em (melancia banana manga)
```

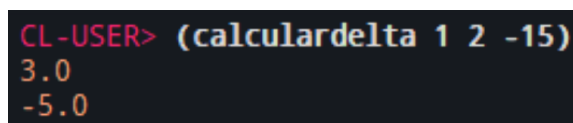
Também podemos concatenar CAR e CDR, sempre lendo da direita para esquerda, por exemplo, CADR é um CDR e depois um CAR, enquanto CDAR é um CAR e depois um CDR.

```
(cadr '((abobora melancia) (banana manga))) ; Resulta em (banana manga)  
(car (cdr '((abobora melancia) (banana manga)))) ; Resulta em (banana manga)  
  
(cdar '((abobora melancia) (banana manga))) ; Resulta em (melancia)  
(cdr (car '((abobora melancia) (banana manga)))) ; Resulta em (melancia)
```

5.0.6. Exemplo de código - Bhaskara

Abaixo faremos a definição de uma função pura de segundo grau simples, conforme a fórmula de bhaskara.

```
(defun calcular-bhaskara (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
```

Figura 3 – Demonstração da função funcionando.

5.0.7. Teorema de Pitagoras

Aqui vamos demonstrar um simples algoritmo que soluciona o teorema de pitágoras:

```
(defun pitagoras (a b)
  (sqrt (+ (* a a) (* b b))))
```

O algoritmo consegue retornar a hipotenusa de um triangulo retângulo, dado os catetos A e B.

5.0.8. Meta programação

Lisp suporta a *meta programação*, uma funcionalidade de altíssima abstração que permite o desenvolvimento de algoritmos muito complexos.

A meta programação consiste na capacidade do próprio código Lisp se alterar durante o tempo de execução. Isso significa que o código pode programar o próprio código.

Não vamos detalhar profundamente sobre as aplicações da meta programação pois isso não é cabível para uma apostila de nível introdutório como essa, mas vale notar que ela é eficiente para:

- Flexibilidade do programa.
- Adaptabilidade do programa.
- Gerenciamento de código.
- Reutilização de código.
- Inteligência artificial.

Aqui temos um exemplo de código que aproveita da meta programação:

```

(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))

```

Entenda, esse código *é complicado demais* para nível introdutório, então não se preocupe se não consegue ler ele.

O que o código faz é criar um novo tipo de dados, uma lista circular. Ele implementa duas funções para adicionar elementos à lista (`push`) e tirar elementos da lista (`pop`), e faz isso por meio da metaprogramação.

Aqui está a demonstração do código sendo utilizado:

```

(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)

```

Figura 4 – Utilizando o código

A meta programação é uma técnica complexa, mas de alto poder de abstração e de elegância na programação funcional.

5.0.9. Inteligência Artificial

Assim como no tópico de meta programação, não cabe a um material introdutório se interessar em entender o código apresentado, mas vale a citação e a demonstração das capacidades que o Lisp possui.

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 *retro-propagação*. 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))))
```

Figura 5 – Código complexo de uma IA.

Em meio a diversos termos técnicos e um código imenso, o que se deve entender deste tópico é que Lisp é uma linguagem poderosa para implementação de Inteligências Artificiais.

6. COMPARATIVO COM OUTROS PARADIGMAS

6.1. COMPARATIVO COM IMPERATIVO

6.1.1. Abordagem

Na programação funcional, o foco está na avaliação de expressões em um sistema imutável, em oposição ao foco na execução de uma sequência de instruções que modificam o estado do programa que o paradigma imperativo trás.

Enquanto o paradigma imperativo está interessado em elaborar instruções sequenciais para o que o computador deve fazer, o paradigma funcional se importa em apenas requisitar resultados do computador pela avaliação de expressões, independente de quais instruções ou fluxo o computador vai tomar para fazer isso.

6.1.2. Estados do sistema e efeitos colaterais

O gerenciamento dos estados se difere imensamente.

O paradigma imperativo não apenas permite, como tem como fundamental característica a capacidade de alterar o estado do sistema. É esperado que se altere o valor de variáveis constantemente, muitas vezes até por utilização de efeitos colaterais.

Já o funcional impede a alteração do valor de variáveis já criadas. Isso já entra em completa oposição às ideias mutáveis do paradigma imperativo. Além disso, efeitos colaterais são fortemente rejeitados ou até mesmo estritamente proibidos em linguagens de programação funcionais, o que também impede algumas práticas imperativas em programas funcionais.

6.1.3. Loops

Hoje em dia, estruturas de controle de fluxo como *While*, *for* e *ifs* são amplamente suportadas em linguagens imperativas. Essas estruturas são vitais para o funcionamento de programas imperativos, permitindo a solução de problemas complexos de forma muito mais simples.

No paradigma funcional, entretanto, essas estruturas também são evitadas. Neste paradigma, o uso de recursão é muito mais difundido e encorajado quando se deseja realizar repetições.

6.1.4. Paralelismo

Por ser imutável e não utilizar efeitos colaterais, o paradigma funcional possui funcionamento mais previsível, e por isso o paralelismo é mais simples de ser feito.

O paradigma imperativo, entretanto, com sua grande mutabilidade e utilização de efeitos colaterais, pode tornar programas menos previsíveis e mais complexos de serem utilizados em paralelismo sem causar erros.

6.2. COMPARATIVO COM ORIENTADA A OBJETO

6.2.1. Abordagem

Novamente, o foco em expressões do paradigma funcional se difere fortemente do foco na modelagem de objetos que encapsulam dados e métodos. No paradigma orientado a objeto, queremos gerenciar objetos por meio de mensagens e construtores.

Isso se distancia fortemente da abordagem focada em funções e avaliações que o paradigma funcional possui, onde o gerenciamento de objetos e encapsulamento de dados não ocorre em oposição ao gerenciamento de funções, avaliações e entradas/saídas.

6.2.2. Estados do sistema e efeitos colaterais

O paradigma orientado a objeto se assemelha ao imperativo no sentido que também está interessado no gerenciamento de estados mutáveis durante tempo de execução. A diferença é que esses estados estão encapsulados em objetos. Além disso, o uso de efeitos colaterais é quase que necessário para programar em orientação a objeto, como na aplicação de métodos *Setters* e afins.

Dessa forma, assim como o imperativo, o paradigma orientado a objetos se difere fortemente do paradigma funcional no quesito de mutabilidade e efeitos colaterais. Enquanto o funcional rejeita ambas ideias, elas são amplamente utilizadas na orientação a objetos.

6.2.3. 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.

6.2.4. 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.

6.3. COMPARATIVO COM LOGICA

6.3.1. Abordagem

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.

Isso já se distancia fortemente da abordagem do paradigma funcional, onde funções são avaliadas para solucionar problemas. Não estamos interessados em inferências lógicas. Estamos interessados em receber retornos de processamentos específicos.

De uma certa forma, ambos paradigmas se aproximam no sentido que estão focados em demonstrar respostas computáveis. A diferença existe justamente no tipo de respostas que cada paradigma gera.

6.3.2. 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.

6.3.3. 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.

6.3.4. 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.

7. QUESTÕES PRODUZIDAS

7.1. QUESTÕES TEÓRICAS

Questão 1

Qual é uma característica fundamental do paradigma funcional?

- A Uso extensivo de loops
- B Ênfase na mutabilidade dos dados
- C Recursividade
- D Programação orientada a objetos

Questão 2

No paradigma funcional, o que significa uma função ser "pura"?

- A Não possui argumentos
- B Sempre retorna o mesmo resultado para os mesmos argumentos
- C Pode modificar variáveis globais
- D É definida por funções que recebem e retornam funções

Questão 3

Qual dos seguintes é um princípio central do paradigma funcional?

- A Herança
- B Polimorfismo
- C Imutabilidade
- D Encapsulamento

Questão 4

Qual a vantagem da imutabilidade no paradigma funcional?

- A Facilita o uso de herança
- B Simplifica o gerenciamento de estados
- C Permite a criação de funções de alta ordem
- D Permite a modificação direta de variáveis

Questão 5

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 I, II e III

Questão 6

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 funções diretamente através de outras funções

Questão 7

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 8

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 9

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 10

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 11

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. Isso permite que funções sejam compostas e reutilizadas independentemente do estado global do programa.”

- A Funções puras no paradigma funcional não produzem efeitos colaterais e podem alterar o estado global do programa.

- B Funções puras no paradigma funcional não produzem efeitos colaterais e podem retornar resultados diferentes para os mesmos argumentos de entrada.
- C Funções puras no paradigma funcional não produzem efeitos colaterais e são dependentes do estado global do programa.
- D Funções puras no paradigma funcional não produzem efeitos colaterais e seu resultado é previsível.

Questão 12

Sobre a programação funcional e seus benefícios, assinale a alternativa incorreta.

- A A programação funcional facilita a criação de programas concorrentes e paralelos devido à ausência de estados mutáveis.
- B A programação funcional torna mais difícil o rastreamento de mudanças de estado, aumentando a complexidade do programa.
- C A programação funcional promove a reutilização de código através de funções puras e composição de funções.
- D A programação funcional ajuda a evitar efeitos colaterais.

Respostas

- 1. C
- 2. B
- 3. C
- 4. B
- 5. B
- 6. C
- 7. B
- 8. B
- 9. B
- 10. D
- 11. D
- 12. B

7.2. QUESTÕES DE CÓDIGO

Questão 1

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

- I. `(+ 3 5)`
- II. `(3 + 5)`
- III. `(+ 3 (5 6))`
- IV. `(+ 3 (* 5 6))`
- V. `'(manha tarde noite)`
- VI. `('manha 'tarde 'noite)`
- VII. `(list 'manha 'tarde 'noite)`
- VIII. `(car nil)`
- IX. `(+ 3 banana)`
- X. `(+ 3 'banana)`

Questão 2

O que essa função faz? Qual seria um nome adequado para ela?

```
(defun FUNCAO-SECRETA (n)
  (if (<= n 1)
      1
      (* n (FUNCAO-SECRETA (- n 1)))))
```

Questão 3

O que essa função faz? Qual será o comportamento dela quando a entrada n for 0?

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

Questão 4

Escreva uma função TORNAR-PAR que recebe um número n como argumento. Ela deve transformar números ímpares em números pares por meio da soma de 1. Se a entrada de TORNAR-PAR já for par, a entrada deve ser retornada sem alteração.

Questão 5

Qual é a utilidade desse predicado? Qual seria um nome apropriado para esse predicado?

```
(defun PREDICADO (x y)
  (or (and (zerop x) (zerop y))
      (and (< x 0) (< y 0))
      (and (> x 0) (> y 0)))))
```

Questão 6

Qual dos seguintes predicados representa a operação lógica xOR?

- a) `(defun PREDICADO (x)
 (not (equal x 1)))`
- b) `(defun PREDICADO (x y)
 (not (equal x y)))`
- c) `(defun PREDICADO (x y)
 (not (or x y)))`
- d) `(defun PREDICADO (x y)
 (or x y))`

Questão 7

Qual função a seguir mais se difere das outras?

- a) `(defun FUNCAO (x)
 (/ x 2))`
- b) `(defun FUNCAO (x)
 (* x 0.5))`
- c) `(defun FUNCAO (x)
 (* x 1/2))`
- d) `(defun FUNCAO (x)
 (/ x -2))`

Respostas

1.

- I. 8
- II. ERRO
- III. ERRO
- IV. 33
- V. (manha tarde noite)
- VI. ERRO
- VII. (manha tarde noite)
- VIII. NIL
- IX. ERRO
- X. ERRO

2. A função calcula o fatorial de um número por meio da recursão. Um nome apropriado seria "fatorial".
3. A função afasta a entrada de 0. Se a entrada for positiva, soma 1. Se a soma for negativa, subtrai 1. A função subtrai 1 ao receber 0 como entrada, considerando o afastamento para os negativos.
4.

```
(defun TORNAR-PAR (N)
  (if (oddp n) (+ n 1) n))
```
5. Esse predicado retorna T se x e y possuírem o mesmo sinal ou se ambos forem 0, e NIL caso contrário. Um nome apropriado para esse predicado seria MESMO-SINAL-P.
6. B
7. D