Universidade Federal de Alagoas, Campus Arapiraca, SEDE - 57309-005
Curso: Ciência da Computação
Disciplina: Paradigmas de Linguagens de Programação
Professor: Fernando Dantas
Autor: Jhony Wictor do Nascimento Santos

Atividade: Entendendo o Pensamento Funcional

Etapa 1 — O que é uma Função?

1. **Pensando em Funções do Mundo Real**
```
dobro(x) ⇒ 2×x
area_circulo(r) ⇒ π × r²
proximo(x) ⇒ x + 1
```

a. Se eu chamar a função dobro(3) várias vezes, o resultado muda?

Não, pois o *dobro(3)* sempre retornará a 6. Isso acontece porque a função sempre faz o mesmo cálculo (multiplica o número de entrada por 2) e não depende de nenhum fator externo.

b. Ela depende de alguma variável externa (fora da função)?

Não *(se a definição for dobro(x) = 2*x)*. Se houvesse dependência de variável externa, deixaria de ser pura. Desse modo, ela não utiliza nenhuma variável global, arquivo ou dado que esteja fora de seu próprio escopo. Portanto, ela é independente de fatores externos.

c. Ela altera algo fora dela, como uma variável global, arquivo ou interface?

Não, uma função pura não altera nada fora dela. Se o *dobro* escrevesse num arquivo ou modificasse uma variável global, deixaria de ser pura (efeito colateral). Logo, podemos concluir que uma função pura depende apenas de suas entradas e não provoca efeitos colaterais; isso facilita raciocínio (testes, paralelismo, memoização).

Etapa 2 — Exemplo: Função Pura vs Função ¬ Pura

`Observe os dois trechos de código abaixo e discuta as perguntas que seguem.`

`Listing 1: Comparando função pura e ¬ pura.`
```
Funcao pura: sempre o mesmo resultado para a mesma entrada
def dobro(x):
     return x * 2

# Funcao $\neg$ pura: depende de vari\'avel global e altera o
estado do programa
contador = 0
def proximo():
     global contador
     contador += 1
     return contador
```

1. **A função proximo() sempre devolve o mesmo resultado?**

   Não, pois na primeira vez retorna 1, depois 2, depois 3 e assim por diante. De modo que ela é dependente do estado anterior da variável contador.

2. **Ela depende de algo externo?**

   Sim, uma vez que, ela é dependente da variável global contador, que está fora da função e influencia diretamente seu resultado.

3. **Ela altera o valor de alguma variável externa?**

   Sim, pois ela modifica o valor de contador, aumentando-o em 1 a cada execução. Portanto, muda o estado do programa.

4. **Qual das duas se comporta como uma função pura? Por quê?**

   Pode-se afirmar que função dobro(x) é a função pura, visto que, ela sempre devolve o mesmo resultado para a mesma entrada; não é dependente de variáveis externas; assim como, não altera o estado do programa ou o ambiente externo.

Tipo	Características principais	Exemplo
Função Pura	Mesmo resultado para as mesmas entradas; não altera o estado do programa; previsível e fácil de testar.	dobro(x)
Função Não Pura (¬ pura)	Depende de variáveis externas; altera o estado do programa; pode gerar resultados diferentes a cada execução.	proximo()

Logo, pode-se concluir que, a pureza de uma função está relacionada à previsibilidade e independência. Enquanto que, o *dobro(x)* representa uma transformação pura de dados, *proximo()* envolve estado e efeitos colaterais, sendo, portanto, não pura.

Etapa 3 — Imutabilidade e Recursão

1. Pensando sem alterar dados

```
# Pensamento imperativo (modifica a lista original)
compras = ["leite", "pão"]
compras.append("café")
# A lista original foi alterada: ["leite", "pão", "café"]

# Pensamento funcional (imutável)
compras = ["leite", "pão"]
nova_lista = compras + ["café"]
# A lista original continua a mesma: ["leite", "pão"]
# A nova lista é: ["leite", "pão", "café"]
```

Por que pode ser vantajoso evitar alterar diretamente os dados originais?

Em programas grandes, diferentes partes do código podem acessar a mesma variável. Se uma parte altera esse dado sem querer, outras partes podem se comportar de forma incorreta. Mantendo os dados imutáveis, garantimos previsibilidade: o valor nunca muda, desse modo, evitando erros por modificações inesperadas.

Além disso, quando os dados não mudam, é mais fácil rastrear a origem de um erro. Onde cada transformação é clara e isolada, sem efeitos colaterais escondidos; o que facilita a depuração *(debug)*.

Por fim, ele garante um paralelismo seguro, visto que, seus programas executam várias tarefas ao mesmo tempo *(threads ou processos paralelos)*, duas funções poderiam tentar alterar o mesmo dado ao mesmo tempo. Portanto, com a imutabilidade, isso não ocorre, pois cada função trabalha com sua própria cópia dos dados.

2. Pensando sem laços — a recursão

`No paradigma funcional, não usamos laços (for, while) para repetir ações. Em vez disso, usamos recursão: a função chama a si mesma até atingir um caso-base.`

Listing 2: Somando os elementos de uma lista de forma recursiva.

```
def soma(lista):
    if not lista:
        return 0
    return lista[0] + soma(lista[1:])

Compare com a versão imperativa:
total = 0
for x in lista:
    total += x
```

a. Qual das versões altera uma variável a cada passo?

A versão imperativa altera uma variável *(total)* a cada repetição do *for.* Ela depende de um estado mutável, o que é típico do pensamento imperativo.

b. Qual delas apenas devolve um novo resultado, sem modificar nada?

A versão recursiva *(funcional)* não altera variáveis nem dados originais. Ela apenas retorna um novo valor a cada chamada, com base nas entradas recebidas.

c. Como a recursão se relaciona com o conceito de imutabilidade?

A recursão é uma forma natural de preservar a imutabilidade, pois, ao invés de atualizar as variáveis *(como total)*, a função cria novos valores a cada chamada, de modo que nenhum dado existente é modificado — cada passo trabalha com cópias ou versões derivadas dos dados. Desse modo, garantindo que o processo seja previsível, puro e sem efeitos colaterais.

Conceito	Versão Imperativa	Versão Funcional (Recursiva)

Uso de variáveis	Modifica total a cada passo	Não altera variáveis
Estado	Mutável	Imutável
Método de repetição	Laço for ou while	Recursão (função chama a si mesma)
Relação com imutabilidade	Quebra a imutabilidade	Mantém a imutabilidade

Logo, pode-se afirmar que a recursão substitui os laços tradicionais para repetir ações sem alterar o estado. Pois, o uso da recursividade expressa o princípio da imutabilidade, pois cada chamada gera novos resultados, sem modificar dados originais ou variáveis globais.

3. Desafio Final

Escreva uma função recursiva que:

a. Receba uma lista de números;
b. Retorne uma nova lista contendo apenas os números pares;
c. Sem usar laços ou variáveis mutáveis.

Dica: use o raciocínio: "Se o primeiro elemento for par, inclua-o no resultado; senão, apenas processe o restante da lista."

1. **Implementação do Código em Linguagem Python (.Py)**

```
def filtrar_pares(lista):
    # Caso base: lista vazia
    if not lista:
        return []

    # Passo recursivo:
    primeiro = lista[0]
    resto = lista[1:]

    if primeiro % 2 == 0:
        # Se for par, inclui no resultado
        return [primeiro] + filtrar_pares(resto)
    else:
        # Se for ímpar, ignora e continua com o resto
        return filtrar_pares(resto)
```

2. **Exemplo de Uso**

```
numeros = [1, 2, 3, 4, 5, 6]
resultado = filtrar_pares(numeros)
print(resultado)
```

3. **Saída**

```
[2, 4, 6]
```

Logo, pode-se afirmar que a função apresentada é considerada funcional, uma vez que, segue os princípios fundamentais da programação funcional, baseados na recursão e na imutabilidade. Em seu funcionamento, ela não utiliza variáveis mutáveis nem estruturas de repetição como for ou while; em vez disso, repete suas operações por meio da recursão, chamando a si mesma até que a lista esteja vazia.

A cada chamada, a função cria uma nova lista contendo apenas os números pares, sem modificar a lista original. Isso significa que, para a mesma entrada, o resultado será sempre o mesmo, tornando-a uma função pura, previsível e livre de efeitos colaterais.

Portanto, essa função exemplifica claramente o raciocínio funcional, no qual o foco não está em alterar o estado do programa, mas em transformar dados em novos valores. Cada passo é independente e gera um novo resultado, mantendo os dados originais intactos. Essa forma de pensar e programar torna o código mais seguro, legível e confiável, pois evita erros causados por modificações inesperadas e facilita o rastreamento da lógica.

Em síntese, a recursão e a imutabilidade, aplicadas nessa função, demonstram como a programação funcional busca previsibilidade, clareza e pureza nas transformações de dados, produzindo programas mais estáveis e fáceis de manter.

4. Pensando sem laços — a recursão (cont.)

Listing 3: Somando os elementos de uma lista de forma recursiva

```
def soma(lista):
    if not lista:
        return 0
    return lista[0] + soma(lista[1:])
```

Traço de execução da função recursiva
Para a chamada inicial soma([3, 1, 4]), temos:

Etapa	Chamada da função	Lista vazia?	Retorno parcial
1	soma([3, 1, 4])	Não	$3 +$ soma([1, 4])
2	soma([1, 4])	Não	$1 +$ soma([4])
3	soma([4])	Não	$4 +$ soma([])
4	soma([])	Sim	0

Ao retornar das chamadas, temos:

$$\begin{aligned} \texttt{soma([])} &= 0 \\ \texttt{soma([4])} &= 4 + 0 = 4 \\ \texttt{soma([1, 4])} &= 1 + 4 = 5 \\ \texttt{soma([3, 1, 4])} &= 3 + 5 = 8 \end{aligned}$$

Visualização em Forma de Pilha:

```
soma([3, 1, 4])
│
├── soma([1, 4])
│    ├── soma([4])
│    │    └── soma([]) → 0
│    └── retorna 4 + 0 = 4
└── retorna 3 + (1 + 4) = 8
```

Logo, podemos afirmar que, a função recursiva soma os elementos de uma lista chamando a si mesma com listas menores até atingir o caso-base, quando a lista está vazia. Cada chamada fica empilhada, aguardando o resultado da próxima, e os valores são somados na volta, do nível mais interno para o externo. Esse processo não provoca alterações na lista original e cada chamada cria uma nova instância da função, mantendo a imutabilidade. Assim, a recursão exemplifica o pensamento funcional, transformando dados em novos valores de forma previsível, limpa e sem efeitos colaterais.

5. Desafio Final

a. Escreva o traço de execução detalhado da chamada soma([2, 5, 7, 1]), seguindo o mesmo modelo da tabela acima.

Etapa	Chamada da função	Lista vazia?	Retorno parcial
1	soma([2, 5, 7, 1])	Não	2 + soma([5, 7, 1])
2	soma([5, 7, 1])	Não	5 + soma([7, 1])
3	soma([7, 1])	Não	7 + soma([1])
4	soma([1])	Não	1 + soma([])
5	soma([])	Sim	0

b. Indique o valor retornado em cada etapa.

soma([]) = 0
soma([1]) = 1 + 0 = 1
soma([7, 1]) = 7 + 1 = 8
soma([5, 7, 1]) = 5 + 8 = 13
soma([2, 5, 7, 1]) = 2 + 13 = 15

Visualização em Forma de Pilha

```
soma([2, 5, 7, 1])
│
├── soma([5, 7, 1])
│   │
│   ├── soma([1,2])
│   │   └── soma([] → 0
│   └── retorna 1 + 0 = 1
└── retorna 2 + (5 + 8) = 15
```

c. Explique por que a função não altera a lista original durante o processo.

A função não modifica a lista original porque cada chamada recursiva cria uma nova sublista *(lista[1:])* em vez de alterar os elementos da lista existente. O cálculo da soma é feito com novos valores criados a cada chamada, enquanto a lista original permanece intacta. O caso-base *(lista == [])* é responsável por garantir que a recursão termine, retornando valores que são somados de volta, sem jamais alterar os dados originais.