Capítulo 7

Avaliação preguiçosa

A forma como expressões são avaliadas em linguagens de programação pode ser classificada em dois tipos principais:

- **Avaliação ansiosa ou estrita (*eager* ou *strict*).** Nesse modelo, uma expressão é avaliada no momento exato em que é definida, ou seja, o valor é calculado imediatamente. Por exemplo, considere o código 7.1.

Código 7.1: Exemplo de avaliação ansiosa

```scala
def inc(x: Int): Int = {
    println("Applying inc to " + x)
    x + 1
}
val eagerVal = inc(10)
println("Binding eagerVal finished")
println("Eager value: " + eagerVal)
```

```
Applying inc to 10
Binding eagerVal finished
Eager value: 11
```

Perceba que, ao definir `eagerVal`, a função `inc` é imediatamente avaliada, resultando na impressão de `"Applying inc to 10"` antes mesmo de qualquer outra operação.

- **Avaliação preguiçosa (*lazy* ou *non-strict*).** Nesse modelo, uma expressão não é avaliada até que seu valor seja realmente necessário. Isso pode ser útil para evitar cálculos desnecessários. No exemplo a seguir, a função `f` não é avaliada até que o valor de a seja realmente necessário. Considere o código 7.2.

Código 7.2: Exemplo de avaliação preguiçosa

```scala
def inc(x: Int): Int = {
    println("Applying inc to " + x)
    x + 1
}
lazy val lazyVal = inc(20)
```

```
println("Binding lazyVal finished")
println("Accessing lazy value: " + lazyVal)
```

```
Binding lazyVal finished
Applying inc to 20
Accessing lazy value: 21
```

Em Scala, quando usamos o modificador `lazy` ao definir uma variável, a avaliação do valor dessa variável (lado direito da amarração) é adiada até que ela seja realmente necessária. No exemplo acima, a função `inc` só é chamada quando tentamos acessar o valor de `lazyVal`, resultando na impressão de `"Applying inc to 20"` apenas nesse momento.

Terminologia A avaliação ansiosa também é conhecida como ordem aplicativa (applicative order). A avaliação preguiçosa é conhecida como ordem normal (normal order). A terminologia *preguiçosa* e *ansiosa* é mais comum em linguagens de programação funcionais, enquanto *normal order* e *applicative order* são mais comuns em teorias de cálculo.

Linguagens estritas e não-estritas Linguagens de programação podem ser classificadas como *estritas* ou *não-estritas* (ou *preguiçosas*). Uma linguagem é estrita quando suas expressões, por padrão, são avaliadas de forma ansiosa. Por outro lado, uma linguagem é não-estrita quando suas todas suas expressões são avaliadas, por padrão, de forma preguiçosa.

A grande maioria das linguagens de programação são estritas, como Java, C, Python e JavaScript e Scala. Por outro lado, linguagens como Haskell e Miranda são exemplos de linguagens não-estritas.

Há uma terceiro grupo, relativamente numeroso, composto por linguagens estritas que possuem algum mecanismo de avaliação preguiçosa, como é o caso de Scala, Python, JavaScript, Clojure, Kotlin, Lisp, etc.

Aplicações A avaliação preguiçosa possui diversas aplicações importantes dentro da programação funcional, como:

- **Estruturas de dados infinitas**. A avaliação preguiçosa permite a criação de listas infinitas, como a sequência de números naturais. Por exemplo, podemos definir uma lista infinita de números naturais e acessar apenas os primeiros elementos dela.

- **Evitar cálculos desnecessários**. Em muitos casos, uma expressão pode ser definida, mas seu valor não será necessário. Por exemplo, em algoritmos que usam memoização, a avaliação preguiçosa pode computar o valor de uma função apenas se ela não tiver sido calculada anteriormente. Isso economiza tempo e recursos computacionais.

- **Criação de fluxo de avaliação customizado**. Algumas expressões podem ser definidas de forma que sua avaliação dependa de condições específicas. Por exemplo, é possível definir a expressão condicional (if-else) e os operadores

lógicos (and, or) de forma que a avaliação de uma expressão dependa do resultado de outra. Isso é útil para evitar erros de execução, como divisão por zero.

Em Scala, os principais recursos para trabalhar com avaliação preguiçosa são:

- **Amarrações preguiçosas (`lazy val`)**. Como vimos no exemplo acima, podemos definir variáveis que só serão avaliadas quando forem realmente necessárias.
- **Chamadas por nome**. Em Scala, podemos definir parâmetros de função como "chamadas por nome" usando a sintaxe `=>`. Isso significa que o parâmetro não é avaliado até que seja realmente necessário.
- **Estruturas de dados preguiçosas**. Podemos usar coleções como `LazyList` para criar listas que são avaliadas de forma preguiçosa, permitindo a manipulação de listas infinitas.
- **Controle de fluxo preguiçoso**. Podemos usar expressões condicionais e operadores lógicos de forma que a avaliação de uma expressão dependa do resultado de outra, evitando erros de execução.
- **Casamento de padrões preguiçoso**. Em Scala, podemos usar o casamento de padrões para definir expressões que só serão avaliadas quando forem realmente necessárias, permitindo a criação de estruturas de dados complexas e eficientes.

Nas próximas seções deste capítulo, exploraremos mais detalhadamente esses recursos e como aplicá-los na prática.

7.1 Amarrações preguiçosas

Amarração Em programação, uma amarração é o processo de associar um nome a um valor ou expressão. Em Scala, usamos a palavra-chave `val` para definir uma amarração imutável. Por exemplo, a linha `val x = 10` associa o nome `x` ao valor `10`.

Amarração ansiosa. Em Scala, a amarração de uma variável, por padrão, é ansiosa, isto é, o lado direito da amarração é avaliado durante a definição da variável. Por exemplo, considere o código 7.3.

Código 7.3: Exemplo de amarração ansiosa

```
val x = {
    println("Evaluating x")
    10
}
println("Binding of x finished")
println(x)
```

```
Evaluating x
Binding of x finished
10
```

AMARRAÇÃO PREGUIÇOSA. Podemos subverter o modelo padrão de amarração por meio da palavra-chave `lazy`. Quando usamos `lazy`, o lado direito da amarração não é avaliado até que o valor seja realmente necessário. Em linhas gerais, a expressão do lado direito fica "pendente" até que a variável seja acessada pela primeira vez. Por exemplo, considere o código 7.4.

Código 7.4: Exemplo de amarração preguiçosa

```
1 lazy val y = {
2     println("Evaluating y")
3     20
4 }
5 println("Binding of y finished")
6 println(y)
```

```
Binding of y finished
Evaluating y
20
```

COMPUTAÇÃO ADIADA O conceito de amarração preguiçosa está relacionado ao conceito de computação adiada (*deferred computation*). Isso significa que a avaliação de uma expressão é adiada até que seja realmente necessária. O conceito de computação adiada é aplicável a todos os paradigmas de programação, inclusive o imperativo. Por exemplo, quando passamos um bloco de código para ser executado por uma thread, estamos adiando a execução desse bloco até que a thread seja iniciada.

AVALIAÇÃO TARDIA Nas linguagens declarativas, o principal mecanismo de computação adiada é a avaliação tardia (*deferred evaluation*). A avaliação tardia é uma técnica de avaliação preguiçosa que adia a avaliação de expressões até seu primeiro acesso. Nesse primeiro acesso, dizemos que ocorreu a operação de forçar a avaliação (*force evaluation*). No exemplo acima, a mensagem `"Evaluating y"` só é impressa quando tentamos forçar a avaliação de `y`.

MEMOIZAÇÃO A técnica de memoização pode ser descrita como um processo envolvendo:

1. **Computação adiada**: a expressão é definida, mas não avaliada imediatamente.
2. **Primeiro acesso (forçamento)**: a expressão é avaliada pela primeira vez quando o valor é necessário.
3. **Avaliação da expressão**: o valor é calculado.

4. **Armazenamento do resultado**: o valor calculado é armazenado (cacheado).

5. **Retorno do valor armazenado**: em acessos subsequentes, o valor já calculado é retornado sem recalcular a expressão.

Perceba que o processo envolvido numa amarração preguiçosa é análogo ao processo de memoização. Quando declaramos uma variável como `lazy`, estamos essencialmente criando uma forma de memoização.

Imutabilidade Vale ressaltar que as amarrações preguiçosas são imutáveis, ou melhor não é possível definir amarrações preguiçosas como variáveis mutáveis. A razão para isso é simples: se uma variável preguiçosa fosse mutável, poderíamos alterar o valor dela após a primeira avaliação, o que quebraria a transparência referencial da expressão, que é uma pré-condição para a memoização.

Adiamento de avaliações custosas. A avaliação preguiçosa é especialmente útil quando o valor de uma variável pode ser computacionalmente caro ou quando a variável pode nunca ser acessada. Isso permite que o programa evite cálculos desnecessários, economizando tempo e recursos. Por exemplo, poderíamos definir uma função custosa, conforme o código 7.5.

Código 7.5: Exemplo de função custosa

```scala
def heavyComputation(): Int = {
    println("Performing heavy computation...")
    Thread.sleep(5000) // Simulate a long computation
    42
}

@main def run = {
    lazy val cachedData = heavyComputation()
    val isDataNeeded = scala.util.Random.nextBoolean()

    if (isDataNeeded) {
        println("Data is needed, accessing cachedData...")
        println(s"Cached data: $cachedData")
        println(s"New access to cachedData: ${cachedData + 1}")
    } else {
        println("Data is not needed, skipping access to cachedData.")
    }
}
```

```
Data is not needed, skipping access to cachedData.
```

```
Data is needed, accessing cachedData...
Performing heavy computation...
Cached data: 42
```

No exemplo do código 7.5, a função `heavyComputation` simula uma computação pesada. Ao definir `cachedData` como `lazy`, garantimos que a função só será executada se realmente precisarmos do valor. Introduzimos aleatoriedade para simu-

lar uma situação em que o valor pode ou não ser necessário. Se `isDataNeeded` for `false`, a função não é chamada, economizando tempo e recursos.

MEMOIZAÇÃO DE PROPRIEDADES DE OBJETOS Uma aplicação interessante das amarrações preguiçosas é o adiamento da inicialização e posterior memoização das propriedades de objetos. Por exemplo, podemos definir uma classe imutável que efetua operações custosas sobre um conjunto de dados, mas adiar a execução dessas operações até que os dados sejam realmente necessários. Veja o código 7.6.

Código 7.6: Exemplo de adiamento de propriedades

```
case class SalesReport(sales: List[Double]) {
    lazy val totalSales: Double = {
        println("Calculating total sales...")
        sales.sum
    }
    lazy val averageSales: Double = {
        println("Calculating average sales...")
        if (sales.isEmpty) 0.0 else totalSales / sales.size
    }
    lazy val biggestSale: Option[Double] = {
        println("Finding biggest sale...")
        if (sales.isEmpty) None else Some(sales.max)
    }
}
```

Perceba que definimos três propriedades preguiçosas: `totalSales`, `averageSales` e `biggestSale`. Essas propriedades só serão calculadas quando forem acessadas pela primeira vez, permitindo que o objeto `SalesReport` seja inicializado rapidamente, mesmo que as operações de cálculo sejam custosas. Incluímos mensagens de log na definição das propriedades para ilustrar quando cada cálculo é realizado. Numa situação real, essas mensagens precisam ser retiradas pois prejudicariam a transparência referencial das propriedades.

Considere uma função utilitária que imprima o relatório de vendas, conforme o código 7.7.

Código 7.7: Exemplo de uso de adiamento de propriedades

```
def printReport(report: SalesReport): Unit = {
    println(s"Total Sales: ${report.totalSales}")
    println(s"Average Sales: ${report.averageSales}")
    report.biggestSale match {
        case Some(sale) => println(s"Biggest Sale: $sale")
        case None => println("No sales data available.")
    }
}
```

Podemos usar essa função para simular a geração repetida desse relatório. Por exemplo, o código 7.8 cria o relatório de vendas duas vezes.

Código 7.8: Exemplo de chamada de adiamento de propriedades

```
@main def run = {
    val report = new SalesReport(List(100.0, 200.0, 300.0))

    println("Sales report created.")
    println("First report...")
    printReport(report)
    println("Second report...")
    printReport(report)
}
```

```
Sales report created.
First report...
Calculating total sales...
Total Sales: 600.0
Calculating average sales...
Average Sales: 200.0
Finding biggest sale...
Biggest Sale: 300.0
Second report...
Total Sales: 600.0
Average Sales: 200.0
Biggest Sale: 300.0
```

Perceba que, na primeira chamada de `printReport`, as propriedades `totalSales`, `averageSales` e `biggestSale` são calculadas. Na segunda chamada, os valores já estão cacheados, evitando cálculos desnecessários.

PROPRIEDADES PREGUIÇOSAS VS. MÉTODOS Uma alternativa para implementar a classe `SalesReport` seria usar métodos em vez de propriedades preguiçosas. No entanto, ao gerarmos o relatório de vendas duas vezes, os corpos dos métodos seriam reavaliados a cada chamada, resultando em cálculos repetidos. Considerando que a fonte de dados é uma lista imutável, não faz sentido recalcular os valores múltiplas vezes pois o resultado é invariante.

PROPRIEDADES PREGUIÇOSAS E CLASSES IMUTÁVEIS As propriedades preguiçosas são especialmente úteis em classes imutáveis, pois podem representar avaliações custosas sobre dados imutáveis. Uma vez que não existe a possibilidade de alterar o estado dos dados, faz todo sentido cachear o resultado dessas avaliações. Em objetos mutáveis, essa otimização não seria relevante, pois não temos garantias de imutabilidade, logo os cálculos precisam ser feitos sempre que acessarmos a propriedade.

7.2 Argumentos preguiçosos

Os argumentos de uma função podem ser avaliados de acordo com três técnicas principais:

- **Chamada por valor (*call-by-value*)**: o argumento é avaliado antes da chamada da função.

- **Chamada por nome (*call-by-name*)**: o argumento é avaliado sempre que é referenciado dentro da função, mas não é avaliado antes da chamada.
- **Chamada por necessidade (*call-by-need*)**: semelhante à chamada por nome, mas o valor é calculado apenas uma vez e armazenado para acessos futuros, evitando reavaliações desnecessárias

A chamada por valor é uma técnica de avaliação ansiosa, enquanto a chamada por nome e a chamada por necessidade são técnicas de avaliação preguiçosa. Nas próximas seções, exploraremos cada uma dessas técnicas em detalhes.

7.2.1 Chamada por valor

Na chamada por valor, os argumentos são avaliados antes da avaliação do corpo da função. Isso implica que, quando o corpo da função é executado, os valores dos argumentos já estão disponíveis. Para ilustração, considere que temos uma função custosa, conforme o código 7.9.

Código 7.9: Exemplo de função custosa

```
def expensiveComputation(n: Int): Int = {
    println(s"Expensive computation for $n")
    Thread.sleep(1000) // Simula processamento custoso
    n * n
}
```

Agora suponha que tenhamos uma função simples que adiciona dois inteiros, que é chamada tendo como argumentos duas chamadas da função `expensiveComputation`, conforme o código 7.10.

Código 7.10: Exemplo de chamada por valor

```
def add(x: Int, y: Int): Int = {
    println("add body started")
    println(s"Computing: $x + $y")
    x + y
}
@main def testCallByValue = {
    println("Before calling add...")

    val result = add(expensiveComputation(3), expensiveComputation(4))

    println(s"Result: $result")
}
```

```
Before calling add...
Expensive computation for 3
Expensive computation for 4
add body started
Computing: 9 + 16
Result: 25
```

Como podemos ver no exemplo do código 7.10, as chamadas para `expensiveComputation(3)` e `expensiveComputation(4)` são avaliadas antes da chamada da função `add`. Esse resultado evidencia a principal característica da chamada por valor: os argumentos são avaliados antes da execução do corpo da função.

Embora essa abordagem seja padrão em muitas linguagens de programação, ela pode levar a ineficiências, especialmente quando os argumentos são computações custosas que podem não ser necessárias. Em tais casos, a avaliação preguiçosa pode oferecer vantagens significativas.

7.2.2 Chamada por nome

Na chamada por nome, os argumentos não são avaliados antes da chamada da função. Em vez disso, eles são avaliados sempre que são referenciados dentro do corpo da função.

THUNKS Um modo simples de implementar a chamada por nome é usando *thunks*, que são funções anônimas (expressões lambda) sem parâmetros que encapsulam uma expressão. Por exemplo, podemos remodelar o exemplo anterior para usar thunks, conforme o código 7.11.

Código 7.11: Exemplo de chamada por nome com thunks

```
def add(x: () => Int, y: () => Int): Int = {
    println("add body started")
    println(s"Computing: ${x()} + ${y()}")
    x() + y()
}

def expensiveComputation(n: Int): Int = {
    println(s"Expensive computation for $n")
    Thread.sleep(1000) // Simula processamento custoso
    n * n
}

@main def testCallByName = {
    println("Before calling add...")

    val result = add(() => expensiveComputation(3), () =>
    ↪  expensiveComputation(4))

    println(s"Result: $result")
}
```

```
add body started
Expensive computation for 3
Expensive computation for 4
Computing: 9 + 16
Expensive computation for 3
Expensive computation for 4
```

```
Result: 25
```

Perceba que fizemos alterações significativas no código. A função `add` agora recebe dois thunks como argumentos, que são avaliados dentro do corpo da função. Isso significa que as expressões `x()` e `y()` são avaliadas apenas quando são referenciadas, permitindo que a função `add` seja chamada sem avaliar os argumentos antecipadamente. Além disso, naturalmente, ao chamar a função `add`, os thunks são passados como argumentos, o que permite que a avaliação seja adiada até que os valores sejam realmente necessários.

Quando observamos a saída do programa, notamos que a operações custosa é chamada quatro vezes: duas considerando a variável `x` e duas considerando a variável `y`. Isso ocorre porque, ao chamar `x()` e `y()` dentro da função `add` — dentro do print e na baliação do resultado —, estamos forçando a avaliação dos thunks, resultando na execução da função `expensiveComputation`. De início, isso pode parecer ineficiente, mas veremos que esse comportamento é útil em certas situações.

NOTAÇÃO CONCISA (SEM PARÊNTESES) Por definição, uma thunk não possui parâmetros. Mesmo assim, pelo modo como definimos cada cada thunk (e.g. `() => ...`), somos obrigados a usar parênteses para aplicar a thunk. No entanto, precisamos lembrar que Scala possui uma sintaxe especial para definir funções sem parâmetros: basta omitir os parênteses tanto na definição quanto na chamada da nossa thunk. O código 7.12 ilustra essa sintaxe simplificada.

Código 7.12: Exemplo de chamada por nome com thunks sem parâmetros

```scala
def add(x: => Int, y: => Int): Int = {
    println("add body started")
    println(s"Computing: $x + $y")
    x + y
}

def expensiveComputation(n: Int): Int = {
    println(s"Expensive computation for $n")
    Thread.sleep(1000) // Simula processamento custoso
    n * n
}

@main def testCallByName = {
    println("Before calling add...")

    val result = add(expensiveComputation(3), expensiveComputation(4))

    println(s"Result: $result")
}
```

```
add body started
Expensive computation for 3
Expensive computation for 4
Computing: 9 + 16
Expensive computation for 3
Expensive computation for 4
```

```
Result: 25
```

Perceba que a principal diferença é que agora não precisamos definir os thunks como funções anônimas. Em vez disso, usamos a sintaxe `x: => Int` para indicar que `x` é um parâmetro de chamada por nome. Isso simplifica a definição e o uso dos thunks, tornando o código mais legível. Por exemplo, no código principal, que aplica a função `add`, não precisamos mais envolver as chamadas de `expensiveComputation` em uma função anônima. Além disso, no corpo da função `add`, podemos referenciar os parâmetros diretamente, sem a necessidade de usar parênteses para aplicar a thunk do argumento.

Adiamento de argumentos custosos

A aplicação mais direta da chamada por nome é o adiamento de argumentos custosos. Por exemplo, vamos relembrar o exemplo do fibonacci memoizado, que utiliza um mapa mutável como cache, conforme o código 7.13.

Código 7.13: Exemplo de fibonacci memoizado

```scala
import scala.collection.mutable

object Fibonacci {
    private val memo: mutable.Map[Int, Int] = mutable.Map()

    def apply(n: Int): Int = {
        if (n <= 1) return n
        memo.getOrElseUpdate(n, {
            println(s"Computing F($n) (thunk evaluation)...")
            apply(n - 1) + apply(n - 2)
        })
    }
}

@main def testMemoizedFibonacci = {
    println(Fibonacci(4))
    println(Fibonacci(5))
}
```

```
Computing F(4) (thunk evaluation)...
Computing F(3) (thunk evaluation)...
Computing F(2) (thunk evaluation)...
3
Computing F(5) (thunk evaluation)...
5
```

Perceba que, ao chamar `Fibonacci(4)`, a função `apply` é chamada recursivamente para calcular os valores de `F(3)` e `F(2)`. Já a chamada `Fibonacci(5)` reutiliza os valores já calculados, evitando reavaliações desnecessárias.

Uma pergunta importante nesse código é: por que o segundo argumento do método `getOrElseUpdate`, que é uma mera expressão de bloco, não é avaliada sempre que o método é chamado? Esse comportamento ocorre pois o método

`getOrElseUpdate` define o segundo argumento via chamada por nome, conforme reproduzido na implementação do método extraída do código fonte do Scala, vide 7.14.

Código 7.14: Exemplo de getOrElseUpdate

```
def getOrElseUpdate(key: K, defaultValue: => V): V =
    get(key) match {
      case Some(v) => v
      case None => val d = defaultValue; this(key) = d; d
    }
```

Como podemos observar, o segundo argumento do método `getOrElseUpdate` é definido como `defaultValue: => V`, o que significa que ele é avaliado apenas quando acessado. Nesse caso, somente quando a expressão `match` corresponde com `None`, ou seja, temos um *cache miss*. Esse exemplo, embora simples, ilustra como a chamada por nome pode ser usada para adiar a avaliação de argumentos custosos, economizando recursos computacionais.

Curto circuito lógico

A chamada por nome é frequentemente usada para implementar o curto circuito lógico, que é um comportamento comum em expressões condicionais. O curto-circuito lógico evita a avaliação completa de uma expressão lógica quando o resultado já é conhecido. Por exemplo, em uma expressão `A && B`, se `A` for `false`, não precisamos avaliar `B`, pois o resultado da expressão inteira já será `false`. Isso é especialmente útil para evitar erros de execução, como divisão por zero ou acesso a elementos fora dos limites de uma lista.

Por exemplo, considere o seguinte código que determina se um número é positivo e par simultaneamente , conforme o código 7.15.

Código 7.15: Exemplo de curto circuito lógico

```
def positive(n: Int): Boolean = {
    println(s"Checking if $n is positive")
    n > 0
}

def even(n: Int): Boolean = {
    println(s"Checking if $n is even")
    n % 2 == 0
}

@main def testShortCircuit = {
    println(positive(4) && even(4))
    println(positive(-4) && even(-4))
}
```

```
Checking if 4 is positive
Checking if 4 is even
```

```
true
Checking if -4 is positive
false
```

Perceba que, no primeiro caso, ambas as funções `positive` e `even` são avaliadas, pois o número 4 é positivo e par. No entanto, no segundo caso, a função `even` não é avaliada, pois a função `positive` retorna `false`, o que torna a expressão inteira falsa. Esse comportamento é intrínseco a esse operador e muitas vezes contamos com ele para evitar erros de execução.

CURTO-CIRCUITO COM CHAMADA POR NOME Podemos reimplementar o operador && usando a chamada por nome, conforme o código 7.16.

Código 7.16: Exemplo de implementação do curto circuito lógico

```
1 def customAnd(left: Boolean, right: => Boolean): Boolean = {
2     if (!left) false else right
3 }
4
5 @main def testCustomAnd = {
6     println(customAnd(positive(4), even(4)))
7     println(customAnd(positive(-4), even(-4)))
8 }
```

```
Checking if 4 is positive
Checking if 4 is even
true
Checking if -4 is positive
false
```

Perceba que, ao usar a chamada por nome, a função `customAnd` avalia o primeiro argumento `left` e, se ele for `false`, não avalia o segundo argumento `right`. Isso demonstra como a chamada por nome pode ser usada para implementar o curto circuito lógico de forma eficiente.

Exercício. Implemente uma função `customOr` que simule o operador `||` usando a chamada por nome, com curto circuito lógico.

Outras aplicações

Abordamos nessa seção algumas aplicações comuns da chamada por nome, como o adiamento de argumentos custosos e o curto circuito lógico. No entanto, a chamada por nome pode ser usada em outras situações interessantes como a definição de estruturas de controle customizadas, condicionais, loops, entre outros.

Esses recursos mostram-se valiosos para extensão da linguagem e para a definição de DSLs (Domain-Specific Languages). Por exemplo, podemos definir uma estrutura de controle que execute um bloco de código apenas se uma condição for verdadeira, ou que itere sobre uma coleção de forma preguiçosa, numa sintaxe que não necessariamente se assemelha à sintaxe padrão da linguagem Scala.

7.2.3 Chamada por necessidade

A chamada por necessidade é uma técnica de avaliação preguiçosa que combina as vantagens da chamada por nome com a eficiência da chamada por valor. De modo resumido, podemos dizer que corresponde à avaliação por nome combinada com memoização. Nessa abordagem, o argumento é avaliado apenas uma vez e o resultado é armazenado para acessos futuros. Isso evita reavaliações desnecessárias, economizando tempo e recursos computacionais.

A linguagem Scala não possui uma sintaxe específica para chamada por necessidade, mas podemos implementá-la usando uma combinação de thunks, amarrações preguiçosas e closures. O exemplo do código 7.17 ilustra uma possível implementação dessa técnica.

Código 7.17: Exemplo de chamada por necessidade

```scala
def divideByNeed(x: => Int, y: => Int): Option[Int] = {
    println("Started division...")
    lazy val cachedX = x // call-by-need
    lazy val cachedY = y // call-by-need
    if (cachedY == 0) None else Some(cachedX / cachedY)
}

def longProcess(x: Int): Int = {
    println("Long process running...")
    Thread.sleep(1000) // Simulate a long computation
    x
}

@main def testDivideByNeed = {
    println(divideByNeed(longProcess(10), longProcess(2)))
    println(divideByNeed(longProcess(10), longProcess(0)))
}
```

```
Started division...
Long process running...
Long process running...
Some(5)
Started division...
Long process running...
None
```

Perceba que, na função `divideByNeed`, os argumentos `x` e `y` são definidos como chamadas por nome. Adicionalmente, usamos amarrações preguiçosas para adiar e memoizar os valores de `x` e `y` em `cachedX` e `cachedY`. Como podemos notar pela saída do programa, o resultado disso é que, quando chamamos `divideByNeed(longProcess(10), longProcess(2))`, a função `longProcess` é avaliada apenas uma vez para cada argumento, mesmo que o resultado seja acessado múltiplas vezes dentro da função.

Isso fica bastante claro, principalmente, quando chamamos `divideByNeed(longProcess(10), longProcess(0))`, onde a função `longProcess` é avaliada apenas uma vez para o argumento `x`, enquanto o argumento `y` não é

avaliado, pois a divisão por zero não é permitida. Podemos dizer que, nesse caso, tivemos uma situação análoga ao curto-circuito lógico, onde o segundo argumento não é avaliado porque não é necessário para o resultado final.

COMPARAÇÃO COM OUTROS MODELOS A eficiência do modelo de chamada por necessidade é evidente quando comparado com a chamada por nome e a chamada por valor. Por exemplo, considere as soluções alternativas para o mesmo problema, conforme o código 7.18.

Código 7.18: Comparação entre chamada por valor e chamada por nome

```
def divideByValue(x: Int, y: Int): Option[Int] = {
    println("Started division...")
    if (y == 0) None else Some(x / y)
}

def divideByName(x: => Int, y: => Int): Option[Int] = {
    println("Started division...")
    if (y == 0) None else Some(x / y)
}

@main def testDivide = {
    println("Testing divideByName:")
    println(divideByValue(longProcess(10), longProcess(2)))
    println(divideByValue(longProcess(10), longProcess(0)))

    println()
    println("Testing divideByName:")
    println(divideByName(longProcess(10), longProcess(2)))
    println(divideByName(longProcess(10), longProcess(0)))
}
```

```
Testing divideByValue:
Long process running...
Long process running...
Started division...
Some(5)
Long process running...
Long process running...
Started division...
None

Testing divideByName:
Started division...
Long process running...
Long process running...
Long process running...
Some(5)
Started division...
Long process running...
None
```

Perceba que a versão com chamada por valor sempre avalia cada argumento

pelo menos uma vez. No caso da divisão por zero, mesmo que o segundo argumento não seja necessário, a função `longProcess` é avaliada duas vezes, resultando em uma ineficiência significativa. Por outro lado, a versão com chamada por nome avalia os argumentos sempre que necessário, o que, para esse problema, gera um resultado controverso. Para o caso de sucesso (`divideByName(longProcess(10), longProcess(2))`), a função `longProcess` é avaliada três vezes, uma vez para cada referência ao argumento `x` e duas vezes para o argumento `y` (uma para a condição e outra para o resultado). Já no caso de falha (`divideByName(longProcess(10), longProcess(0))`), a função `longProcess` é avaliada apenas uma vez, pois o segundo argumento não é necessário.

Em resumo, tanto para a chamada por nome quanto para a chamada por valor, tivemos um total de quatro avaliações para os dois casos, enquanto para chamada por necessidade tivemos três avaliações no total. Esse exemplo nos mostra que a avaliação preguiçosa de argumentos pode ter um impacto positivo (e em algumas situações, negativo) na eficiência do nosso programa.

7.3 Listas preguiçosas

Definição Uma lista preguiçosa é uma lista encadeada cujos nós sofrem avaliação preguiçosa, ou seja, tanto a cabeça quanto a cauda da lista sofrem avaliação preguiçosa. A biblioteca padrão de Scala possui uma implementação de listas preguiçosas chamada `LazyList`. A definição simplificada do tipo `LazyList` está listada no código 7.19.

Código 7.19: Definição simplificada de LazyList

```scala
sealed trait LazyList[+A] {
    def isEmpty: Boolean
    def head: A
    def tail: LazyList[A]
}

case object Empty extends LazyList[Nothing] {
    (...)
}

case class #::[+A](h: => A, t: => LazyList[A]) extends LazyList[A] {
    lazy val head: A = h        // Memoização da cabeça
    lazy val tail: LazyList[A] = t    // Memoização da cauda
    def isEmpty: Boolean = false
}
```

Podemos compreender um `LazyList` como um tipo soma, de modo análogo a uma `List` tradicional. No entanto, a principal diferença é que tanto os componentes `head` quanto `tail` são avaliados de forma preguiçosa. Como podemos observar no construtor primário do método `#::` (cons preguiçoso), tanto `h` quanto `t` são definidos como parâmetros de chamada por nome, mas são armazenados como amarrações preguiçosas. Isso implica uma variação de chamada por necessidade, onde o valor

é calculado apenas uma vez e armazenado para acessos futuros.

CONSTRUÇÃO DE `LazyList` A construção de uma `LazyList` é feita de forma semelhante à construção de uma lista tradicional, porém para uma lista preguiçosa, usamos um célula cons preguiçosa — `#::`. Por exemplo, podemos criar uma lista preguiçosa de números inteiros, conforme o código 7.20.

Código 7.20: Exemplo de LazyList

```scala
val lazyList: LazyList[Int] = 1 #:: 2 #:: 3 #:: Empty
```

O que diferencia uma `LazyList` de uma lista tradicional é que, ao acessar os elementos da lista, a avaliação é feita de forma preguiçosa. Por exemplo, considere o código 7.22, onde criamos uma lista preguiçosa com o quadrado dos números de 1 a 5.

Código 7.21: Demonstração de funcionamento da LazyList

```scala
def expensiveElement(n: Int): Int = {
    println(s" Computing element $n...")
    Thread.sleep(500) // Simula processamento custoso
    n * n
}

val lazyNumbers = expensiveElement(1)
    #:: expensiveElement(2)
    #:: expensiveElement(3)
    #:: expensiveElement(4)
    #:: expensiveElement(5)
    #:: LazyList.empty[Int]
```

Devido à avaliação preguiçosa, os elementos da lista só são calculados quando realmente acessados. Para visualizar melhor esse efeito, podemos emular o acesso incremental aos elementos da lista, conforme o código 7.22.

Código 7.22: Acesso incremental aos elementos da LazyList

```scala
@main def run = {
    println("First access to lazyNumbers:")
    lazyNumbers.foreach{ n =>
        println(s"Element: $n")
    }
    println("Second access to lazyNumbers:")
    lazyNumbers.foreach{ n =>
        println(s"Element: $n")
    }
}
```

```
First access to lazyNumbers:
 Computing element 1...
Element: 1
 Computing element 2...
```

```
Element: 4
 Computing element 3...
Element: 9
 Computing element 4...
Element: 16
 Computing element 5...
Element: 25
Second access to lazyNumbers:
Element: 1
Element: 4
Element: 9
Element: 16
Element: 25
```

Perceba que, na primeira iteração, os elementos da lista são calculados e impressos. Já na segunda iteração, os valores já estão cacheados, evitando reavaliações desnecessárias. Isso demonstra a eficiência das listas preguiçosas em situações onde o custo de computação é alto ou quando nem todos os elementos da lista são necessários num primeiro momento.

Estudo de caso: listagem de números primos

Uma das vantagens das listas preguiçosas é a capacidade de tornar alguns algoritmos mais eficientes em termo de tempo e espaço. A avaliação preguiçosa permite reduzir o número de avaliações e o espaço ocupado na memória, principalmente nos algoritmos que envolvem composição de funções e recursão estrutural. Vamos explorar essa ideia com um exemplo prático de listagem de números primos em um intervalo específico.

VERSÃO ANSIOSA Por exemplo, considere o algoritmo ansioso para listar os primeiros k números primos no intervalo $[x, y]$, conforme o código 7.23. Nessa versão, optamos por projetar o algoritmo usando recursão estrutural sobre os inteiros do intervalo, representados por uma lista.

Código 7.23: Exemplo de listagem de primos (versão ansiosa)

```scala
def isPrime(n: Int): Boolean = {
  n >= 2 && (2 to math.sqrt(n).toInt).forall(n % _ != 0)
}

def range(start: Int, end: Int): List[Int] = {
    if (start > end) Nil
    else start :: range(start + 1, end)
}

def kPrimesInRange(x: Int, y: Int, k: Int): List[Int] = {
    range(x, y).filter(isPrime).take(k).toList
}
```

Esse código define uma função `kPrimesInRange` que gera uma lista de números primos no intervalo de $[x, y]$ e limita o resultado a k elementos. No entanto, essa abordagem não é eficiente. Para compreender a origem dessa ineficiência, vamos

introduzir efeitos colaterais em algumas das funções, incluindo prints para acompanhar o processo, conforme o código 7.24.

Código 7.24: Traço do exemplo de listagem de primos não preguiçosos

```
def isPrime(n: Int): Boolean = {
  println(s"Checking if $n is prime")
  n >= 2 && (2 to math.sqrt(n).toInt).forall(n % _ != 0)
}

def range(start: Int, end: Int): List[Int] = {
    if (start > end) Nil
    else {
        println(s"Generating number $start")
        start :: range(start + 1, end)
    }
}
def kPrimesInRange(x: Int, y: Int, k: Int): List[Int] = {
  println(s"Finding first $k primes in range [$x, $y]")
  range(x, y).filter(isPrime).take(k).toList
}
@main def run = {
    println(kPrimesInRange(1, 15, 5))
}
```

```
Finding first 5 primes in range [1, 15]
Generating number 1
Generating number 2
Generating number 3
Generating number 4
Generating number 5
Generating number 6
Generating number 7
Generating number 8
Generating number 9
Generating number 10
Generating number 11
Generating number 12
Generating number 13
Generating number 14
Generating number 15
```

```
Checking if 1 is prime...
Checking if 2 is prime...
Checking if 3 is prime...
Checking if 4 is prime...
Checking if 5 is prime...
Checking if 6 is prime...
Checking if 7 is prime...
Checking if 8 is prime...
Checking if 9 is prime...
Checking if 10 is prime...
Checking if 11 is prime...
Checking if 12 is prime...
Checking if 13 is prime...
Checking if 14 is prime...
Checking if 15 is prime...
List(2, 3, 5, 7, 11)
```

Perceba que, para encontrar os primeiros 5 números primos no intervalo de 1 a 15, o algoritmo gera todos os números do intervalo. Além disso, todos os números do intervalo são avaliados para verificar se são primos, mesmo que não sejam necessários para o resultado final. Isso resulta em um número excessivo de avaliações, o que pode ser ineficiente em termos de tempo e espaço. Precisamos de uma abordagem mais eficiente que permita diminuir esse custo.

VERSÃO PREGUIÇOSA Podemos reimplementar a função `kPrimesInRange` usando `LazyList` para tornar o algoritmo mais eficiente, conforme o código 7.25.

Código 7.25: Exemplo de listagem de primos com `LazyList`

```
def rangeLazy(start: Int, end: Int): LazyList[Int] = {
    if (start > end) LazyList.empty
    else start #:: rangeLazy(start + 1, end)
}

def kPrimesInRange(x: Int, y: Int, k: Int): LazyList[Int] = {
  rangeLazy(x, y).filter(isPrime).take(k)
}
```

Perceba que, na nova implementação, usamos `LazyList` para gerar a lista de números no intervalo de $[x, y]$. O algoritmo `kPrimesInRange` sofreu quase nenhuma alteração, exceto pela substituição de `List` por `LazyList`: como essas duas estruturas compartilham a mesma API, podemos usar `LazyList` como se fosse uma lista comum.

Para nos convencermos da eficiência dessa abordagem, vamos novamente tirar um pouco da pureza de algumas funções, colocando prints para acompanhar o processo, conforme o código 7.26.

Código 7.26: Traço do exemplo de listagem de primos com `LazyList`

```
def rangeLazy(start: Int, end: Int): LazyList[Int] = {
  if (start > end) LazyList.empty
  else {
    println(s"Generating number $start")
    start
  } #:: rangeLazy(start + 1, end)
}

def kPrimesInRangeLazy(x: Int, y: Int, k: Int): LazyList[Int] = {
  println(s"Finding first $k primes in range [$x, $y]")
  rangeLazy(x, y).filter(isPrime).take(k)
}

@main def testLazyPrimes(): Unit = {
  println(kPrimesInRangeLazy(1, 15, 5).toList)
}
```

```
Finding first 5 primes in range [1, 15]
Generating number 1
Checking if 1 is prime...
Generating number 2
Checking if 2 is prime...
Generating number 3
Checking if 3 is prime...
Generating number 4
Checking if 4 is prime...
Generating number 5
Checking if 5 is prime...
Generating number 6
Checking if 6 is prime...
Generating number 7
Checking if 7 is prime...
Generating number 8
Checking if 8 is prime...
Generating number 9
Checking if 9 is prime...
Generating number 10
Checking if 10 is prime...
Generating number 11
Checking if 11 is prime...
List(2, 3, 5, 7, 11)
```

Perceba que, na nova implementação, apenas os números necessários para solucionar o problema são gerados e avaliados. Além disso, eles são gerados apenas no momento correto, evitando a criação de uma lista completa de números no intervalo. Outra característica interessante é que as operações se intercalam, ou seja, cada número passa por todas as operações antes do próximo ser avaliado.

Intercalação de operações A intercalação de operações durante a composição de funções sobre uma lista preguiçosa ocorre devido à natureza de computação sob demanda dessas estruturas. A composição de funções do código 7.26 ocorre na seguinte linha:

```
1 rangeLazy(x, y).filter(isPrime).take(k)
```

A sequência de operações ocorre na seguinte ordem:

1. `rangeLazy(1, 15)` gera lista preguiçosa
2. `filter(isPrime)` gera lista preguiçosa filtrada
3. `take(5)` solicita (força) o primeiro elemento para `filter`
4. `filter` solicita o primeiro elemento para `rangeLazy`
5. `rangeLazy` gera o primeiro elemento e o passa para `filter`
6. `filter` avalia se o elemento é primo
7. Se for primo, `filter` passa o elemento para `take`
8. `take` solicita o próximo elemento para `filter`
9. O processo se repete até que `take` tenha recebido k elementos ou que `filter` não tenha mais elementos para processar.

Operações intermediárias e operações terminais Quando efetuamos a composição de funções sobre uma `LazyList`, é importante diferenciar entre *operações intermediárias* e *operações terminais*. As operações intermediárias, como `map`, `filter` e `flatMap`, `take`, etc., são aquelas que retornam uma nova `LazyList` e não forçam a avaliação imediata dos elementos. Essas operações não solicitam a avaliação dos elementos da lista, mas sim criam uma nova lista que será avaliada sob demanda.

As operações que consomem a lista, como `toList`, `foreach` ou `fold`, `reduce`, `sum`, etc., são chamadas de *operações terminais*. Essas operações forçam a avaliação da lista preguiçosa e iniciam o processo de computação sob demanda. Por exemplo, `toList` irá forçar a avaliação de todos os elementos da lista, resultando em uma lista ansiosa.

Pipelines e streams A intercalação de operações em listas preguiçosas é uma característica fundamental que permite a construção de pipelines e streams. Essa

abordagem é especialmente útil para processar grandes volumes de dados ou fluxos contínuos, onde a eficiência e a economia de recursos são essenciais.

Um **pipeline** é uma sequência de operações encadeadas, onde a saída de uma operação é a entrada da próxima. Em Scala, podemos construir pipelines por meio da chamada encadeada de funções de ordem superior intermediárias, como `map`, `filter` e `flatMap`. Por exemplo, podemos criar um pipeline para processar uma lista de números, aplicando uma série de transformações e filtragens, conforme o código 7.27.

Código 7.27: Exemplo de pipeline com `LazyList`

```
val numbers = (1 to 15).to(LazyList)
val evenSquares = numbers
  .filter(_ % 2 == 0) // Filtra números pares
  .map(n => n * n) // Eleva ao quadrado
  .take(5) // Pega os primeiros 5 elementos

@main def runPipeline = {
    println(evenSquares.toList) // Imprime a lista resultante
}
```

```
List(4, 16, 36, 64, 100)
```

Um **stream** é um fluxo contínuo de dados que pode ser processado de forma incremental. Em Scala, podemos criar streams usando `LazyList` para representar sequências infinitas ou finitas de dados que são avaliados sob demanda. Por exemplo, podemos criar um stream de números aleatórios ou de eventos em tempo real, conforme o código 7.28.

Código 7.28: Exemplo de stream com `LazyList`

```
val randomNumbers: LazyList[Int] =
  ↪ LazyList.continually(scala.util.Random.nextInt())

@main def runStream = {
    val firstTenRandomNumbers = randomNumbers.take(10).toList
    println(firstTenRandomNumbers) // Imprime os primeiros 10 números
     ↪ aleatórios
}
```

```
List(123456, 789012, 345678, 901234, 567890, 234567, 890123, 456789, 123456, 789012)
```

7.3.1 Listas infinitas

Uma possibilidade intrigante das listas preguiçosas é a capacidade de criar listas infinitas. Uma lista infinita é uma lista preguiçosa cuja cauda é gerada por uma recursão infinita. Isso é possível pois, como as listas preguiçosas são avaliadas sob demanda, podemos definir uma lista que nunca termina, de modo que os elementos

são gerados apenas quando necessários. Essa característica é especialmente útil para trabalhar com fluxos de dados contínuos.

Em Scala, podemos criar listas infinitas usando `LazyList` e a recursão estrutural. Por exemplo, podemos criar uma lista infinita de números inteiros, conforme o código 7.29.

Código 7.29: Exemplo de lista infinita com LazyList

```
val infiniteNumbers: LazyList[Int] = LazyList.from(1)
@main def runInfiniteList = {
    println(infiniteNumbers.take(10).toList) // Imprime os primeiros 10 números
}
```

```
List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Além do construtor `from`, podemos criar listas infinitas usando outras funções de ordem superior, como `iterate` e `continually`. Por exemplo, podemos criar uma lista infinita de números aleatórios, conforme o código 7.30.

Código 7.30: Exemplo de lista infinita de números aleatórios

```
val infiniteRandomNumbers: LazyList[Int] = LazyList.continually(scala.util.Random.nextInt())
@main def runInfiniteRandomList = {
    println(infiniteRandomNumbers.take(10).toList) // Imprime os primeiros 10 números aleatórios
}
```

```
List(123456, 789012, 345678, 901234, 567890, 234567, 890123, 456789, 123456, 789012)
```

Ímpares e pares Podemos usar listas infinitas para gerar sequências de números ímpares e pares, conforme o código 7.31.

Código 7.31: Exemplo de lista infinita de números ímpares e pares

```
val infiniteOdds: LazyList[Int] = LazyList.from(1).filter(_ % 2 != 0)
val infiniteEvens: LazyList[Int] = LazyList.from(0).filter(_ % 2 == 0)
@main def runInfiniteOddEven = {
    println(infiniteOdds.take(10).toList) // Imprime os primeiros 10 números ímpares
    println(infiniteEvens.take(10).toList) // Imprime os primeiros 10 números pares
}
```

```
List(1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21)
List(0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20)
```

Fibonacci infinito Outro exemplo interessante é a geração da sequência de Fibonacci, que pode ser feita de forma infinita usando listas preguiçosas, conforme

o código 7.32.

Código 7.32: Exemplo de lista infinita de Fibonacci

```
def makeFibonacci: LazyList[Int] = {
    def fibs(a: Int, b: Int): LazyList[Int] = {
        a #:: fibs(b, a + b)
    }
    fibs(0, 1)
}
@main def runFibonacci = {
    println(makeFibonacci.take(10).toList) // Imprime os primeiros 10 números
    ↪  de Fibonacci
}
```

```
List(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55)
```

Um aspecto fascinante da lista infinita de Fibonacci é que ela tem a mesma complexidade de tempo e espaço que a versão recursiva na cauda com memoização, ou seja, $O(n)$ para tempo e $O(n)$ para espaço. Isso ocorre porque a lista preguiçosa avalia cada elemento apenas uma vez e armazena o resultado, evitando reavaliações desnecessárias.

RECURSÃO INFINITA PREGUIÇOSA A função fibonacci retorna uma lista preguiçosa infinita. Para definir uma lista infinita, usamos uma recursão infinita, ou seja, uma recursão sem caso base. Na presença de avaliação preguiçosa, apenas o caso recursivo é suficiente para definir o problema, pois os nós da lista serão materializados sob demanda.

No entanto, é preciso ter cuidado para não aplicar operações que exijam a materialização de toda a lista, como `toList` ou `length`. Essas operações podem levar a um loop infinito ou a um estouro de pilha, pois tentam avaliar todos os elementos da lista infinita. Por exemplo, o código 7.33 tenta calcular o comprimento de uma lista infinita, resultando em um loop infinito.

Código 7.33: Exemplo de erro com lista infinita

```
@main def runInfiniteError = {
    println(fibonacci.length) // Isso causará um loop infinito
}
```

```
Exception in thread "main" java.lang.StackOverflowError
   at scala.collection.immutable.LazyList.length(LazyList.scala:100)
   at scala.collection.immutable.LazyList.length$(LazyList.scala:100)
   at scala.collection.immutable.LazyList$$anon$1.length(LazyList.scala:0)
```

Podemos evitar esse problema limitando o número de elementos avaliados ou usando operações que não exigem a materialização completa da lista. Por exemplo, podemos usar `take` para obter apenas os primeiros n elementos da lista infinita, conforme o código 7.34.

Código 7.34: Exemplo seguro com lista infinita

```
@main def runInfiniteSafe = {
    println(fibonacci.take(10).toList) // Imprime os primeiros 10 números de
    ↪ Fibonacci
}
```

```
List(0, 1, 1, 2, 3, 5, 8, 13, 21, 34)
```

7.4 Resumo

Neste capítulo, exploramos o conceito de avaliação preguiçosa em Scala, que é uma técnica poderosa para otimizar o desempenho e a eficiência do código. A avaliação preguiçosa permite que a avaliações de expressões seja efetuada sob demanda, evitando cálculos desnecessários e economizando recursos computacionais. Vimos como a avaliação preguiçosa pode ser implementada por meio de chamadas por necessidade, listas preguiçosas (`LazyList`) e recursão infinita.

Discutimos também a importância de entender as diferenças entre avaliação preguiçosa e avaliação ansiosa, e como isso pode impactar a eficiência do código. Além disso, exploramos o conceito de listas infinitas e como elas podem ser usadas para criar sequências de dados que são avaliadas sob demanda, permitindo a construção de algoritmos eficientes e escaláveis.

Por fim, vimos como a avaliação preguiçosa pode ser combinada com processamento paralelo para otimizar ainda mais o desempenho do código, aproveitando os recursos de múltiplos núcleos de CPU. Essa combinação é especialmente útil em cenários de big data e processamento em tempo real, onde a latência e a escalabilidade são críticas.

7.4.1 Recursos adicionais

Para leitores interessados em aprofundar seus conhecimentos sobre avaliação preguiçosa, recomendamos a exploração dos seguintes tópicos:

- **Vazamentos de memória (*space leaks*)**: Compreenda como propriedades preguiçosas podem inadvertidamente manter referências a grandes estruturas de dados, causando vazamentos de memória, e aprenda técnicas para mitigá-los.
- **Segurança de threads com `lazy val`**: Explore os mecanismos internos de sincronização do Scala para propriedades preguiçosas e como isso afeta a performance em ambientes concorrentes.
- **Views e operações preguiçosas**: Investigue o uso de `.view` em coleções do Scala para criar pipelines de transformação verdadeiramente preguiçosos sem materializações intermediárias.
- **Avaliação preguiçosa com mônadas**: Descubra como a avaliação preguiçosa se integra com `for-comprehensions` e estruturas monádicas para criar computações mais expressivas.

- **Estruturas de dados preguiçosas customizadas**: Aprenda a implementar suas próprias árvores binárias preguiçosas, grafos infinitos e outras estruturas de dados avançadas.
- **Benchmarking e profiling**: Desenvolva habilidades para medir e comparar a performance de código preguiçoso versus ansioso usando ferramentas como JMH (*Java Microbenchmark Harness*).
- **Bibliotecas de streaming**: Explore bibliotecas especializadas como Akka Streams, FS2 e ZIO Stream, que oferecem abstrações mais poderosas para processamento de fluxos de dados.
- **Padrões de inicialização preguiçosa**: Estude padrões avançados como *lazy singleton*, injeção de dependência preguiçosa e inicialização condicional em sistemas distribuídos.

Esses tópicos representam áreas de estudo mais avançadas que complementam os conceitos fundamentais apresentados neste capítulo, oferecendo oportunidades para aplicar a avaliação preguiçosa em cenários mais complexos e especializados.

Aviso de licença de uso