Capítulo 3

Funções: recursos básicos

No paradigma funcional, as funções são o principal mecanismo de abstração, usadas para encapsular comportamentos e transformar dados. Neste capítulo, exploramos como definir e utilizar funções em Scala, incluindo conceitos como funções anônimas (lambdas), currying e composição de funções.

3.1 Sintaxe básica

Definição de funções Para definir uma função em Scala, usamos a palavra-chave `def`, seguida pelo nome da função, os parâmetros entre parênteses, o tipo de retorno (opcional) e o corpo da função. O código 3.1 mostra uma função simples que soma dois números inteiros.

Código 3.1: Definição de uma função simples em Scala

```
1 def sum(a: Int, b: Int): Int = a + b
```

Vale ressaltar alguns pontos sobre a sintaxe:

- **Parâmetros**: Devem ser obrigatoriamente tipados. No exemplo, `a` e `b` são do tipo `Int`.
- **Tipo de retorno**: Especificado após os parâmetros, separado por dois pontos. No exemplo, a função `sum` retorna um `Int`.
- **Inferência do tipo de retorno**: O tipo de retorno é opcional, pois pode ser inferido automaticamente. Ainda assim, é boa prática especificá-lo para melhorar a documentação e garantir o tipo esperado.
- **Comando `return`**: Opcional em Scala. Se o corpo for uma única expressão, seu valor é retornado automaticamente. Se for um bloco, o valor da última expressão é retornido. O uso explícito do comando `return` é comum em situações de controle de fluxo imperativo, pois é um tipo de salto. Em programas funcionais, o uso de `return` é desencorajado.
- **Corpo da função**: Pode ser uma expressão única ou um bloco de código.

FUNÇÕES VS. PROCEDIMENTOS Em Scala, uma função retorna um valor, enquanto um procedimento executa uma ação, mas não retorna valor, sendo definido como uma função que retorna `Unit` (equivalente ao `void` de outras linguagens). Procedimentos geralmente produzem efeitos colaterais e, por isso, recomenda-se evitá-los, preferindo funções puras sempre que possível. O código 3.2 mostra um exemplo de procedimento.

Código 3.2: Definição de um procedimento em Scala

```scala
def readThenPrint(): Unit =
  val a = scala.io.StdIn.readInt()
  val b = scala.io.StdIn.readInt()

  println(s"A soma é: ${sum(a, b)}")
```

```
A soma é: 8
```

FUNÇÕES VS. MÉTODOS Em Scala, métodos são funções associadas a objetos ou classes. Até Scala 2, funções nomeadas eram definidas como métodos dentro de classes, objetos ou traits; funções anônimas eram armazenadas em variáveis. A partir do Scala 3, é possível definir funções nomeadas no nível superior do programa (*top level definitions*). Na prática, o termo "função" costuma ser usado para ambos os conceitos. Neste texto, consideramos:

- Função: termo geral para qualquer bloco de código que recebe parâmetros e retorna um valor, podendo ser definida no nível superior ou dentro de classes, objetos ou traits.
- Método: função associada a um objeto ou classe.

FUNÇÕES ANINHADAS É possível definir funções dentro de outras funções. Essas funções internas, chamadas de funções aninhadas, podem acessar os parâmetros e variáveis da função externa. O código 3.3 ilustra esse conceito.

Código 3.3: Definição de uma função aninhada em Scala

```scala
def f1(a: Int, b: Int): Int =
  def f2(c: Int): Int =
    a + b + c
  f2(a - b)

def testF1(): Unit =
  val a = 5
  val b = 3
  println(s"O resultado de f1($a, $b) é: ${f1(a, b)}")
```

```
O resultado de f1(5, 3) é: 7
```

APLICAÇÃO DE FUNÇÕES Aplicar uma função consiste em chamá-la com argumentos específicos, escrevendo o nome da função seguido dos argumentos entre parênteses. O código 3.4 mostra um exemplo.

Código 3.4: Aplicação de uma função em Scala

```scala
val result = sum(3, 5)
println(s"A soma é: $result")
```

```
$ scala Funcoes.scala
A soma é: 8
```

USO DE PARÊNTESES Em Scala, funções sem parâmetros podem ser definidas e chamadas sem parênteses. O código 3.5 mostra uma função sem parâmetros e sua aplicação.

Código 3.5: Uso de parênteses em funções sem parâmetros

```scala
def doublePi = 2 * math.Pi

println(s"O dobro de Pi é: ${doublePi}")
```

```
$ scala Funcoes.scala
O dobro de Pi é: 6.283185307179586
```

Scala adota convenções para funções sem argumentos:

- **Sem parênteses**: Consideradas "valores", não têm efeitos colaterais e são usadas para encapsular valores constantes ou expressões puras.
- **Com parênteses**: Consideradas "procedimentos", podem ter efeitos colaterais, como ler dados, imprimir ou gerar valores aleatórios.

Funções sem argumentos são mais comuns quando há efeitos colaterais ou para encapsular valores constantes. Nos dois primeiros casos, devem ser evitadas em programação funcional pura. No último caso, podem ser ineficientes, pois a expressão é recalculada a cada chamada.

PARÂMETROS IMUTÁVEIS POR PADRÃO Em Scala, os parâmetros de funções são imutáveis por padrão, ou seja, não podem ser alterados dentro da função. Ou seja, é como se, implicitamente, os parâmetros da função fossem declarados como `val`. Isso promove a segurança e previsibilidade do código, evitando efeitos colaterais indesejados. Caso seja feita uma tentativa de alterar o estado de um parâmetro, o compilador gerará um erro. O código 3.6 ilustra essa característica.

Código 3.6: Parâmetros imutáveis por padrão em Scala

```scala
def increment(x: Int): Int = {
  x = x + 1 // Isso causará um erro de compilação
  x
}
```

```
error: reassignment to val
  x = x + 1 // Isso causará um erro de compilação
  ^
```

PARÂMETROS COM VALORES PADRÃO É possível definir valores padrão para parâmetros, permitindo chamadas com menos argumentos. O código 3.7 mostra um exemplo.

Código 3.7: Definição de uma função com valores padrão

```scala
def multiply(a: Int = 1, b: Int = 1): Int =
  a * b

println(multiply(2, 3))
println(multiply())
println(multiply(4))
```

```
$ scala Funcoes.scala
6
1
4
```

PARÂMETROS NOMEADOS Permitem especificar argumentos independentemente da ordem dos parâmetros, útil em funções com muitos parâmetros ou valores padrão. Veja o código 3.8.

Código 3.8: Uso de parâmetros nomeados em Scala

```scala
def fullName(first: String = "Fulano", last: String): String =
  s"$first $last"

def testFullName(): Unit =
  println(s"Nome completo: ${fullName(first = "João", last = "Silva")}")
  println(s"Nome completo: ${fullName(last = "Silva", first = "João")}")
  println(s"Nome completo: ${fullName(last = "Silva")}")
```

```
$ scala Funcoes.scala
Nome completo: João Silva
Nome completo: João Silva
Nome completo: Fulano Silva
```

PARÂMETROS VARIÁDICOS Funções podem aceitar um número variável de argumentos usando o operador `*`. O código 3.9 mostra um exemplo.

Código 3.9: Definição de uma função com parâmetros variádicos

```scala
def doubleAll(numbers: Int*): Seq[Int] =
  numbers.map(n => n * 2)

```

```scala
def testDoubleAll(): Unit = {
  println(s"Números dobrados: ${doubleAll(1, 2, 3, 4, 5)}")
}
```

```
$ scala Funcoes.scala
Números dobrados: ArraySeq(2, 4, 6, 8, 10)
```

3.2 Funções anônimas (lambdas)

Funções anônimas, ou *lambdas*, não possuem nome explícito. São úteis para armazenar funções em variáveis, passá-las como argumentos ou retorná-las de outras funções, sendo fundamentais para funções de ordem superior.

Definição de funções anônimas As funções anônimas são definidas pela sintaxe:

```
(A: TipoA, B: TipoB, ..., N: TipoN) => Corpo
```

Onde:

- `A`, `B`, ..., `N`: parâmetros da função.
- `TipoA`, `TipoB`, ..., `TipoN`: tipos dos parâmetros.
- `Corpo`: expressão ou bloco executado na aplicação.

O tipo de retorno é inferido pelo compilador. Não é possível explicitá-lo diretamente nessa sintaxe. O código 3.10 mostra exemplos.

Código 3.10: Definição de funções anônimas

```scala
def typingTheLambda() = {
    val identity = (x: Int) => { x }
    val add = (x: Int, y: Int) => { x + y }
    val multiply = (x: Int, y: Int) => { x * y }

    println(s"Identidade de 5: ${identity(5)}")
    println(s"Soma de 5 e 3: ${add(5, 3)}")
    println(s"Multiplicação de 5 e 3: ${multiply(5, 3)}")
}

@main def run(): Unit = {
    typingTheLambda()
}
```

```
Identidade de 5: 5
Soma de 5 e 3: 8
Multiplicação de 5 e 3: 15
```

3.3 Tipo-função

Ao atribuir uma função anônima a uma variável, ela assume um tipo-função, que descreve a assinatura da função (tipos dos parâmetros e do retorno):

```
(Tipo1, Tipo2, ..., TipoN) => TipoRetorno
```

Onde:

- `Tipo1`, `Tipo2`, ..., `TipoN` são os tipos dos parâmetros da função.
- `TipoRetorno` é o tipo do valor retornado pela função.

Alguns exemplos de tipos-função são:

- `Int => Int`: Uma função que recebe um inteiro e retorna um inteiro.
- `String => Int`: Uma função que recebe uma string e retorna um inteiro.
- `(Int, Int) => Int`: Uma função que recebe dois inteiros e retorna um inteiro.
- `(String, String) => String`: Uma função que recebe duas strings e retorna uma string.

Tipagem de variáveis Para tipar uma variável com um tipo-função, basta usar a sintaxe usual de tipagem, aplicando o tipo-função como o tipo da variável. O exemplo de código 3.11 mostra como tipar variáveis com um tipo-função, para que elas possam receber funções anônimas como valor.

Código 3.11: Inferência de tipos em funções anônimas

```scala
val identity: (Int) => Int = x => x
val add: (Int, Int) => Int = (x, y) => x + y
val multiply: (Int, Int) => Int = (x, y) => x * y

@main def run(): Unit = {
    println(s"Identidade de 5: ${identity(5)}")
    println(s"Soma de 5 e 3: ${add(5, 3)}")
    println(s"Multiplicação de 5 e 3: ${multiply(5, 3)}")
}
```

```
Identidade de 5: 5
Soma de 5 e 3: 8
Multiplicação de 5 e 3: 15
```

Inferência de tipos Ao tipar a variável com um tipo-função, a lambda pode ser atribuída sem tipagem explícita, pois o compilador infere os tipos pelo contexto.

FUNÇÕES ANÔNIMAS COMO OBJETOS Em Scala, funções anônimas são representadas como objetos. Os tipos-função são definidos como *traits*, semelhantes a interfaces em Java. Por exemplo, `Int => Int` é açúcar sintático para `Function1[Int, Int]`. O código 3.12 mostra uma implementação simplificada.

Código 3.12: Implementação simplificada do trait `Function2`

```scala
trait Function2[A, B, R] {
  def apply(a: A, b: B): R
}
```

Existem traits para diferentes aridades (`Function0` até `Function22`), cada um com método `apply`. Podemos obter o trait de uma lambda com `getClass`. Veja o código 3.13.

Código 3.13: Obtenção do trait de uma função anônima

```scala
def getTraits() = {
    val identity: Int => Int = x => x
    val add: (Int, Int) => Int = (x, y) => x + y
    val multiply: (Int, Int) => Int = (x, y) => x * y

    identity.getClass.getInterfaces.foreach { i =>
        println(s"Identity interfaces: ${i.getName}")
    }
    add.getClass.getInterfaces.foreach { i =>
        println(s"Add interfaces: ${i.getName}")
    }
    multiply.getClass.getInterfaces.foreach { i =>
        println(s"Multiply interfaces: ${i.getName}")
    }
}
```

```
Identity interfaces: scala.runtime.java8.JFunction1$mcII$sp
Add interfaces: scala.runtime.java8.JFunction2$mcIII$sp
Multiply interfaces: scala.runtime.java8.JFunction2$mcIII$sp
```

Como podemos observar, retirando informações de tempo de execução, o trait `Function1` é representado pelo nome `JFunction1`, e o trait `Function2` pelo nome `JFunction2`. Esses nomes são gerados pelo compilador e podem variar entre versões da linguagem.

Em suma, vemos aqui muitas similaridades com o modo como Java define interfaces funcionais. De fato, há evidências de que os projetistas de Java 8.0 se inspiraram em Scala para trazer os recursos de programação funcional para a linguagem.

UTILIDADE DAS LAMBDAS As lambdas são importantes na programação funcional, pois permitem algumas aplicações importantes:

- **Funções de ordem superior**. Permitem passar funções como argumentos ou retorná-las de outras funções, facilitando a composição e abstração de comportamentos. Veremos exemplos de funções de ordem superior na seção **??**.

- **Tratamento de eventos**. Permitem definir callbacks para eventos assíncronos, como cliques de botões ou respostas de requisições HTTP, facilitando a programação reativa.

- **Programação assíncrona**. Permitem escrever código que pode ser executado de forma não bloqueante, melhorando a eficiência e a capacidade de resposta de aplicações.

- **Closures**. Permitem capturar o contexto em que foram definidas, acessando variáveis locais mesmo fora do escopo original. Isso é útil para criar funções que mantêm estado ou comportamento específico.

- **Funções "descartáveis"** Permitem definir funções que são usadas apenas uma vez, sem a necessidade de nomeá-las. Isso é útil para operações simples ou temporárias, como filtros ou transformações em coleções.

- entre outras aplicações.

3.4 Funções de ordem superior

Funções de ordem superior recebem outras funções como argumentos ou as retornam como resultado. São essenciais na programação funcional e permitem criar abstrações poderosas. A seguir, exemplos de funções de ordem superior da biblioteca padrão de Scala.

Método `List.map` O método `map`, da classe `List`, é um exemplo clássico de função de ordem superior. Esse método é usado para aplicar uma função a cada elemento de uma coleção, retornando uma nova coleção com os resultados. O método map é definido no trait `IterableOps`, com isso todos os seus subtipos, como `List`, `Set`, etc., herdam esse método. O exemplo de código 3.14 mostra como usar a função `map` para aplicar transformações a uma lista de inteiros.

Código 3.14: Uso da função `map` em Scala

```scala
val numbers = List(1, 2, 3, 4, 5)

val doubled = numbers.map(n => n * 2)
val squared = numbers.map(n => n * n)

@main def run(): Unit = {
  println(s"Números originais: $numbers")
  println(s"Números dobrados: $doubled")
  println(s"Números ao quadrado: $squared")
}
```

```
Números originais: List(1, 2, 3, 4, 5)
Números dobrados: List(2, 4, 6, 8, 10)
Números ao quadrado: List(1, 4, 9, 16, 25)
```

Inferência de tipos Note que que o argumento passado para a função `map` não precisa ser tipado, pois o compilador pode inferir os tipos usados na lambda a partir do contexto. Ao consultarmos a documentação[1] do método `map` de `List`, constateremos que sua assinatura é como segue:

```
def map[B](f: A => B): List[B]
```

Onde `A` é o tipo dos elementos da lista original e `B` é o tipo dos elementos da nova lista. Tanto `A` e `B` são tipos genéricos que serão especializados pelo compilador quando a função for aplicada. O compilador infere os tipos `A` e `B` a partir do contexto, ou seja, a partir do tipo da lista original e do tipo do valor retornado pela função passada como argumento.

Função `List.filter` O método `filter` de `List` é outra função de ordem superior, que permite filtrar elementos de uma coleção com base em um critério definido por uma outra função. Ela recebe como argumento um *predicado*, isto é, uma função que retorna um valor booleano. Como resultado da filtragem, a função retorna uma nova coleção contendo apenas os elementos que, quando aplicados à lambda passada como argumento, retornam verdadeiro. A assinatura do método `filter` é como segue:

```
def filter(p: A => Boolean): List[A]
```

O exemplo de código 3.15 mostra como usar a função `filter` para obter apenas os números pares de uma lista.

Código 3.15: Uso da função `filter` em Scala

```scala
val numbers = List(1, 2, 3, 4, 5)
val evenNumbers = numbers.filter(n => n % 2 == 0)
val oddNumbers = numbers.filter(n => n % 2 != 0)
val greaterThanFive = numbers.filter(n => n > 5)

@main def run(): Unit = {
  println(s"Números originais: $numbers")
  println(s"Números pares: $evenNumbers")
  println(s"Números ímpares: $oddNumbers")
  println(s"Números maiores que cinco: $greaterThanFive")
}
```

```
Números originais: List(1, 2, 3, 4, 5)
Números pares: List(2, 4)
Números ímpares: List(1, 3, 5)
Números maiores que cinco: List()
```

Método `List.reduce` O método `reduce` é um pouco mais complexo, pois ele reduz uma coleção a um único valor, aplicando uma função de agregação a cada elemento da coleção. A função passada como argumento deve receber dois parâmetros e retornar um único valor, de acordo com a seguinte assinatura:

[1] https://www.scala-lang.org/api/3.x/scala/collection/immutable/List.html#map-fffff812

```
def reduce[B >: A](op: (B, A) => B): B
```

Onde `A` é o tipo dos elementos da lista original e `B` é o tipo do valor retornado pela função. Lembrando que os tipos são genéricos e serão determinados por inferência. A notação `B >: A` é um pouco avançada, mas vale esclarecer: indica que o tipo `B` deve ser um supertipo do tipo `A`, ou seja, a função pode receber valores do tipo `A` e retornar valores do tipo `B`.

A lambda do tipo `(B, A) => B` deve receber dois parâmetros: o primeiro é o acumulador (o valor parcial da redução) e o segundo é o elemento atual da coleção. A função deve retornar o novo valor do acumulador após aplicar a operação de agregação. Por padrão, o método adota o primeiro elemento da coleção como o valor inicial do acumulador e aplica a função de agregação a cada elemento subsequente. O código 3.16 mostra exemplos de uso do método `reduce` para reduzir os valores de uma lista de inteiros.

Código 3.16: Uso do método `reduce` em Scala

```scala
val numbers = List(1, 2, 3, 4, 5)
val sum = numbers.reduce((a, b) => a + b)
val product = numbers.reduce((a, b) => a * b)

@main def run(): Unit = {
  println(s"Números originais: $numbers")
  println(s"Soma: $sum")
  println(s"Produto: $product")
}
```

```
Números originais: List(1, 2, 3, 4, 5)
Soma: 15
Produto: 120
```

Traço da redução. O método `reduce` funciona da seguinte forma:

1. Inicializa o acumulador com o primeiro elemento da lista.
2. Itera sobre os elementos restantes da lista, aplicando a função de agregação ao acumulador e ao elemento atual.
3. Atualiza o acumulador com o resultado da função de agregação.
4. Retorna o valor final do acumulador após iterar por todos os elementos da lista.

O exemplo de código 3.17 mostra um passo a passo do processo de redução, onde a função de agregação é aplicada a cada elemento da lista.

Código 3.17: Traço do método `reduce` em Scala

```scala
val numbers = List(1, 2, 3, 4, 5)
val sum = numbers.reduce((a, b) => {
  println(s"Acumulador: $a, Elemento atual: $b")
  a + b
```

```
})
@main def run(): Unit = {
  println(s"Números originais: $numbers")
  println(s"Soma: $sum")
}
```

```
Números originais: List(1, 2, 3, 4, 5)
Acumulador: 1, Elemento atual: 2
Acumulador: 3, Elemento atual: 3
Acumulador: 6, Elemento atual: 4
Acumulador: 10, Elemento atual: 5
Soma: 15
```

Casos de borda. É importante ressaltar que o método `reduce` não possui um valor inicial explícito, ou seja, ele assume que o primeiro elemento da coleção é o valor inicial do acumulador. Considerando essa situação, há dois casos de borda:

- **Coleção com um único elemento**: Nesse caso, o método `reduce` inicia o acumulador com o único elemento da coleção e aplica a função de agregação à cauda da lista, que é vazia. Isso faz com que o processo de redução não tenha efeito, e o resultado seja o próprio elemento. O código 3.18 mostra um exemplo de uso do método `reduce` com uma lista contendo apenas um elemento.

Código 3.18: Uso do método `reduce` com uma lista de um único elemento

```
val singleElementList = List(42)
val singleElementSum = singleElementList.reduce((a, b) => a + b)
@main def run(): Unit =
  println(s"Lista com um único elemento: $singleElementList")
  println(s"Soma: $singleElementSum")
```

```
Lista com um único elemento: List(42)
Soma: 42
```

- **Coleção vazia**: Nesse caso, o método `reduce` não pode ser aplicado, pois não há elementos para reduzir. Se tentarmos aplicar o método em uma coleção vazia, ele lançará uma exceção `UnsupportedOperationException`. O código 3.19 mostra um exemplo de uso do método `reduce` com uma lista vazia.

Código 3.19: Uso do método `reduce` com uma lista vazia

```
val emptyList = List.empty[Int]
val emptySum = emptyList.reduce((a, b) => a + b)

@main def run(): Unit =
  println(s"Lista vazia: $emptyList")
```

```
Exception in thread "main" java.lang.UnsupportedOperationException:
    empty.reduceLeft
  at scala.collection.immutable.Nil.reduceLeft(Nil.scala:42)
  at scala.collection.immutable.Nil.reduce(Nil.scala:44)
```

```
  at Funcoes$.run(Funcoes.scala:6)
  at Funcoes.main(Funcoes.scala)
```

MÉTODO `List.foreach` O método `foreach` é uma função de ordem superior que itera sobre os elementos de uma coleção e aplica uma função a cada elemento. Ele é usado principalmente para executar efeitos colaterais, como imprimir valores na tela ou modificar o estado de um objeto. Isso ocorre pois esse método não é uma função pura, mas um comando que retorna `Unit`. Em suma, trata-se de uma estrutura com orientação imperativa. Como nosso foco reside na programação funcional, o uso de `foreach` deve ser evitado sempre que possível. No entanto, é importante conhecer esse método, pois ele é amplamente utilizado em código Scala. O exemplo de código 3.20 mostra como usar o método `foreach` para imprimir os valores de uma lista de inteiros.

Código 3.20: Uso do método `foreach` em Scala

```scala
val numbers = List(1, 2, 3, 4, 5)

def printNumbers(): Unit = {
  numbers.foreach(n => println(s"Número: $n"))
}

@main def run(): Unit = {
    println(s"Números originais: $numbers")
    printNumbers()
}
```

```
Números originais: List(1, 2, 3, 4, 5)
Número: 1
Número: 2
Número: 3
Número: 4
Número: 5
```

3.5 Fechamentos (closures)

Fechamentos, mais comumente conhecidas como *closures*, são um conceito proeminente na programação funcional. Uma closure é a combinação de uma função e o ambiente em que ela foi definida (ou melhor, o seu ambiente léxico). Considere o exemplo do código 3.21.

Código 3.21: Exemplo de closure em Scala

```scala
def f(x: Int) = {
  val y = 2
  (z: Int) => x + y + z
}

```

```
@main def run() = {
  val c = f(3)
  val result = c(4) // This will compute 3 + 2 + 4
  println(s"Result of the closure: $result") // Should print 9
}
```

```
Result of the closure: 9
```

No código 3.21, a função `f` define uma variável `y` e retorna uma função anônima que captura as variáveis `x` e `y`. Quando chamamos `f(3)`, obtemos uma closure que mantém o valor de `x` como 3 e `y` como 2. Assim, quando aplicamos a closure com o argumento 4, ela calcula $3 + 2 + 4$, resultando em 9.

Definições. Para melhor compreender porque o exemplo funciona desse modo, vamos definir alguns conceitos importantes:

- **Escopo léxico**. O escopo léxico é a região do código onde uma variável é visível e acessível. Por exemplo, em Scala, um escopo léxico é definido por blocos de código, como funções, classes e objetos. Variáveis definidas dentro de um escopo léxico só são acessíveis dentro desse escopo. Por exemplo, a variável `y` no código 3.21 só é acessível dentro da função `f`.

- **Cadeia de escopos**. Quando aninhamos escopos léxicos, criamos uma cadeia de escopos. Cada escopo léxico pode acessar variáveis definidas em escopos superiores, mas não em escopos inferiores. No exemplo, a função anônima criada dentro da função `f` pode acessar as variáveis `x` e `y` dessa função.

- **Ambiente léxico**. O ambiente léxico é o conjunto de variáveis e seus valores que estão disponíveis em uma cadeia de escopos. No exemplo, o ambiente léxico da função anônima inclui as variáveis `x` e `y` definidas na função `f`, além da sua própria variável local, `z`.

- **Variáveis livres**. Variáveis livres são aquelas que: i) não foram definidas no escopo léxico da função; ii) estão disponíveis no ambiente léxico da função; e iii) são acessadas no escopo léxico da função. No exemplo, as variáveis `x` e `y` são variáveis livres na função anônima, pois não estão definidas dentro dela, mas estão acessíveis no ambiente léxico da função `f`.

- **Closure**. Uma closure é uma estrutura que contém a definição de uma função e um registro do ambiente léxico contendo as variáveis livres que ela captura. Em outras palavras, uma closure é uma função que ”lembra” o ambiente em que foi criada.

A figura 3.1 ilustra a closure do exemplo 3.21. Note que uma closure é a composição entre a função anônima e o ambiente léxico onde ela foi definida (representado pelas variáveis `x` e `y`).

Tempo de vida. Uma característica importante das closures é que elas “prolongam” o tempo de vida das variáveis livres que capturam. Num cenário sem a interferência de closures, as variáveis `x` e `y` da função `f` seriam destruídas após a

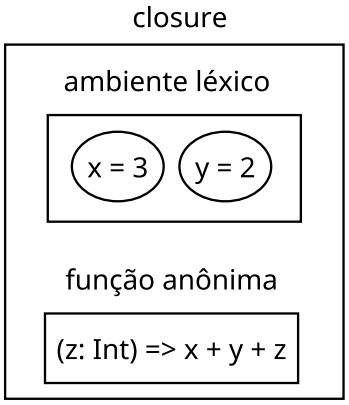

Figura 3.1: Representação gráfica para a closure do exemplo 3.21.

execução da função. No entanto, como a closure captura essas variáveis, elas permanecem acessíveis enquanto a closure existir. Isso permite que as closures mantenham estado entre chamadas, o que é uma característica poderosa na programação funcional.

Uso de closures As closures são amplamente utilizadas em linguagens funcionais. Entre as principais aplicações, podemos destacar:

- **Encapsulamento.** Permitem emular variáveis "privadas" que não podem ser acessadas no escopo externo à função anônima. A função, por outro lado, pode acessar as variáveis livremente.
- **Fábricas de funções.** Por meio de variáveis livres, podemos parametrizar a criação de funções. Com isso, podemos definir funções de ordem superior que retornam funções adaptáveis, caracterizando de fato uma fábrica de funções.
- **Funções com estado**. Em linguagens com mutabilidade, podemos utilizar as variáveis livres para manter estado entre diferentes chamadas da função.
- **Callbacks.** São frequentemente usadas como callbacks em programação assíncrona, permitindo que funções anônimas capturem o contexto e passem informações adicionais para a função de ordem superior que as recebe.
- **Modularização**. Podem ser utilizadas para emular o recurso de módulos: podemos encapsular variáveis e comportamentos em uma closure, criando um escopo isolado que pode ser reutilizado em diferentes partes do código.
- Entre outros.

3.6 Resumo

Neste capítulo, apresentamos os conceitos essenciais sobre funções no paradigma funcional, com ênfase na linguagem Scala. Abordamos a sintaxe para definição

de funções, procedimentos e métodos, além de recursos como parâmetros padrão, nomeados e variádicos. Também discutimos funções anônimas (lambdas), tipos-função e o papel das funções de ordem superior, fundamentais para tratar funções como valores e criar abstrações poderosas na manipulação de coleções.

Foram apresentados exemplos práticos do uso dos métodos `map`, `filter`, `reduce` e `foreach`, evidenciando como a programação funcional facilita a composição de operações e o uso de funções puras. Destacamos ainda a distinção entre funções que retornam valores e procedimentos que produzem efeitos colaterais, recomendando a preferência por funções puras sempre que possível.

O domínio desses conceitos é fundamental para a escrita de código mais expressivo, modular e reutilizável em Scala. Nas próximas aulas, aprofundaremos temas como recursividade e funções de ordem superior avançadas, ampliando ainda mais as possibilidades da programação funcional.

3.7 Exercícios

Os exercícios a seguir devem ser necessariamente resolvidos por meio da utilização das funções de ordem superior: `map`, `filter` e `reduce`.

1. Defina uma função que receba uma lista de números inteiros e retorne outra lista, contendo apenas os números ímpares dobrados. Por exemplo, dada a lista `List(1, 2, 3, 4, 5)`, a função deve retornar `List(2, 6, 10)`.

2. Defina uma função que receba uma lista de strings, e dois inteiros, um contendo um limite inferior e outro contendo um limite superior. A função deve retornar outra lista contendo apenas as strings cujo tamanho esteja entre os limites especificados. Por exemplo, dada a lista `List("Scala", "Java", "Python", "C++")`, com limites 3 e 5, a função deve retornar `List("Java", "C++")`.

3. Defina uma função que receba uma lista de números inteiros e retorne a soma dos quadrados dos números pares. Por exemplo, dada a lista `List(1, 2, 3, 4, 5)`, a função deve retornar 20 (pois $2^2 + 4^2 = 4 + 16 = 20$).

4. As notas dos estudantes de uma turma são representadas por uma lista de tuplas, onde cada tupla contém o nome do estudante e sua nota. Por exemplo:

```
List(
  ("Alice", 8.5),
  ("Bob", 7.0),
  ("Charlie", 9.0),
  ("David", 6.5),
  ("Eve", 8.0)
)
```

   Defina uma função que receba essa lista e retorne a média da turma, considerando apenas os que foram aprovados (notas maiores ou iguais a 7.0). A função deve ter a seguinte assinatura:

```
def averageApproved(students: List[(String, Double)]): Double = {
  // implementação
}
```

5. Um sensor reporta valores de temperatura em graus Celsius anotados com timestamps que indicam o momento em que a medição ocorreu. Esses valores são armazenados em uma lista de tuplas, conforme o exemplo a seguir, que representa seis medições de temperatura:

```
List(
  (1678886400, 25.5),
  (1678972800, 26.1),
  (1679059200, -60.0),
  (1679145600, 24.8),
  (1679232000, 27.0),
  (1679318400, 25.9)
)
```

   Defina uma função que faça o seguinte:

   - Elimine todas as temperaturas negativas
   - Elimine as leituras cujos timestamps sejam menores que um timestamp informado
   - Calcule a média das temperaturas restantes

   A função deve ter a seguinte assinatura:

```
def avgTemperature(temps: List[(Int, Double)], minTimestamp: Int):
↪  Double = {
  // implementação
}
```

   Nota: para acessar os elementos de uma tupla, utilize a notação `x._1` para o primeiro elemento e `x._2` para o segundo elemento, considerando que `x` é a tupla.

6. Defina uma função que receba uma lista de strings e um inteiro, representando um comprimento mínimo e faça o seguinte:

   - Elimine todas as strings cujo comprimento seja menor que o valor informado
   - Retorne a média dos comprimentos das strings restantes.

7. Uma loja online precisa calcular o preço final dos itens no carrinho de compras após a aplicação dos descontos e a inclusão do frete. O frete e o desconto *de cada item* são dependentes do peso desse item, conforme a tabela a seguir:

Peso (kg)	Desconto (%)	Frete (R$)
0.0 - 0.5	5%	1.0
0.6 - 1.0	10%	1.5
1.1 - 2.0	15%	2.0
Acima de 2.0	20%	3.0

Os dados do carrinho são representados por uma lista de tuplas, conforme o exemplo:

```
List(
  (1, 10.0, 0.5),  // (id, preço, peso)
  (2, 20.0, 1.0),
  (3, 15.0, 0.8)
)
```

Defina uma função que calcule o preço final do carrinho, considerando:

- A soma dos preços dos itens
- A aplicação de um desconto sobre o total
- A inclusão do frete, que depende do peso de cada item

A assinatura da função deve ser:

```
def calculateFinalPrice(cart: List[(Int, Double, Double)]): Double = {
  // implementação
}
```

3.8 Soluções dos Exercícios

1 **Solução:**

```
def oddDoubled(data: List[Int]): List[Int] =
  data.filter(x => x % 2 != 0).map(x => x * 2)
```

2 **Solução:**

```
def filterBySize(strings: List[String], minSize: Int, maxSize: Int):
↪  List[String] = {
  strings.filter(s => s.length >= minSize && s.length <= maxSize)
}
```

3 **Solução:**

```
def sumOfSquaresOfEvens(data: List[Int]): Int = {
  val evens = data.filter(x => x % 2 == 0)

```

```
  if (evens.isEmpty) 0
  else evens.map(x => x * x).reduce((x, y) => x + y)
}
```

4 Solução:

```
def averageApproved(students: List[(String, Double)]): Double = {
  val approvedStudents = students.filter(student => student._2 >= 7.0)
  if (approvedStudents.isEmpty) 0.0
  else {
    val approvedGrades = approvedStudents.map(student => student._2)
    val total = approvedGrades.reduce((a, b) => a + b)
    val count = approvedStudents.length
    total / count
  }
}
```

5 Solução:

```
def avgTemperatures(temps: List[(Int, Double)], timestamp: Int): Double
↪  = {
  val filtered = temps.filter(x => x._2 >= 0.0 && x._1 <= timestamp)
  if (filtered.isEmpty) 0.0
  else filtered.map(x => x._2).reduce((x, y) => x + y) /
  ↪  filtered.length
}
```

6 Solução:

```
def avgLengthOfStrings(strings: List[String], minLength: Int): Double =
↪  {
  val filtered = strings.filter(s => s.length >= minLength)
  if (filtered.isEmpty) 0.0
  else filtered.map(s => s.length).reduce((x, y) => x + y).toDouble /
  ↪  filtered.length
}
```

7 Solução:

```
def calculateFinalPrice(cart: List[(Int, Double, Double)]): Double = {
  def discount(peso: Double): Double =
    if (peso <= 0.5) 0.05
    else if (peso <= 1.0) 0.10
    else if (peso <= 2.0) 0.15
    else 0.20

  def frete(peso: Double): Double =
    if (peso <= 0.5) 1.0
    else if (peso <= 1.0) 1.5
```

```
    else if (peso <= 2.0) 2.0
    else 3.0

  val totalDescontado = cart.map(item =>
    item._2 * (1 - discount(item._3))
  ).reduce((a, b) => a + b)

  val totalFrete = cart.map(item =>
    frete(item._3)
  ).reduce((a, b) => a + b)

  totalDescontado + totalFrete
}
```

Aviso de licença de uso