

CENTRO UNIVERSITÁRIO CARIOSA

LUCAS ROCHA GONÇALVES SOARES

APLICAÇÕES DA LINGUAGEM LAMBDA: JAVA 8 E KOTLIN

RIO DE JANEIRO

2018



CENTRO UNIVERSITÁRIO CARIOWA

LUCAS ROCHA GONÇALVES SOARES

APLICAÇÕES DA LINGUAGEM LAMBDA: JAVA 8 E KOTLIN

Trabalho de Conclusão de Curso apresentado
ao Centro Universitário Carioca, como requisito
parcial para a obtenção do grau de Bacharel
em Ciência da Computação.

Orientador: Prof. D.Sc Sérgio Assunção
Monteiro

RIO DE JANEIRO
2018

LUCAS ROCHA GONÇALVES SOARES

APLICAÇÕES DA LINGUAGEM LAMBDA: JAVA 8 E KOTLIN

Trabalho de Conclusão de Curso apresentado
ao Centro Universitário Carioca, como requisito
parcial para a obtenção do grau de Bacharel
em Ciência da Computação.

BANCA EXAMINADORA

Prof. Sérgio Assunção Monteiro, D.Sc – Orientador
Centro Universitário Carioca

Prof. André Luis Avelino Sobral, M. Sc. - Coordenador
Centro Universitário Carioca

Prof. Alberto Tavares da Silva – Professor convidado
Centro Universitário Carioca

AGRADECIMENTOS

Gostaria de agradecer, primeiramente a Deus, por ter me dado forças para conseguir realizar este trabalho de conclusão de curso, nesta tarefa árdua que levou alguns meses para ser concluída.

Em seguida, quero agradecer o suporte dos meus pais, moral e financeiramente, para que a obtenção do bacharelado em Ciência da Computação tivesse sido possível.

Além disto gostaria de agradecer o apoio do meu orientador Prof. Sérgio Assunção Monteiro em me estimular a trabalhar com este tema que combinou mais com as afinidades que eu possuía e o suporte através das troca de e-mails durante estes meses, provou ser bastante eficaz para a realização do trabalho.

Gostaria de agradecer também aos professores e colegas de turma e alguns amigos no decorrer da caminhada da graduação, tanto da Unicarioca – onde cursei parte da graduação, quanto aos colegas e professores que deixei para trás na faculdade a qual vim transferido – a UEZO (Universidade Estadual da Zona Oeste) por motivos que não compensa detalhá-los aqui – mas que também contribuíram no meu conhecimento e, em especial, a alguns amigos que sinto bastante saudades por não ter mais tanto contato, mas que me marcaram de certa forma, como o Rodrigo Veiga, Fernando Silva e Yuri Abreu, meus melhores amigos de lá que tive que perder contato para conseguir concluir a minha graduação na Unicarioca.

Queria também agradecer a paciência das pessoas que me acompanham meus trabalhos audiovisuais nas redes sociais. Neste período de alguns meses que tive que focar em escrever este trabalho, minha parte criativa para escrever roteiros ficou prejudicada, o que me obrigou a simplificar o conteúdo produzido lá para conseguir me dedicar no término deste trabalho.

E, por fim, agradeço a você, leitor que irá consumir este conteúdo. O futuro é incerto e não tenho como afirmar se permanecerei nesta área pelo resto da minha vida, mas espero que este trabalho seja útil e que seja uma contribuição válida na área da computação, servindo de base para seus estudos.

Resumo

O cálculo lambda é um assunto de extrema relevância na área da teoria da computação. Os primeiros estudos que levaram à invenção do calculo lambda pelo matemático Alonzo Church datam-se em 1936. Embora seja uma linguagem de mais de 80 anos e baseada apenas em funções, o que a torna a menor linguagem de programação universal, é uma ferramenta rica e flexível, servindo de alicerce para as linguagens de programação funcionais que seriam criadas na segunda metade do século XX, sem contar com a sua presença em outras linguagens não-funcionais amplamente conhecidas no desenvolvimento de aplicativos comerciais, como o Java 8 e a linguagem Kotlin, voltada para aplicativos móveis. Neste trabalho será demonstrado as vantagens da utilização das funções lambda nestas duas linguagens comerciais citadas através de exemplos de trechos de código, provando que o uso de funções lambda torna o algoritmo menos verboso e mais conciso, permitindo que digite mais instruções com menos linhas de código.

Palavras-chave: Cálculo lambda; Java; Kotlin; Desenvolvimento; Programação

Abstract

Lambda Calculus is a highly relevant topic in the Theory of Computation branch. Initial studies that led to the invention of lambda calculus by a mathematician called Alonzo Church date back to 1936. Although it's a 80-year-old language based only on functions, the main reason of being known as the shortest universal programming language, it's also a rich and flexible tool, being a bedrock of functional programming languages which will be created in the second half of 20th century, not to mention featuring in others non-functional languages widely known in commercial software development, such as Java 8 and Kotlin language, the last one being focused on mobile apps. In this paper, some advantages of using lambda function on Java 8 and Kotlin will be demonstrated by showing some examples of pieces of codes, proving that by using lambda function makes a algorithm less verbose and more concise, allowing to type more instructions with less lines of code.

Keywords: Lambda Calculus; Java; Kotlin; Development; Programming

Lista de Figuras

Figura 1: Alonzo Church, inventor do cálculo lambda.....	18
Figura 2: Imagem de Alonzo Church e Alan Turing.....	20
Figura 3: John McCarthy, criador da linguagem LISP.....	22
Figura 4: Representação dos átomos em LISP.....	23
Figura 5: Tabela presente no artigo de Lindin, 1965 que mostra uma correspondência entre o Algol 60 e cálculo lambda.....	25
Figura 6: Aparelho de dispositivo pessoal portátil StarSeven.....	27
Figura 7: Este é Duke, o mascote oficial da linguagem Java.....	27
Figura 8: Código-fonte em Kotlin mostrando o erro de compilação na atribuição de valor nulo.....	32
Figura 9: Os paradigmas da linguagem de programação.....	34
Figura 10: Esquema de um computador simplificado.....	35
Figura 11: Representação das listas em LISP através de "pares pontilhados"	45
Figura 12: Código-fonte da classe Pessoa.....	48
Figura 13: Algoritmo que conta ocorrência de palavras num arquivo texto com Expressão lambda.....	65
Figura 14: Imagem do Console da IDE Netbeans ao executar o algoritmo da Figura 12.....	68
Figura 15: Algoritmo equivalente ao da figura 12, sem cálculo lambda.....	70
Figura 16: Continuação do algoritmo da figura 15.....	71
Figura 17: Código-fonte de uma interface gráfica sem expressão lambda.....	73
Figura 18: Janela do programa contador.....	74
Figura 19: Código-fonte de uma interface gráfica com expressão lambda.....	75
Figura 20: Algoritmo que compara cálculos em vetores utilizando fluxo sequencial e paralelo.....	78
Figura 21: Desempenho final do algoritmo, comparando fluxo sequencial com fluxo paralelo.....	79
Figura 22: Desempenho do algoritmo da figura 14 com 100 mil elementos do tipo long.....	80
Figura 23: Função contadorDeErros em Kotlin.....	85

Figura 24: Erro de compilação ao tentar incrementar a variável myVar.....	86
Figura 25: Funcionamento do método flatMap() em Kotlin.....	90
Figura 26: Utilização do método filter e depois o map.....	92
Figura 27: Utilização do método map antes do filter.....	93
Figura 28: Ramificação após cálculo recursivo da Sequência de Fibonacci.....	100
Figura 29: Algoritmo que calcula o tempo de processamento dos métodos da sequência de Fibonacci.....	101
Figura 30: Resultado do processamento do algoritmo da figura 29.....	102

Lista de Tabelas

Tabela 1: Versões do Java.....	28
Tabela 2: As seis interfaces básicas citadas por Deitel (2016).....	29
Tabela 3: As linguagens mais utilizadas em 2018.....	36
Tabela 4: Tabela com algumas operações intermediárias.....	54
Tabela 5: Tabela com algumas operações terminais.....	55
Tabela 6: Alguns métodos presentes na classe ArrayList.....	57
Tabela 7: Comparação de resultados obtidos por Deitel e pelo autor do trabalho...103	103
Tabela 8: Tempo relativo após invocar a função Fibonacci para vários valores.....104	104

SUMÁRIO

CAPÍTULO 1 – INTRODUÇÃO.....	11
CAPÍTULO 2 – ANÁLISE CONTEXTUAL DO CÁLCULO LAMBDA.....	15
2.1 Histórico do cálculo lambda.....	16
2.2 Cálculo lambda e o desenvolvimento das linguagens funcionais.....	21
2.3. Sua aplicação em linguagens comerciais de sucesso.....	26
2.3.1 Java.....	26
2.3.2. Kotlin.....	31
CAPÍTULO 3 – DESCRIÇÃO DA PROPOSTA DO TRABALHO.....	34
3.1. Apresentação das funções lambdas para somar e multiplicar.....	38
3.2. Condicionais em cálculo lambda.....	42
3.3. Exemplo de códigos do cálculo lambda.....	43
CAPÍTULO 4 – ESTUDO DE CASO: COMPARAÇÃO E VANTAGENS DO USO DE CÁLCULO LAMBDA NAS DUAS LINGUAGENS.....	47
4.1. Cálculo Lambda em Java8.....	49
4.1.1 Arrays e ArrayLists.....	55
4.1.2 Manipulação de arquivos textos.....	64
4.1.3. Componentes gráficos.....	72
4.1.4. Aplicações concorrentes.....	76
4.2. Cálculo lambda em Kotlin.....	82
4.2.1 Manipulação de Coleções em Kotlin.....	87
4.2.2 Sequências em Kotlin.....	91
4.2.3. Funções de Alta Ordem e Interface Gráfica em Kotlin.....	93
4.2.4: Currying e Memoization: Outros recursos da programação funcional em Kotlin.....	96
4.3. Considerações finais sobre o capítulo 4.....	102
CAPÍTULO 5 – CONCLUSÃO.....	105
REFERÊNCIAS.....	107

CAPITULO 1 – INTRODUÇÃO

O objetivo do trabalho de conclusão de curso (TCC) Aplicações da Linguagem Lambda em Java8 e Kotlin é servir de referência como um estudo introdutório de um outro paradigma de programação que os desenvolvedores não estão acostumados a mexer, acessível, de forma simples, em língua portuguesa. Durante os quatro meses de pesquisa para a realização deste trabalho, percebi que há poucos conteúdos sobre o cálculo lambda disponível em nosso idioma e era um assunto pouco explorado. Embora seja um assunto predominante na computação desde os anos 30 que serviu de pilar para a criação das linguagens de programação funcional na segunda metade do século XX, a maioria dos artigos sobre o assunto estava disponível apenas em inglês, o que se torna inacessível para alguns estudantes de computação que não dominam o idioma. Inclusive, algumas terminologias utilizadas neste trabalho foram traduzidas, na medida do possível. Outras, por não ter achado correlatos em nosso idioma, foi utilizada no idioma original. Além da barreira do idioma, como já citado, é uma forma de apresentar um conteúdo introdutório sobre as possibilidades que a programação funcional permite nos *softwares* a serem criados, como o uso de paralelismo e *multithreading* em cálculos pesados com milhões de entrada, possibilitando uma melhora na performance do código-fonte.

Os aspirantes a programadores foram acostumados ao paradigma imperativo desde o início de seus estudos no curso de graduação em Ciência da Computação. Neste, o programador tem algumas preocupações em construir os seus algoritmos: que variáveis utilizar e como proceder passo-a-passo são algumas destas, uma vez que o conceito é baseado em mudanças de estados na computação. Este paradigma pode acarretar em alguns erros, como variáveis não inicializadas corretamente ou problemas lógicos que atrapalham o desenvolvimento do programa.

No paradigma funcional, no entanto, oferece uma maneira mais concisa e prática de solucionar determinados problemas na computação, uma vez que o foco está em como esta tarefa deverá ser realizada, resultando num código-fonte mais enxuto e mais fácil de ser compreendido. Neste, de acordo com Verma (2016), as funções são consideradas “cidadãos de primeira classe” (*first-class citizens*). Ou seja, através de funções é possível construir os programas. Segundo o autor citado, quando a linguagem permite funções de primeira classe, ela é capaz de tratar

funções como valores podendo atribuí-las a variáveis, passá-las como argumentos de outras funções e, inclusive retornar a função de uma subrotina.

A escolha das linguagens base para o trabalho foram determinadas através de sua relevância no mundo do desenvolvimento. De acordo com a pesquisa feita pela TIOPE em 2018, quase 18% de todo o código escrito é na linguagem Java e, a partir da versão 8, ganhou suporte a programação funcional. E, Kotlin, apesar de ser uma linguagem cuja primeira versão data-se de 2016, ela já está entre as 50 linguagens mais usadas, segundo a pesquisa e, além disso, ela foi criada com objetivo de ser utilizado nos ambientes onde Java predomina, seja na programação backend ou na programação para dispositivos móveis. Além disto, em sua natureza, Kotlin é uma linguagem que também suporta funções de alta-ordem (*high order functions*) podendo utilizada alguns dos recursos da programação funcional.

Mas, se é uma linguagem tão antiga assim, porque explorar este assunto nos tempos atuais? Machado (2013) lista alguns motivos:

- Cálculo lambda ainda é influente nos dias atuais, uma vez que serviu de inspiração para as linguagens funcionais que surgiram em seguida, tais como Lisp e Haskell
- Ela é a base para o estudo formal das linguagens de programação e sistemas de tipo (mesmo as que não seguem o paradigma funcional)
- Ela possui uma correlação com a lógica matemática, também servindo de base de assistente de prova como Coq e Agda. Segundo a página oficial do Coq, ela fornece uma linguagem formal para escrever definições matemáticas, executar algoritmos e teoremas com o objetivo de utilizar a computação para provar os teoremas escritos pelo usuário.
- Cálculo lambda é um cálculo minimalista e elegante. Tanto que, em sua definição, é considerada a “menor linguagem de programação universal do mundo” (Rojas, 1997).
- Permite que saiba utilizar de forma eficaz as novas linguagens de programação.

O cálculo lambda, por ser bastante enxuta, consiste em apenas regras simples de transformação (substituição de variáveis) e um esquema simples de definição de função e ela é universal no sentido de que qualquer função computável pode ser expressada e validada utilizando este formalismo. Neste ponto, é equivalente às máquinas de Turing, porém é uma abordagem mais voltada para o lado de *software*, em vez de *hardware*, uma vez que não se preocupa com a máquina que irá implementá-lo. (Rojas, 1997)

A gramática do cálculo lambda também é bastante simples. Tudo consiste em “expressões”. Rojas (1997) utiliza “nomes” para designar variáveis, que é basicamente um identificador e pode ser representado por qualquer letra do alfabeto. Abaixo está representado a gramática do cálculo lambda:

```

<expressão> := <nome> | <função> | <aplicação>
<função> := λ <nome>. <expressão>
<aplicação> := <expressão><expressão>
```

Para facilitar a leitura, uma expressão E pode estar entre parênteses e uma cadeira de expressões é associa-se a expressão à esquerda. Em outras palavras, a cadeia de expressões abaixo:

$$E_1 E_2 E_3 \dots E_n$$

É validada da maneira abaixo:

$$(\dots((E_1 E_2) E_3) \dots E_n)$$

O trabalho está estruturado em cinco capítulos. Este é um capítulo introdutório onde é explicado a motivação do trabalho, por que pesquisar sobre cálculo lambda e a apresentação de sua gramática. O *Capítulo 2 – Análise Contextual do Cálculo Lambda* introduz ao leitor o formato das expressões lambda e o conceito de beta-redução, que é a base para a transformação das expressões lambda em um valor final, além de contar a história do cálculo lambda, dividida em três grandes períodos: o primeiro, relacionado a história da Lógica Matemática, mostra algumas descobertas de matemáticos que levaram Alonzo Church inventar o cálculo lambda, em 1928. O segundo período compreende-se aos anos 30 e 40, quando Alonzo

Church contou com Alan Turing para resolver o problema de decisão - Entscheidungsproblem, em 1936, além de formularem a Tese de Church-Turing a qual os conceitos de cálculo lambda e máquina de Turing são equivalentes e o terceiro período, que engloba os anos 50 e 60, o qual os cientistas da computação aproveitaram os avanços feito pelo estudo do Church para a criação das primeiras linguagens funcionais. No final do capítulo, uma introdução sobre a sintaxe das expressões lambdas nas linguagens Java e Kotlin.

O *Capítulo 3 – Descrição da Proposta do Trabalho*, revisa os conceitos dos paradigmas de programação: imperativos e declarativos e apresenta os numerais de Church que servem para fazer operações matemáticas em cálculo lambda – tais como soma e multiplicação, além de operações booleanas (AND, OR e NOT) e condicionais (IF-THEN-ELSE) e mostra alguns exemplos de aplicação destas operações na linguagem LISP, citada no Capítulo 2.

O *Capítulo 4 – Estudo de Caso: Comparaçāo e Vantagens do Uso de Cálculo Lambda* nas duas linguagens é a linguagem que ensina o leitor algumas ferramentas que permite a construção de algoritmos em cálculo lambda. Os exemplos abordados mostram duas versões de um mesmo código-fonte: uma utilizando o paradigma funcional e o outro utilizando o paradigma declarativo. Este recurso facilita o processamento de Arrays, Listas, Strings, além de simplificar a criação de elementos gráficos e Threads, além de permitir um considerável ganho de performance em aplicações concorrentes ao aplicar paralelismo no código-fonte. No subtópico 4.2 – Cálculo lambda em Kotlin a abordagem muda, mostrando as principais diferenças em relação ao Java8 e alguns recursos que apenas esta linguagem possui, como *Currying* e *Memoization*, que alivia o processamento de funções recursivas, como Sequências de Fibonacci.

E, por fim, o *Capítulo 5 – Conclusão*, resume o resultado da pesquisa realizada e salienta a relevância do trabalho, além de sugestões para futuros alunos que desejam prosseguir com a pesquisa.

CAPÍTULO 2 – ANÁLISE CONTEXTUAL DO CÁLCULO LAMBDA

Cálculo lambda é uma coleção de alguns sistemas formais inventados nos anos 30 por Alonzo Church, com o objetivo de descrever, da forma mais básica possível, maneiras de como funções ou operadores podem se combinar para gerar outros operadores (HINDLEY e SELDIN, 2008), através da abstração, tornando a sintaxe do cálculo lambda (também referenciada como cálculo-λ ou apenas “λ”) elegante e bastante rica na representação da lógica e na matemática, principalmente quando se trata de funções compostas.

Com o intuito de exemplificar a notação λ (lê-se “lambda”), utilizaremos uma expressão matemática simples, como “ $2x$ ”. Uma função f em função de x seria escrita como:

$$f(x) = 2x$$

A notação de Church é uma maneira sistemática de construir, para cada expressão envolvendo “ x ”, uma função correspondente a variável anterior, utilizando a letra grega lambda ‘λ’ como símbolo de auxílio (HINDLEY e SELDIN, 2008). Transformando a equação acima em notação λ, o resultado seria:

$$f = \lambda x. 2x$$

Nota-se que a letra lambda é apenas um recurso matemático que serve para vincular-se a variável “ x ”, esperando um argumento a ser atribuído para gerar um resultado final a equação. “Escreveremos “ Ma ” para demonstrar a aplicação de uma função M ao argumento a ” (ALAMA e KORBMARCHER, 2018). Supondo a variável “ a ” um valor numérico qualquer (digamos o número 3), basta substituirmos o valor na variável da função para gerar o resultado final:

$$\begin{aligned} & (M)a \\ & (\lambda x[2x])3 = 2 \times 3 = 6 \end{aligned}$$

Este princípio de substituição acima é o conceito básico que sustenta o cálculo lambda e ele se chama **β -redução**, o qual pode ser definido por:

Definição: A β -redução é definida por $(\lambda x.e_1)e_2 \Rightarrow e_1[e_2/x]$ na qual a notação $e_1[e_2/x]$ indica o resultado da substituição de e_2 em todas as ocorrências das variável x em e_1

Esta equação pode ser confusa a primeiro momento mas é importante destacar que o uso desta notação é destinada principalmente para funções compostas. Não apenas para números. O mais importante é que ela é sistemática, ideal para implementação em linguagens de programação. (HINDLEY e SELDIN, 2008).

2.1 Histórico do cálculo lambda

A história de cálculo lambda e da lógica combinatória estão correlacionadas e, de acordo com Cardone e Hindley (2006), podem ser divididas em três grandes períodos: até os anos 30 foi um período com bastante estudo nesta área; o segundo período, até mais ou menos o final dos anos 60, foi uma época mais calma e o último período, quando houve um boom incentivado pela correlação entre os estudos da teoria de funções compostas com as linguagens de programação funcionais.

As primeiras representações de funções abstratas e de substituição foram encontradas no livro de Giuseppe Peano, em 1889. O matemático descrevia que para $\varphi = \alpha[x]$, resultaria na equação $\varphi x' = \alpha[x]x'$, onde subentende-se que este x' seria substituído por x na em α .

Outros matemáticos como o Gottlob Frege e Bertrand Russel (1903) também fizeram notações de funções abstratas, mas nenhum deles realmente a descreveu

formalmente até o nascimento da Lógica Combinatória nos anos 20, inventado pelo matemático Moses Schönfinkel.

Ele inventou o conceito de “combinadores” com o propósito de eliminar as variáveis ligadas a funções. Para explicar o conceito de “variáveis ligadas”, precisamos retornar a Lógica de Predicados para melhor compreensão:

Definição: “Seja x uma variável e E uma fórmula, se x ocorre em E dentro do escopo de um quantificador, $\forall x$ ou $\exists x$, então x é uma variável ligada; caso contrário, x é uma variável livre”

Desta forma, se temos uma função $f(x)=x+y$, nota-se a presença de duas variáveis “ x ” e “ y ”. A variável “ x ” seria uma variável ligada porque há uma ocorrência de x dentro do escopo $f(x)$ e a variável “ y ” seria a variável livre da equação acima.

O conceito de combinadores apareceu novamente na tese de doutorado do Jon Von Neumann, em 1925, de forma modificada a do Schönfinkel, que se resultou na teoria dos conjuntos de Von Neumann-Bernays-Gödel (NBG)

Com o passar do tempo, Schönfinkel descobriu que poderia resolver funções com mais de uma variável, tal como $F(x, y)$, por exemplo, através de resoluções de funções simples de uma só variável, de forma consecutiva, na qual o resultado de uma função seria um parâmetro de entrada para resolver a próxima função (CARDONE e HINDLEY, 2006). Esta técnica é conhecida atualmente como **currying**, nome oriundo do Haskell Curry, que utilizou as notações dos trabalhos do Schönfinkel. Eis um exemplo abaixo deste recurso:

Supondo que há uma função $f(a, b) = a - b$ cujo domínio esteja representado por $f: \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$, poderíamos escrever uma outra função f' equivalente a $f(x, y)$ desta forma $f(x, y) = f'(x)(y) = a - b$ cujo domínio $f': \mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z}$. Seja $x = 1$ e $g = f'(x)$, a função gerada seria $g(b) = 1 - b$, concluindo que a função g foi abstraída da função f , demonstrando o efeito “**currying**”.

O início, de fato, do cálculo lambda foi por volta de 1928 através do matemático americano Alonzo Church¹, como visto na figura 1 abaixo. Nesta época, ele começou a desenvolver uma linguagem formal com o propósito de estabelecer uma base para a lógica de forma mais natural do que as teorias de conjunto de Russell e Zermelo, sem utilizar variáveis livres (CARDONE e HINDLEY, 2006). Ele utilizou, no entanto, funções em vez de conjuntos, utilizando a abstração no formato $\lambda x.M$, sendo M uma função qualquer.



Figura 1: Alonzo Church, inventor do cálculo lambda

Não se sabe ao certo a origem do símbolo lambda. Em alguns artigos de Church, ele cita que utilizou a notação de abstração " x " estabelecida por Whitehead e Russell e substituiu o acento circunflexo pelo lámbda " λ ", com o intuito de facilitar a impressão. Porém, antes da morte do matemático, ele confessou que a escolha foi acidental. Desta forma, ele foi o primeiro matemático a formalizar as regras de notação de abstração de funções.

Em 1933, Church representou os números positivos através dos Numerais de Church, o qual se define:

¹ A Figura 1 do Alonzo Church está disponível em
https://sites.google.com/site/pioneiroscomputacao/_/rsrc/1494289408423/p-ate1949/p15/Alonzo.jpg

Definição: Para cada $n \in \mathbb{N}$, o numeral de Church para n é um termo que devemos chamar \bar{n} que seria equivalente, em λ , por:

$$\bar{n} \equiv \lambda xy.x^n y$$

Seguindo a definição, o número 0 (zero) seria representado por $\bar{0} \equiv \lambda xy.x^0 y = \lambda xy.y$. Os números sucessores nos numerais lambda é definido através desta função sucessora abaixo:

$$\text{Sucessor} \equiv \lambda zxy.x(zxy)$$

Com a fórmula acima, podemos descobrir quaisquer números naturais positivos. Para achar o número 1 (um), por exemplo, teríamos que calcular o sucessor de zero, que será denominado de $S(0)$, bastando apenas aplicar a fórmula:

$$S(0) \equiv \lambda zxy.x(zxy)(\lambda xy.y) \quad (1)$$

$$S(0) \equiv \lambda xy.x((\lambda xy.y)xy) \quad (2)$$

$$S(0) \equiv \lambda xy.x((\lambda y.y)y) \quad (3)$$

$$S(0) \equiv \lambda xy.xy \quad (4)$$

Foram aplicados sucessivas operações de substituição. Na linha (1), colocamos o valor zero ($\lambda xy.y$) como argumento na função sucessora e a inserimos em todas as ocorrências da variável z .

Logo em seguida, na linha (2), aplicamos a substituição na expressão lambda $\lambda xy.y(xy)$, dentro dos parênteses, em todas as instâncias de x . Como não há instâncias de x na função $f(x,y)=y$, reduziu-se a expressão lambda $\lambda y.y(y)$ na linha (3).

Mais uma operação de substituição na função $f(y)=y$ realizada, chegamos ao resultado final descrito na linha (4). Valor o qual também se concluiria, caso aplicássemos a expressão genérica do numeral de Church na descrição acima:

$$\bar{n} \equiv \lambda xy . x^n y \text{ (com } n = 1\text{)}$$

$$\bar{n} \equiv \lambda xy . x^1 y$$

$$\bar{n} \equiv \lambda xy . xy$$

De forma geral, um valor n no numeral de Church teria este formato:

$$n \equiv \lambda xy . x \underbrace{\dots (xy) \dots}_{x \text{ repetidos } n \text{ vezes}}$$

Alonzo Church contou com ajuda de dois brilhantes estudantes – Stephen Kleene e Barkley Rosser – para realizar descobertas importantes que dariam base para o cálculo-λ puro. Embora tenham descoberto algumas inconsistências ao longo de seus estudos – o que serviu de base para a criação do **paradoxo de Curry**, em 1942, o cálculo-λ puro se tornou rico ao ponto de auxiliar na solução do **Entscheidungsproblem** (“problema de decisão”, em alemão) em 1936. Este problema proposto por David Hilbert em 1928, consiste em responder se haveria algum algoritmo que, dada uma expressão lógica, ela seria válida ou não.

Para resolvê-lo, houve contribuições também de um matemático inglês chamado Alan Turing, que fez doutorado em Princeton cujo orientador seria o próprio Alonzo Church.



Figura 2: Imagem de Alonzo Church e Alan Turing

Juntos, os dois matemáticos² - os quais podem ser vistos na figura 2 logo acima - conseguiram resolver o *Entscheidungsproblem*. De acordo com Araújo (2014):

Tanto Church quanto Turing demonstraram que, em seus respectivos modelos de computação, o Entscheidungsproblem é indecidível: não existe uma máquina de Turing ou uma função no cálculo lambda capaz de determinar se uma proposição arbitrária em lógica de primeira ordem é verdadeira.

Logo em seguida, ainda no mesmo ano, Turing provou que os conceitos de cálculo lambda e da máquina de Turing são equivalentes, ou seja, capazes de computar a mesma classe de funções (ARAÚJO, 2014), surgindo assim a **Tese de Church-Turing**.

Na segunda metade dos anos 30, o interesse entre os matemáticos voltados para a área da lógica no assunto cálculo lambda não foi tão intenso.

Contudo, Church expôs seus estudos lambda num livro lançado em 1941 [Church, 1941] que permitiu avanços na parte técnica que permitiu construir uma base sólida para sua futura expansão que estava por vir (CARDONE e HINDLEY, 2006).

2.2 Cálculo lambda e o desenvolvimento das linguagens funcionais

Como dito no subtítulo anterior, a história do cálculo lambda foi subdividida em três grandes períodos para a melhor compreensão. Até aqui foi relatado os primeiros períodos de grandes avanços por parte de diversos matemáticos, desde o final do século XIX até por volta de 1936 na resolução do *Entscheidungsproblem*. Logo em seguida foi um período de duas décadas de consolidação (nos anos 40 e 50).

Contudo, nos anos 60, outro setor de cientistas se interessaram no estudo de cálculo lambda: eram os cientistas da computação voltados no desenvolvimento da parte teórica e prática das linguagens de programação. Neste tópico, citaremos três

2 A figura 2 do matemático Alan Turing e do Alonzo Church está disponível em
[https://geopoliticus.files.wordpress.com/2012/11/church-and-turing.png?w=460&h=347>](https://geopoliticus.files.wordpress.com/2012/11/church-and-turing.png?w=460&h=347)

deles que criaram linguagens de programação baseados na notação lambda de Church.

O primeiro deles é o John McCarthy³ - a foto do cientista está na figura 3 abaixo, que desenvolveu a linguagem LISP no final dos anos 50, no intuito de aplicá-la em problemas, voltado a inteligência artificial, além de estimular o que seria conhecido futuramente como paradigma funcional. Segundo Sebesta (2006), a única linguagem de alto nível que havia na época – a FORTRAN – não tinha fluxos de controles que John McCarthy necessitava, tais como recursão e expressões condicionais.



Figura 3: John McCarthy, criador da linguagem LISP

A linguagem criada por McCarthy possui apenas duas estruturas de dados, de acordo com o diagrama da figura 4 abaixo, representados por átomos e listas. Átomos é a menor estrutura simbólica da linguagem que serve para construir outras expressões. Eles podem representar números, símbolos, caracteres e strings, como mostra a figura⁴ na próxima página

3 A figura 3 do John McCarthy está disponível em <https://static.independent.co.uk/s3fs-public-thumbnails/image/2011/10/31/20/48-John-McCarthy-AP.jpg?w968>

4 Figura 4: Fonte – Elaborada pelo autor (Adaptado de Schubert (2008))

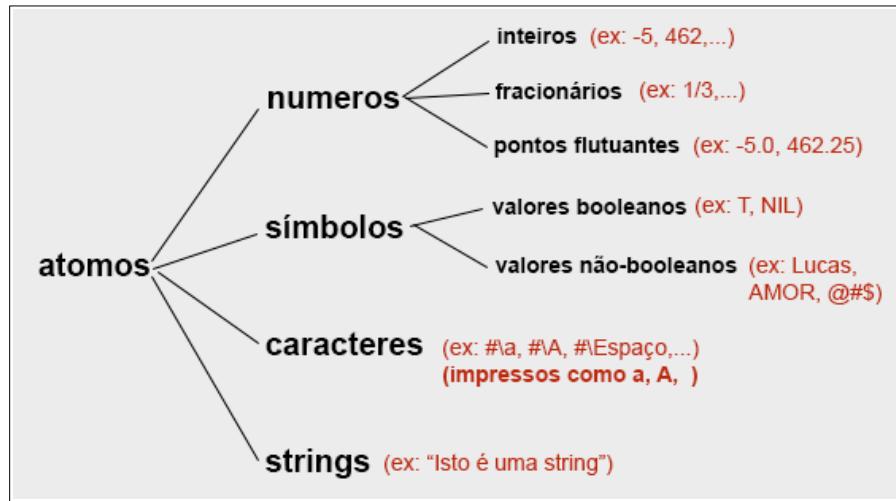


Figura 4: Representação dos átomos em LISP

As listas são estruturas de dados cujos átomos que as compõem estão representadas entre parênteses.

Exemplos de listas:

- 1 - (A B C D)
- 2 - (A (B C) D (E (F G)))
- 3- NIL (o mesmo que ())

Em LISP, todo o programa é visto como uma lista de funções que recebem argumentos e retornam um valor (ou outra função) e não há um identificador (como, por exemplo, a função “main()” na linguagem C) que indique o ponto de partida da execução de um programa. Toda computação desta linguagem é composta por validação de expressões (Schubert, 2008) e possui o formato abaixo:

$$(\text{nome da função } arg_1 \ arg_2 \dots arg_n)$$

A salientar que o nome da função é opcional. Quando quisermos atribuir um nome a função, utilizando o comando **defun**.

Abaixo um exemplo de função “soma” em LISP e a impressão do resultado da função soma – através do comando “format t”:

```
(defun soma(x y)(+x y))
(format t(soma 7 4));imprime 11
```

Caso não deseje nomear a função, basta utilizarmos o comando **lambda**. O exemplo abaixo retrata a criação de uma função anônima que soma 3 ao valor de entrada.

```
((lambda(x)(+x 3)) 2);retorna 5
```

Em LISP, tudo o que estiver escrito após o “;” é um comentário.

Segundo Sebesta (2006), as primeiras linguagens de programação dependiam muito da arquitetura do computador sob o qual o algoritmo escrito naquela linguagem era executado, o que dificultava a comunicação entre usuários, por isso a necessidade da criação de uma linguagem universal. Desta forma, nos anos 60, Peter Lindin se baseou nos termos-λ para criar a linguagem Algol 60, a qual tinham características como recursividade, passagem de parâmetro por valor ou por nome em subprogramas, a introdução de estrutura em bloco e matrizes dinâmicas de pilha.

Apesar de ter sido a primeira linguagem que utilizou o formalismo BNF (Backus-Naur Form) para descrever formalmente a sua sintaxe, sem contar com o fato de ser independente de máquina, não foi uma linguagem muito aceita. De acordo com Sebesta (2006), alguns de seus recursos eram muito flexíveis, os tornando difíceis de ser implementados e, embora atualmente, utilizar a BNF para descrever a sintaxe de uma linguagem seja elegante, naquela época era considerada estranha e complicada.

No artigo de Lindin, 1965, ele fez uma correlação entre notação lambda de Alonzo Church com a linguagem Algol 60, como mostra na figura 5⁵ (em inglês) a seguir. Em seu trabalho, ele definiu a notação de Church como um modelo de linguagem

⁵ A figura encontra-se na página 94 do artigo “A Correspondence Between ALGOL 60 and Church’s Lambda-Notation: Part 1” disponível em <https://fi.ort.edu.uy/innovaportal/file/20124/1/22-landin_correspondence-between-algol-60-and-churchs-lambda-notation.pdf>

baseado no cálculo lambda chamada, em inglês, de *imperative applicative expressions* (IAEs).

TABLE 1	
<i>ALGOL 60</i>	<i>IAEs</i>
Identifiers, operator symbols, also some special words and configurations	Identifiers
Local identifiers	Variables bound in a λ -expression occurring as operator
Formal parameters	Variables bound in a λ -expression occurring as operand
Function designator, subscripted variable, and procedure statement	Operator/operand combination
Procedure	λ -expression
Actual parameter called by name	$\lambda ()$ -expression
Occurrence of a formal called by name	Application to null operand list
Value part of a procedure declaration	Auxiliary definition qualifying the procedure body, redefining some of the formals
Specification part	Auxiliary definition qualifying the procedure body, redefining some of the formals
Block	Combination whose operator is the block-body, and whose operand denotes the (possibly concocted) initial values of the locals
Statement	Expression denoting a non-adic function, changing the environment by side-effects
Compound statement	Functional product of non-adic functions
Label	Identifier defined by a non-adic program point
Labeled segment of program	Program point whose body, denotes a non-adic function
go to —statement	Last (i.e. outer) term of a functional product
Switch	Vector of program closures
Conditional expression or statement	Selection of an item from a listing

Figura 5: Tabela presente no artigo de Lindin, 1965 que mostra uma correspondência entre o Algol 60 e cálculo lambda

O terceiro cientista é o Corrado Böhm. Em 1964, ele percebeu que poderia utilizar expressões lambda como base de uma linguagem bastante flexível para máquinas, programas e algoritmos. Foi o ponto de partida para a linguagem CuCh (Hindley, 2006). Esta linguagem foi criada como uma alternativa as linguagens imperativas

que focavam mais na arquitetura Von Neumann (Bono e Salvo, 2001). Além disto, os autores citados mencionam que a linguagem CuCh resolia sistemas de equações recursivas.

2.3. Sua aplicação em linguagens comerciais de sucesso

Como já mencionado, o estudo de cálculo lambda fez com que surgisse na segunda metade do século XX um novo paradigma de programação baseado nas operações de funções e recursividade que é o paradigma funcional. Embora cálculo lambda e paradigma funcional estejam relacionados, outras linguagens não funcionais suportam o recurso de expressões lambda, como as linguagens C++, Javascript, Python, Ruby, C#, Java e Kotlin. Neste trabalho, focaremos em duas linguagens em particular: Java e Kotlin.

2.3.1 Java

De acordo com Deitel (2016), uma das linguagens mais utilizadas no mundo é o Java. Ela está presente em quase 90% dos desktop de mesa e 3 bilhões de dispositivos. E é uma linguagem relativamente recente, com menos de trinta anos no mercado.

Tudo começou em 1991 quando a Sun Microsystems financiou a pesquisa Green, reunindo os melhores engenheiros que a empresa tinha disponível para examinar o que estaria por vir no mercado de eletrônicos e concluiu que a uma das tendências seria a convergência de produtos eletrônicos digitais. Isto resultou na criação de uma nova linguagem de programa que batizaram de “Oak”, porém tiveram que alterar o nome porque já havia uma linguagem de programação batizada com mesmo nome, então James Gosling decidiu renomeá-la de Java. Esta linguagem herdava funções de outras linguagens, como C e C++. De acordo com Deitel (2016), “um objetivo-chave do Java é ser capaz de escrever programas a serem executados em uma grande variedade de sistemas computacionais e dispositivos controlados por computador”.



Figura 6: Aparelho de dispositivo pessoal portátil StarSeven

No ano seguinte, a Sun decidiu inventar um dispositivo de assistente pessoal portátil chamado *7 (StarSeven⁶), que pode ser visualizado na figura 6 acima. Este aparelho foi resultado da pesquisa Green, inclusive com a criação de um mascote que auxiliava o usuário a manusear o aparelho. Seu nome era Duke⁷ – que se tornaria o mascote da linguagem Java. A representação do mascote está na figura 7 abaixo.

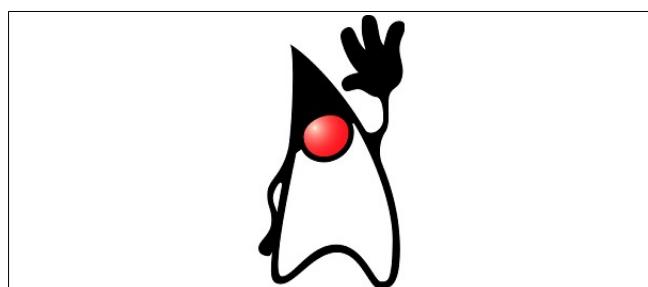


Figura 7: Este é Duke, o mascote oficial da linguagem Java

Este aparelho não teve o sucesso esperado, contudo, em 1993, surgia a World Wide Web e os criadores do projeto viram a oportunidade de utilizar esta recente linguagem neste novo universo através de conteúdos interativos e dinâmicos. Em 1995, a Sun Microsystems em fim lançava a linguagem Java formalmente. Em apenas dois anos, passou a ter 400 mil desenvolvedores Java, tornando-a a 2^a

⁶ Imagem da figura 6 disponível em <<https://tech-insider.org/java/research/1998/05-a.html>>

⁷ Imagem da figura 7 disponível em <http://www.dsc.ufcg.edu.br/~pet/jornal/dezembro2012/materias/historia_da_computacao.html>

linguagem de programação mais utilizada no mundo. Deitel (2016) cita que o Java é utilizado atualmente para aplicativos corporativos de grande, dispositivos de consumo popular (telefones, computadores, televisores, etc.), além de ser uma linguagem-chave para desenvolvimento de aplicativos Android.

Desde a primeira versão, lançada em 1996, Java sofreu diversas mudanças nestes últimos vinte anos. Na tabela⁸ 1 abaixo, pode verificar a data de todas as versões Java, da mais antiga para a mais recente.

Versão	Ano de lançamento
JDK 1.0	1996
JDK 1.1	1997
J2SE 1.2	1998
J2SE 1.3	2000
J2SE 1.4	2002
J2SE 5.0	2004
Java SE 6	2006
Java SE 7	2011
Java SE 8	2014
Java SE 9	2017
Java SE 10	2018

Tabela 1: Versões do Java

Antes do lançamento da versão 7 do Java, a Sun Microsystems foi adquirida pela Oracle – a atual portadora da linguagem Java. Em 2014 foi lançado a 8^a versão do Java. Entre as principais novidades foi a introdução de Expressões Lambda. Até então, a linguagem suportava paradigmas procedural, orientada a objeto e programação genérica (Deitel, 2016). Nestes paradigmas citados, você determina como uma determinada tarefa deve ser realizada.

Já no paradigma funcional não funciona desta maneira: você escreve o que quer alcançar com a tarefa, mas não dirá como alcançar isto. (Deitel, 2016). No capítulo 4 deste trabalho será abordado, com detalhes, a implementação do cálculo lambda em

⁸ A tabela 1 foi elaborada pelo autor, adaptada de <<https://www.codejava.net/java-se/java-se-versions-history>>

Java8, mas, por ora, vamos mostrar as interfaces funcionais presente no pacote `java.util.function` e como são representadas as expressões lambdas em Java 8.

Deitel (2016) descreve seis interfaces funcionais presentes no pacote `java.util.function` e nomeia T e R como tipos genéricos. Tais interfaces estão na tabela 2 logo abaixo⁹:

Interface	Descrição
BinaryOperator<T>	Representa uma operação de cálculo entre dois argumentos T e retorna um valor do mesmo tipo dos operandos
Consumer<T>	Representa uma operação com apenas um argumento de tipo T e não retorna um valor. É usada para gerar uma saída do objeto ou chamar um método do mesmo.
Function<T,R>	Representa uma função que chama um método do tipo T e retorna um resultado do tipo R desse método.
Predicate<T>	Ele testa se o argumento T atende uma condição. Retorna um valor booleano (verdadeiro ou falso)
Supplier<T>	Não recebe argumentos e produz um valor do tipo T. É utilizado para criar um objeto de coleção em que os resultados de uma operação de fluxo são inseridos
UnaryOperator<T>	É uma operação de apenas um operando que produz um resultado de mesmo tipo do operando.

Tabela 2: As seis interfaces básicas citadas por Deitel (2016)

Deitel (2016) define expressões lambdas como um método anônimo e que são utilizadas nos lugares nas quais as interfaces funcionais são esperadas. O autor citado explica a forma genérica de uma expressão lambda como uma lista de parâmetros com um símbolo de seta (\rightarrow) e um corpo onde contém um bloco de instruções. O formato das expressões lambda em Java 8 é exemplificado abaixo:

$(\text{lista de Parâmetros}) \rightarrow \{ \text{bloco de instruções} \}$

Respeitando o formato acima, ao determinarmos uma função soma de dois termos em lambda – utilizarei a mesma denotação de Deitel (2016) ao se referir

⁹ A Tabela 2 foi elaborada pelo autor, adaptada de “Java – Como Programar”, 10^a ed. Pearson (2016) e utilizando a documentação oficial como referência, disponível em <https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html>

“expressões lambas” como apenas “lambda” - pode ser representada de duas formas. Os tipos de parâmetros dos parâmetros em lambda são opcionais, mostrado na expressão 2.

1. (int x, int y) → { return x + y; }
2. (x, y) → { return x + y; }

Quando o bloco de instruções contém apenas uma única instrução, tanto as chaves quanto o comando “return” podem ser omitidas

3. (x, y) → x + y

Quando há apenas um parâmetro na função, os parênteses podem ser omitidos, como visto na expressão 4 abaixo:

4. valor → System.out.printf("%d ", valor)

Para uma função que não há parâmetros a ser passado, representamos com parênteses vazios à esquerda da seta, como mostra a expressão 5:

5. () → System.out.println("Bem-vindo a expressões lambda!")

2.3.2. Kotlin

Kotlin é uma linguagem de programação recente – a sua versão 1.0 foi lançada apenas em 2016. Ela foi criada com o intuito de ser utilizada onde atualmente o Java está presente, como programação *backend* e aplicativos móveis com a mesma performance. Como afirma Jemerov e Isakova (2016), o objetivo principal desta linguagem é fornecer uma alternativa mais concisa, produtiva e segura comparado com o Java em todos os contextos onde esta linguagem é utilizada hoje, além de ser uma linguagem *pragmática*, focada em resolver problemas do mundo real. Não tenta “reinventar a roda” em explorar novidades na área da computação, aproveitando a solução já existente em outras linguagens, o que o seu aprendizado mais fácil. Ela também pode ser executada em navegadores, uma vez que também pode ser compilado em JavaScript. Entre suas características principais:

- É uma linguagem fortemente tipada – assim como o Java – ou seja, os tipos de variáveis são conhecidas em tempo de compilação do programa. Ainda assim se difere a linguagem Java no quesito de nem sempre ser necessário especificar o tipo de cada variável explicitamente no código por causa de um recurso que o compilador possui de inferência de tipos.
- Todas as bibliotecas, o compilador é open source com suporte a pelo menos três IDEs para escrever o código – IntelliJ IDEA Community Edition, Android Studio e Eclipse
- Possui interoperabilidade com a linguagem Java. Em outras palavras é possível fazer chamadas a métodos, estender classes e implementar interfaces em Java dentro de uma classe Kotlin (JEREMOV e ISAKOVA, 2016)

- Ela procura atingir um nível de segurança maior em relação ao Java a custo pequeno. A linguagem é capaz de diferenciar tipos nulos e não-nulos (SAMUEL e BOCUTIU, 2017) bastando apenas colocar um ponto de interrogação (?) logo após o tipo da variável para indicar que a variável aceita valores nulos, como mostra a figura 8 abaixo¹⁰, a qual mostra um erro de compilação ao atribuir o valor nulo quando não inserimos o ponto de interrogação após a tipagem da variável.

```

1 fun main(args: Array<String>){
2     val nome: String = null; //Erro de compilação
3     val nome_2: String? = null //compila
4 }

Compilation completed successfully

teste.kt
    Warning:(2, 8) Variable 'nome' is never used
    Error:(2, 23) Null can not be a value of a non-null type String
    Warning:(3, 8) Variable 'nome_2' is never used

```

Figura 8: Código-fonte em Kotlin mostrando o erro de compilação na atribuição de valor nulo

Kotlin é uma linguagem orientada-a-objetos (POO) com suporte a funções de alta-ordem e lambdas (SAMUEL e BOCUTIU, 2017). Isto lhe permite passar funções como parâmetro de funções, armazená-las em variáveis e uma vez que sempre uma função retorna o mesmo valor dadas as mesmas entradas, não há efeitos colaterais. Além disto, permite um *multithreading* seguro pois, uma vez que estes dados são imutáveis, o processo de sincronização para definir qual *thread* vai acessar a região crítica do programa é desnecessário e, por fim, facilita testar o código, uma vez que, segundo Jemerov e Isakova (2016), as funções podem ser testadas de forma isolada sem a necessidade de escrever linhas de código para construir todo um ambiente de teste.

¹⁰ Imagem elaborada pelo autor. A interface é do site <https://try.kotlinlang.org>

A sintaxe das expressões lambda na linguagem Kotlin está escrita abaixo: Os parâmetros à esquerda da seta (\rightarrow) e à direita temos o corpo da função, onde contém o bloco de código. Toda a função lambda é fechada está entre chaves (“{ } ”)

{ parâmetros \rightarrow corpo da função }

Uma função lambda que soma dois valores inteiros na linguagem Kotlin teria esta sintaxe:

{ x: Int, y: Int \rightarrow x + y }

A linguagem permite que escreva mais de uma linha de comandos dentro das chaves que delimitam a função lambda, como visto no exemplo abaixo:

```
val soma = {x: Int, y: Int ->
    println("Computando a soma de $x e $y ")
    x + y}
```

Na segunda linha do código acima, no `println`, há um cífrão '\$' antes do valor da variável. Aquele símbolo permite a referência de variáveis locais dentro de uma String na linguagem Kotlin.

É possível armazenar a função lambda em uma variável para ser utilizada como se fosse uma função normal:

```
val soma = { x: Int, y: Int  $\rightarrow$  x + y }
    println(soma(3,4)) //imprimirá o valor 7
```

No capítulo 4 será abordado em detalhes as demais aplicações de cálculo lambda na linguagem Kotlin.

CAPÍTULO 3 – DESCRIÇÃO DA PROPOSTA DO TRABALHO

O objetivo do trabalho é dar um aspecto introdutório ao universo da programação funcional nas linguagens Java8 e Kotlin, amplamente utilizadas no mercado de desenvolvimento de software comercial, mostrando através de exemplos que, através da linguagem funcional, é possível obter um código mais legível, eficaz em termos de performance (principalmente ao utilizar o conceito de Stream API em Java 8, no capítulo 4, em permitir o paralelismo que consegue aproveitar os processadores multinúcleos no processamento de cálculos quando há milhões de entrada de dados no programa, significando uma economia de tempo e melhora de desempenho do software) e focando sempre em um código mais conciso – em outras palavras, é a possibilidade de multitarefas com poucas linhas de código no algoritmo, e desta maneira, incentivar o desenvolvimento e otimização de programas em caso da linguagem permitir aproveitar os recursos do paradigma funcional.

Varejão (2004) define paradigmas a “um conjunto de características que servem para categorizar o grupo de linguagens”. A princípio, os paradigmas da linguagem de programação são subdivididos em dois grandes grupos: **paradigma imperativo** e **paradigma declarativo**. O diagrama da figura 9 abaixo¹¹ mostra como são classificados os paradigmas da linguagem de programação:

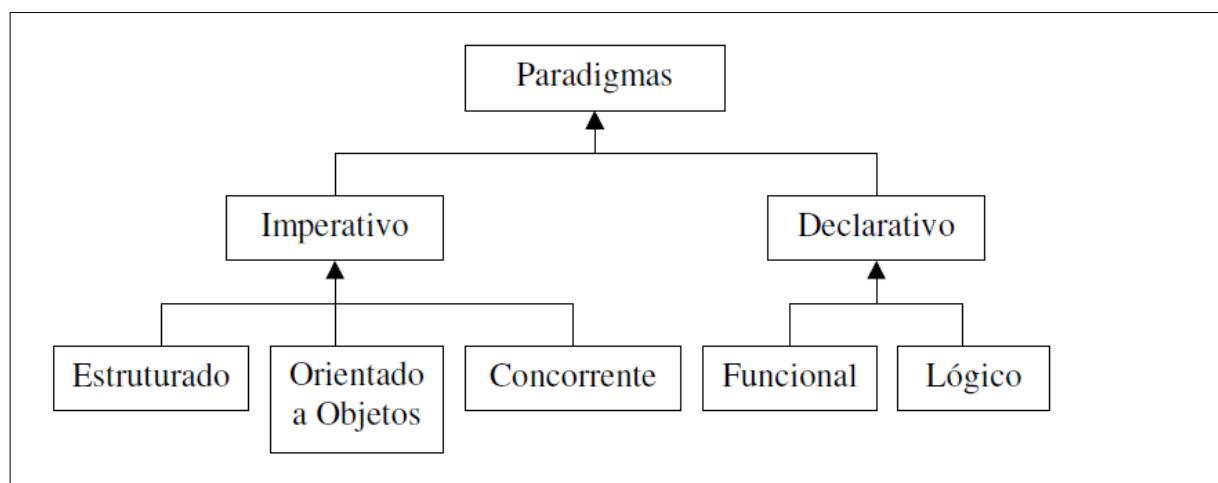


Figura 9: Os paradigmas da linguagem de programação

¹¹ Imagem retirada do livro “Linguagens de Programação Java, C, C++ e outras” (2004) de Flávio Varejão.

O paradigma imperativo é aquele o qual o desenvolvedor já está acostumado a criar seus programas. Este paradigma é baseado no conceito de mudança de estado. Varejão (2004) define estes estados como uma configuração da memória do computador. Além disto, deve ser especificado todas as etapas de como um processamento deve ser realizado. Aqui é introduzido conceitos de variáveis, valor e atribuição.

E é o primeiro paradigma que todo estudante que inicia o curso de Ciência da Computação se depara. Guimarães e Lages (1985) em sua obra “Introdução da Ciência da Computação” explica o conceito de Computador simplificado com o objetivo de abstrair o funcionamento de um computador eletrônico, como demonstra a figura 10 abaixo¹²:

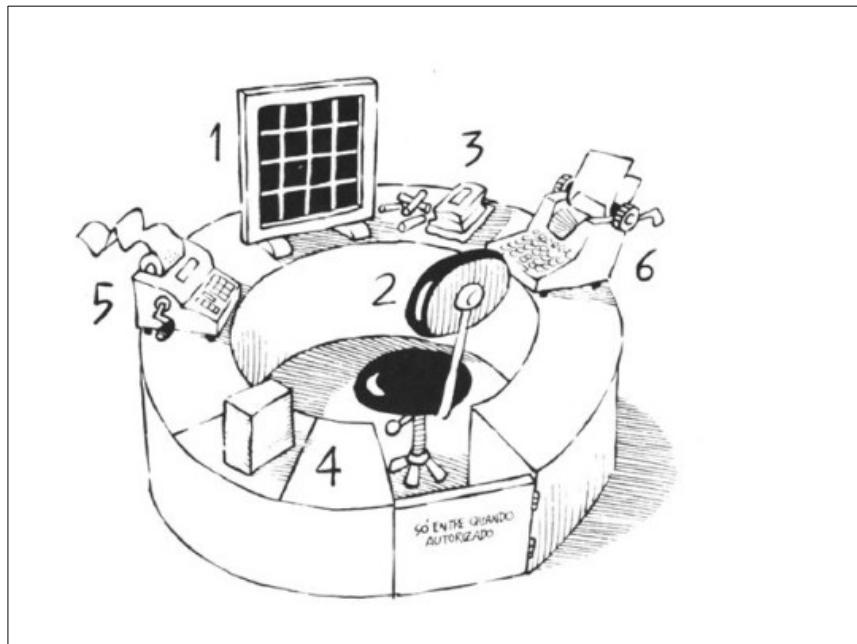


Figura 10: Esquema de um computador simplificado

O ítem número 1 da figura acima é um escaninho subdividido em dezesseis quadrados. O ítem 2 é a cadeira do operador. O ítem 3 é o giz e o apagador para poder manipular o escaninho. O ítem 4 é a bandeja com cartões numerados. O ítem 5 é uma máquina de calcular e o ítem 6 é a máquina de escrever.

O escaninho tem o papel análogo a memória de um computador, onde se escreve dados (no caso acima, números provenientes da bandeja de cartões) e instruções.

¹² Imagem retirada do livro “Introdução à Ciência da Computação” (1985) de Guimarães/Lajes.

O giz e o apagador são responsáveis pela operação de escrita na memória (no caso, o escaninho). A máquina de escrever é a saída do programa enquanto os cartões numerados é a entrada de dados. Os itens 1, 2, 3 e 5 representam o processamento. O conceito descrito por Guimarães e Lages (1985) prepara o aluno para o paradigma imperativo. Este é o paradigma é o mais utilizado entre os desenvolvedores. De acordo com uma pesquisa feita pela TIOBE, apenas o SQL é a única linguagem de programação declarativa entre as dez linguagens de programação mais utilizadas em 2018, considerando o quesito porcentagem do total de quantidade de linhas de código escritas, como mostra a tabela 3 abaixo¹³:

Posição	Linguagem	Porcentagem
1	Java	17.8%
2	C	15.3%
3	C++	7.59%
4	Python	7.15%
5	Visual Basic .NET	5.88%
6	C#	3.48%
7	PHP	2.79%
8	Javascript	2.28%
9	SQL	2.03%
10	Swift	1.5%
41	Kotlin	0.22%

Tabela 3: As linguagens mais utilizadas em 2018

O paradigma imperativo se subdivide-se em paradigmas estruturados, orientados a objetos e concorrente. A linguagem Java – a mais utilizada de acordo com a imagem acima - é uma linguagem que suporta múltiplos paradigmas e, como mencionado no subtópico 2.3.1 – Java, a partir do Java 8 ela passou a suportar o paradigma funcional. Este paradigma é uma subclassificação do **paradigma declarativo**. Neste, o programador não precisa se preocupar com a implementação em si e sim, em como esta tarefa deve ser realizada, de forma abstrata (VAREJÃO, 2004).

13 A tabela está disponível no link <<https://www.tiobe.com/tiobe-index/>>

Entre algumas aplicações da programação funcional introduzida no Java8 e será vista com mais profundidade no Capítulo 4, está uma nova forma de iterar coleções através do método `forEach()`. É uma forma mais compacta e menos propensa a erros, uma vez que dispensa variáveis de controle para iteração de um conjunto de dados que exigem numa estrutura `for` tradicional em outras linguagens de paradigmas imperativos.

Outra novidade desta linguagem é a Stream API que permite manipular todos os elementos de uma coleção na linguagem JAVA através de múltiplas operações sucessivas que podem filtrar ou mapear elementos através de uma função fornecida, facilitando manipulações em Arrays ou Listas de elementos. Achar, por exemplo, uma média ou o maior valor de um array será uma tarefa bem mais prática de ser realizada, em basicamente uma linha de código.

A programação lambda também facilitará manipulação de um conjunto de Strings em Java 8. Entre os exemplos do capítulo 4 terá um algoritmo que conta a ocorrência de palavras num texto lido em um arquivo de texto. Enquanto a contagem destas palavras na programação funcional foi realizada com apenas três operações, no paradigma tradicional fui obrigado a criar um método para contar a quantidade de vezes que uma palavra aparece num texto, outro método para *linkar* a quantidade de vezes que a palavra aparece à palavra através da estrutura de mapeamento `TreeMap<>`. Um processo que exige algumas variáveis intermediárias e que no fim foi gasto, pelo menos, o dobro de número de linhas de código.

Sem contar com a praticidade de atrelar eventos a componentes gráficos e um enorme ganho de desempenho em aplicações concorrentes ao utilizar o conceito de paralelismo. Em um número grande de elementos, a economia de tempo de processamento chega a 70%.

A abordagem utilizada em todo subtópico 4.1 – Cálculo Lambda em Java8 foi mostrar os algoritmos em duas versões: uma utilizando os recursos provenientes do paradigma funcional e o outro algoritmo, utilizando a abordagem tradicional do paradigma imperativo.

A segunda parte do capítulo 4 é dedicada a linguagem Kotlin. Como já mencionado no capítulo 2, é uma linguagem relativamente recente – e mesmo assim, como visto na tabela 3, conseguiu atingir o top 50 das linguagens mais utilizadas – e seu

objetivo é ser uma alternativa a linguagem Java com performance similar, porém com a vantagem de ser uma linguagem menos verbosa e um nível de segurança maior como a verificação se a variável aceita valores nulos. Por atuar em territórios onde a linguagem Java é predominante, seria repetitivo se mantivesse a mesma abordagem comparativa do subtópico anterior, uma vez que alguns recursos de programação funcional em Kotlin são equivalente em Java.

Desta forma, optou-se por destacar algumas diferenças que Kotlin nos fornece como o fato de poder manipular e alterar variáveis dentro de uma função lambda – que não é possível em Java 8 e o recurso de Sequências – que é equivalente a Stream API de Java 8, exceto o paralelismo que é exclusivo da linguagem Java.

Algumas características da programação funcional serão apontadas no subtópico 4.2 – Cálculo Lambda em Kotlin, como funções de alta ordem, que é a possibilidade de passar funções inteiras como parâmetros de outras funções; o recurso de *Currying* que é uma técnica que executa uma função com n variáveis em n sucessivas funções com uma variável apenas como parâmetro e o recurso de *Memoization* onde utiliza a cache da memória para agilizar a computação de funções recursivas, com o intuito de perceber a ineficiência da abordagem recursiva tradicional do algoritmo de sequência de Fibonacci comparado com a abordagem que utiliza *memoization* para realizar o mesmo procedimento.

3.1. Apresentação das funções lambdas para somar e multiplicar

Para explicar como o cálculo lambda funciona, deve-se ressaltar alguns conceitos mencionados no tópico 2.1 – *Histórico de cálculo lambda*, quando foi explicado a definição de variáveis livres e variáveis ligadas para entender como é o processo do cálculo da função lambda. Geralmente, um “programa lambda” é apresentado pelo símbolo lambda, seguido das variáveis. O ponto liga as variáveis ao corpo da função – que no caso seria uma equação. Supondo a equação abaixo:

$$\lambda x.(x+1)$$

Lê-se: “Aquele função lambda de x a qual adiciona x ao valor 1”. Considerando que, os operadores são pré-fixados, o programa lambda acima poderia ser reescrito para:

$$\lambda x.(+x1)$$

Os atributos da função vem do lado de fora da função, como demonstrado abaixo:

$$(\lambda x.(+x 1))5$$

Neste caso, x é a variável ligada a função, a qual o parâmetro 5 será substituído em todas as ocorrências de x e, após a soma, chega-se ao resultado final 6:

$$(+51)=6$$

O processo de substituição acima se chama **β -redução**, definido no inicio do capítulo 2.

No próximo exemplo, repare que a variável y é livre na equação abaixo, uma vez que não ocorre **β -redução** ao inserir apenas o parâmetro 4 a ser substituído pela variável x na função:

$$\begin{aligned} & (\lambda x.(+x y))4 \\ & (+4 y)=4+y \end{aligned}$$

No terceiro exemplo, repare que a ordem dos parâmetros afeta o processamento do cálculo lambda. O valor 4 é substituído a todas as ocorrências de x, apenas em seguida, o valor 5 é substituído em todas as ocorrências da variável y.

$$\begin{aligned} & (\lambda x.(\lambda y.-yx))\ 4\ 5 \\ & (\lambda y.(-y4)5) \\ & (-45) = -1 \end{aligned}$$

Funções com múltiplas variáveis como esta $(\lambda x.(\lambda y.-yx))$ podem ser abreviadas para $(\lambda xy.-yx)$.

No tópico 2.1 – *Histórico de cálculo lambda*, também mostrou uma forma de como os algarismos numéricos estão representados em cálculo lambda através dos Numerais de Church. Estes numerais são representados, de acordo com a definição, através da fórmula geral $\lambda xy.x^n y$ e que aplicando valores a n, temos os algarismos:

$$0 \equiv \lambda xy.y$$

$$\begin{aligned} 1 &\equiv \lambda xy.x(y) \\ 2 &\equiv \lambda xy.x(x(y)) \\ 3 &\equiv \lambda xy.x(x(x(y))) \end{aligned}$$

E assim sucessivamente para todo valor de n positivo.

A representação da soma em cálculo lambda é aplicar a função sucessora $Sucessor \equiv \lambda zxy.x(zxy)$ n vezes ao valor que deseja somar. Em outras palavras, numa hipotética soma $2 + 3$, seria equivalente em cálculo lambda a **2S3**. Duas vezes a aplicação da função sucessora S ao valor 3, como demonstra abaixo:

$$\begin{aligned} 2+3 &\equiv 2S3 \equiv [\lambda zxy.x(zxy)][\lambda zxy.x(zxy)][\lambda xy.x(x(x(y)))] \quad (1) \\ &[\lambda zxy.x(zxy)][\lambda xy.x([\lambda xy.x(x(x(y)))]xy)] \quad (2) \\ &[\lambda zxy.x(zxy)][\lambda xy.x([\lambda y.x(x(x(y)))]y)] \quad (3) \\ &[\lambda zxy.x(zxy)](\lambda xy.x(x(x(x(y)))) \quad (4) \\ &\lambda xy.x((\lambda xy.x(x(x(x(y))))xy) \quad (5) \\ &\lambda xy.x((\lambda y.x(x(x(x(y))))y) \quad (6) \\ &\lambda xy.x(x(x(x(x(y))))) \equiv 5 \quad (7) \end{aligned}$$

O uso de colchetes na linha um serve apenas para separar as duas funções sucessoras de forma mais clara. Das linhas 1 a 4 foi aplicado a função sucessora ao valor 3 (os detalhes de como é aplicado a função sucessora já foi abordado no tópico *2.1 – Histórico de cálculo lambda*), resultando em outra função sucessora, agora ao valor 4, como demonstra a linha 4. Computando a nova função, chegamos ao valor equivalente ao cinco, que é o resultado da soma $2 + 3$.

A multiplicação em cálculo lambda é baseada na aplicação da função da multiplicação $M \equiv \lambda xyz.x(yz)$ aos dois termos que deseja multiplicar. Abaixo, uma demonstração da aplicação da função para o cálculo de 2×3 :

$$\begin{aligned} 2 &\equiv \lambda wy.w(w(y)) \quad (1) \\ 3 &\equiv \lambda py.p(p(p(y))) \quad (2) \\ M(2)(3) &\equiv [\lambda xyz.x(yz)](\lambda wy.w(w(y)))(\lambda py.p(p(p(y)))) \quad (3) \end{aligned}$$

$$[\lambda z.(\lambda wy.w(w(y)))((\lambda py.p(p(p(y))))z)] \quad (4)$$

$$[\lambda z.(\lambda wy.w(w(y)))((\lambda py.p(p(p(y))))z)] \quad (5)$$

$$[\lambda z.(\lambda y.((\lambda y.z(z(z(y)))))(((\lambda y.z(z(z(y)))))((y))))] \quad (6)$$

$$\lambda z.(\lambda y.((\lambda y.z(z(z(y)))))((z(z(z(y)))))) \quad (7)$$

$$\lambda z.(\lambda y.(z(z(z(z(z(y))))))) \quad (8)$$

$$\lambda zy.z(z(z(z(z(y)))))\equiv 6 \quad (9)$$

Abaixo, o passo a passo de todas as operações realizadas acima:

- No passo (1) e (2) definiu-se a função que representa o numeral de Church para os valores 2 e 3. Foi alterado o nome da variável x para w e p apenas para uma melhor leitura. Atente-se apenas ao formato da função que é o mesmo dos exemplos anteriores.
- No passo (3), a função foi montada com os dois parâmetros do lado de fora da função multiplicação M, representada entre colchetes.
- No passo (4), as variáveis x e y da função M foram substituídos pelas funções 2 $\lambda wy.w(w(y))$ e 3 $\lambda py.p(p(p(y)))$.
- No passo (5), a variável p presente na função do número 3 foi substituída pela variável externa z.
- No passo (6), a variável y isolada no final da linha foi parâmetro para substituir na função $\lambda y.z(z(z(y)))$, tornando-se apenas $z(z(z(y)))$.
- No passo (7), foi a vez de substituir y em $\lambda y.z(z(z(y)))$ por $z(z(z(y)))$, resultado em $z(z(z(z(z(y))))))$
- No passo (8), a equação foi reescrita de forma equivalente para eliminar o segundo lambda, resultado o valor final escrito no passo (9).

3.2. Condicionais em cálculo lambda

Assim como em qualquer linguagem de programação, cálculo lambda permite representação de expressões condicionais, com o auxílio de duas funções que definem os valores booleanos verdadeiros (True) e falso (False). A função True (T) é aquela que recebe dois argumentos e retorna o primeiro, como mostra abaixo:

$$T \equiv \lambda xy.x$$

Enquanto a função False (F) é aquela que retorna o segundo argumento, como mostra abaixo:

$$F \equiv \lambda xy.y$$

Através das funções True e False é possível obter as funções das operações lógicas mais conhecidas, tais como AND, OR e NOT:

$$\text{AND} \equiv \lambda xy.xy(\lambda uv.v) \equiv \lambda xy.xy F$$

$$\text{OR} \equiv \lambda xy.x(\lambda uv.u)y \equiv \lambda xy.x T y$$

$$\text{NOT} \equiv \lambda x.x(\lambda uv.v)(\lambda ab.a) \equiv \lambda x.x FT$$

O primeiro exemplo mostra a computação da função AND, recebendo um FALSE e um TRUE como argumentos. O resultado esperado após as reduções é um FALSE, como mostra-se abaixo:

$$\begin{aligned} \text{ANDFT} &\equiv (\lambda xy.xyF)(\lambda xy.y)(\lambda xy.x) \\ &(\lambda y.(\lambda xy.y)yF)(\lambda xy.x) \\ &(\lambda xy.y)(\lambda xy.x)F \\ &(\lambda y.y)F = F \end{aligned}$$

Da mesma forma que, aplicando a função OR com os mesmos parâmetros, o resultado esperado seria um TRUE:

$$\begin{aligned} \text{OR} &\equiv \lambda xy.x(\lambda uv.u)y \equiv \lambda xy.x T y \\ &(\lambda y.(\lambda xy.y)T y)(\lambda xy.x) \\ &(\lambda xy.y)T(\lambda xy.x) \\ &(\lambda y.y)(\lambda xy.x) = \lambda xy.x = T \end{aligned}$$

O terceiro exemplo demonstra a computação da função NOT TRUE, cujo resultado esperado é um FALSE:

$$\begin{aligned}\text{NOT } T &\equiv (\lambda x.x \text{FT})(\lambda xy.x) \\ &(\lambda xy.x) \text{FT} \\ &(\lambda y.F)T = F\end{aligned}$$

Para representar uma condicional no estilo “Se P então A, se não B”, utilizamos a expressão lambda abaixo:

$$\text{COND} \equiv \lambda pab.pab$$

A lógica desta função funciona da seguinte maneira: quando atribuímos TRUE a ‘P’ retorna a variável ‘A’. Caso contrário, ao atribuir FALSE a ‘P’, a expressão condicional retornará a variável B, como mostra o quarto exemplo abaixo:

$$\begin{aligned}\text{COND TRUE } AB &\equiv (\lambda pab.pab)(\lambda xy.x)AB \\ &(\lambda ab.(\lambda xy.x)ab)AB \\ &(\lambda b.(\lambda xy.x)Ab)B \\ &(\lambda xy.x)AB \\ &(\lambda y.A)B = A\end{aligned}$$

3.3. Exemplo de códigos do cálculo lambda

No tópico 2.2 – *Cálculo Lambda e o desenvolvimento das linguagens funcionais*, foi mencionado que os estudos do Alonzo Church interessaram cientistas da computação para desenvolver um novo paradigma baseado em cálculo lambda. Um destes cientistas, John McCarthy fundou a linguagem LISP – uma linguagem de programação voltada a inteligência artificial, baseado em apenas duas estruturas básicas: átomos e listas.

Nota-se a similaridade em algumas funções nesta linguagem como a função condicional ‘if’, com comportamento idêntico ao demonstrado no tópico anterior. A função If também recebe três argumentos e, para a linguagem, a expressão NIL significa Falso – retornando o terceiro parâmetro. Para qualquer outro valor, contudo,

é considerado verdadeiro e, portanto, a expressão retorna o valor do segundo parâmetro, como visto no exemplo abaixo:

```
(if NIL 3 4)
```

```
> 4
```

```
(if 5 3 4)
```

```
> 3
```

A linguagem LISP permite condicionais mais complexas no estilo *if-then-else* através da função COND. Ele é formado por várias listas cujo primeiro elemento é a condição e o elemento restante é a consequência e avalia o primeiro elemento de cada cláusula e caso seja verdadeiro, ele executa a ação.

No código abaixo ele atribui a variável x ao valor 5 através da função setq e, em seguida compara: primeiramente avalia se o valor de x é maior que zero. Caso seja, ele imprimirá a palavra “positivo”. Se não, ele vai para a segunda cláusula. Caso o valor de x seja menor que zero, ele imprimirá a palavra “negativo”. A última lista seria o *else* da condição, o qual apenas imprimirá a palavra “zero”.

```
(setq x 5)
```

```
(cond
```

```
 ((> x 0) (print "positivo"))
```

```
 ((< x 0) (print "negativo"))
```

```
 (print "zero") )
```

LISP possui duas funções que manipulam listas – que são as funções *car* e *cdr*. A função *car* retorna o primeiro elemento da lista, enquanto o *cdr* mostra todos os demais elementos da lista, como visto abaixo:

```
(setq lista '(A B C))
```

```
(car(lista))
```

```
> A
```

```
(cdr(lista))
```

```
> (B C)
```

Há também funções em LISP, tais como **second**, **third** para acessar os demais elementos da lista assim como também pode chamar (car (cdr ...)), (car (cdr (cdr ...))) ou mesmo as confunções **cadr**, **caddr** para executar as mesmas funções (Schubert, 2008), como visto abaixo:

```
(second lista)
> B
(car(cdr lista))
> B
(cadr lista)
> B
(third lista)
> C
(car(cdr(cdr lista)))
> C
(caddr lista)
> C
```

A linguagem trata internamente a lista como “pares pontilhados” (*dotted pairs*). Estas células possuem um ponteiro que aponta para um átomo ou para um “par pontilhado”, como demonstra a figura 11 abaixo¹⁴:

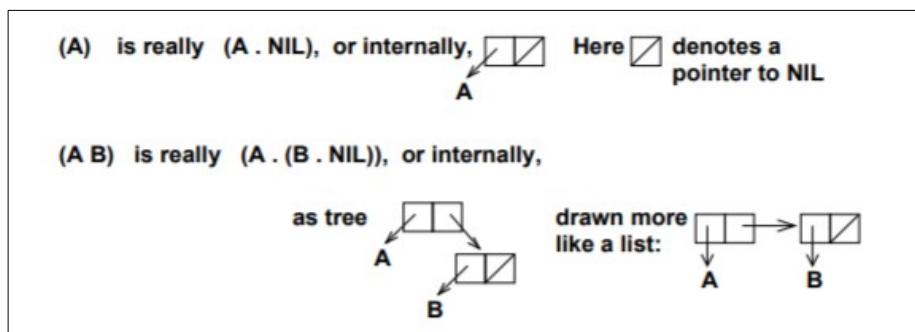


Figura 11: Representação das listas em LISP através de "pares pontilhados"

A função CONS retorna em Lisp um registro com os campos “car” e “cdr” que representam um “par pontilhado”, como mostra o exemplo abaixo:

14 Figura retirada do artigo “A Quick Introduction to Common Lisp” (2008) p.7

```
(setq lista '(A B C))
```

```
(cons 1 2)
```

```
> (1 . 2)
```

```
(cons (cons 4 5) 6)
```

```
> ((4 . 5) . 6)
```

Em cálculo lambda podemos representar as funções CONS, HEAD (equivalente ao CAR em LISP) e TAIL (equivalente ao CDR em LISP) através das seguintes funções lambdas abaixo:

$$\text{CONS} \equiv \lambda ht. (\lambda s. sht)$$

$$\text{HEAD} \equiv \lambda l. lT$$

$$\text{TAIL} \equiv \lambda l. lF$$

As funções T e F referem-se as funções lambdas Verdadeiro e Falso vistas no tópico 3.2 – Condicionais em cálculo lambda.

CAPÍTULO 4 – ESTUDO DE CASO: COMPARAÇÃO E VANTAGENS DO USO DE CÁLCULO LAMBDA NAS DUAS LINGUAGENS

Neste capítulo mostraremos diversas aplicações de expressões lambdas nas linguagens Java e Kotlin, duas linguagens de programação utilizadas no meio comercial. No capítulo 2 foi apresentado uma introdução de expressões lambdas nestas duas linguagens de programação – mostrando a sintaxe e alguns exemplos básicos do uso desta sintaxe. Agora demonstraremos a verdadeira vantagem do uso de cálculo lambda na programação funcional através da comparação de algoritmos, focando-se na concisão do código-fonte. Em outras palavras, o leitor notará que o uso de expressões lambda permite ao programador escrever mais funcionalidades com menos linhas de código através de exemplos mais elaborados ao comparar com a versão do código fonte sem o uso das expressões lambdas, que permite ao programador desempenhar sua função com mais eficácia no desenvolvimento de aplicativos comerciais.

Com o intuito de demonstrar os exemplos a seguir, criarei uma classe Pessoa com os seguintes atributos *Nome* e *Sobrenome*, *idade* e *altura*, além dos seus getters() e setters() e o método *toString()*, que retorna as informações dos atributos do objeto em formato de *String*. E um método *getNomeCompleto()*, o qual retornará o Nome e Sobrenome da pessoa. O código-fonte da classe Pessoa está descrito na página 48 na figura 12¹⁵.

15 Figura 12 foi elaborado pelo autor

```

public class Pessoa {
    String nome;
    String sobrenome;
    int idade;
    float altura;

    public Pessoa() {
    }

    public Pessoa(String nome, String sobrenome, int idade, float altura) {
        this.nome = nome;
        this.sobrenome = sobrenome;
        this.idade = idade;
        this.altura = altura;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getSobrenome() {
        return sobrenome;
    }

    public void setSobrenome(String sobrenome) {
        this.sobrenome = sobrenome;
    }

    public int getIdade() {
        return idade;
    }

    public void setIdade(int idade) {
        this.idade = idade;
    }

    public float getAltura() {
        return altura;
    }

    public void setAltura(float altura) {
        this.altura = altura;
    }

    public void nomeCompleto(){
        System.out.println(getNome() + " " + getSobrenome());
    }

    @Override
    public String toString() {
        return "nome=" + nome + ", sobrenome=" + sobrenome + ", idade=" + idade + ", altura=" + altura;
    }
}

```

Figura 12: Código-fonte da classe Pessoa

O código-fonte da figura 11 na página anterior¹⁶ mostra os `getters()` e `setters()` da classe Pessoa – automaticamente gerado pelo IDE Netbeans. Esta classe contém os atributos nome, sobrenome, idade e altura e uma plataforma sobreescrita `toString()` que retorna as informações do objeto criado em formato de String e o método `nomeCompleto()` que retorna o nome e sobrenome da pessoa.

4.1. Cálculo Lambda em Java8

Como referenciado no capítulo 2, a programação funcional em Java foi introduzido na versão 8 (Java SE 8), em 2014. A vantagem deste paradigma é que, de acordo com Deitel (2016), permite ao programador escrever programas mais rápidos, concisos e menos erros em comparação aos demais paradigmas de programação, uma vez que não especificamos diretamente como realizar uma determinada tarefa e, em vez disto, o que quero alcançar ao digitar uma tarefa.

Ainda segundo Deitel (2016), na programação funcional não é necessário iterar pelos elementos em uma estrutura de looping (como um `for` – que exige uma variável de controle), nem declarar variáveis que podem acarretar em erros de programação, se o programador Java não tiver o cuidado de inicializá-las de forma correta ou então verificar se a condição de parada numa estrutura de repetição esteja correta, permitindo focar apenas o que fazer com os elementos da lista percorrida. A forma tradicional de iteração, o autor denomina como **iteração externa**. Deitel (2016) afirma que, apesar de ser propenso a erros, a maioria dos programadores se sentem confortável com a iteração externa. No código-fonte abaixo descreverá como era feito a iteração externa até o Java8, utilizando o formato `for` tradicional. O algoritmo cria uma lista do tipo Pessoas, cinco objetos da classe Pessoa e os insere nesta lista e, em seguida, o itera, retornando o nome completo de cada um dos objetos criados:

16 Imagem elaborada pelo autor

Algoritmo 1:

```

1. public class IteracaoExplicita {
2.
3.     public static void main(String[] args) {
4.         List<Pessoa> listaPessoas = new ArrayList<>();
5.
6.         listaPessoas.add(new Pessoa("Lucas", "Rocha", 25, (float)1.85));
7.         listaPessoas.add(new Pessoa("Naira", "Rocha", 50, (float)1.70));
8.         listaPessoas.add(new Pessoa("Felipe", "Neves", 60, (float)1.68));
9.         listaPessoas.add(new Pessoa("Sérgio", "Monteiro", 45, (float)1.81));
10.        listaPessoas.add(new Pessoa("Ana", "Souza", 19, (float)1.90));
11.
12.        System.out.println("Lista de pessoas: ");
13.        for (int i = 0; i < listaPessoas.size(); i++)
14.            listaPessoas.get(i).nomeCompleto();
15.    }
16. }
```

A partir do Java5, há uma nova maneira para percorrer os objetos de uma lista ou array a qual Deitel (2016) chama de **instrução for aprimorada** cuja sintaxe é:

```

for (parâmetro : nomeDoArray)
    instrução
```

O *parâmetro* obrigatoriamente tem que ser do mesmo tipo que os elementos do array *nomeDoArray* e é recomendável utilizá-la quando não precisarmos modificar o conteúdo dos elementos iterados. Neste caso, o código das linhas 13 e 14 podem ser substituídos por:

```

for (Pessoa pessoa : listaPessoas)
    pessoa.nomeCompleto();
```

Deitel (2016) afirma que tanto a instrução *for* quanto a *for* aprimorada iteram sequencialmente os elementos de um ponto de partida a um ponto final e que no Java8 terá uma maneira mais elegante, concisa e menos propensa a erros para realizarmos a mesma tarefa, devido a programação funcional.

Na programação funcional, uma vez que não é necessário o programador escrever explicitamente como percorrer os dados armazenados numa estrutura de dados (como vetores e listas, por exemplo), Deitel (2016) denomina esta iteração como **iteração interna**. O autor citado afirma que você pode informar a biblioteca que deseja realizar a iteração através processamento paralelo para tirar proveito da arquitetura multiprocessada, melhorando a performance. Além disto, Deitel (2016) diz que, uma vez que não modifica a origem dos dados processados ou o estado do programa, a programação funcional foca a imutabilidade.

E a iteração interna foi implementada no Java8 através do novo método *forEach()*. De acordo com o Silveira e Turini (2014), utiliza a interface Consumer – uma das novas interfaces funcionais do Java8 (consultar capítulo 2 para mais detalhes) – que contém o método *accept()*, responsável por pegar um objeto do tipo estabelecido pelo programador e realizar uma tarefa com ele – no caso, mostrar o nome completo do objeto da classe Pessoa. Se utilizarmos a implementação antiga para simular o comportamento do método *forEach()*, teríamos o código-fonte abaixo. Para este exemplo, reutilizaremos a *listaPessoas* criado no algoritmo 1 para então iterá-los, retornando o nome completo destas pessoas inseridas. Repare o quanto verboso fica o código para apenas iterar pelos elementos da classe Pessoa:

```
8.     listaPessoas.forEach(new Consumer<Pessoa> = new Consumer<Pessoa>{  
9.         public void accept(Pessoa p){  
10.             p.nomeCompleto();  
11.         }  
12.     });
```

Graças as expressões lambdas, estas cinco linhas de código compreendidas entre as linhas 8 e 11 podem ser reduzidas a um único comando que realiza a mesma tarefa de forma mais concisa, a qual pode ser lida abaixo:

1. listaPessoas.forEach(p → p.nomeCompleto());

Uma forma ainda mais enxuta é obtida através da **referência de método**. Esta é uma notação abreviada de uma expressão lambda quando o objetivo é apenas invocar um método já existente. Sua sintaxe é constituída por um tipo em seguida pelo limitador :: e o nome do método, sem parênteses (SILVEIRA E TURINI, 2014), como pode ser lido abaixo:

tipo::nomeDoMetodo

Assim, o código poderia ser escrito desta forma:

1. listaPessoas.forEach(Pessoa::nomeCompleto)

De acordo com Silveira e Turini (2014), “esse poderoso recurso é tratado pelo compilador praticamente da mesma forma que uma expressão lambda” e que, ainda de acordo com os autores citados, tudo é resolvido em tempo de compilação, sem afetar a performance do algoritmo.

Em Java8 foi introduzido o conceito de **Stream** que realizam determinadas operações sobre uma coleção de elementos, de forma análoga ao Iterator, que é um objeto que implementa uma interface *Iterator*. Medeiros (2012) afirma que o objetivo do Iterator é:

[...] acessarmos um a um os elementos de um agregado mesmo sem saber como eles estão sendo representados, assim torna-se irrelevante se a coleção de objetos está num ArrayList, HashTable ou que quer que seja. Além disso, o Padrão Iterator assume a responsabilidade de acessar sequencialmente os elementos e transfere essa tarefa para o objeto Iterador, dessa forma o objeto agregador tem a sua interface e implementação simplificadas, não sendo mais o responsável pela iteração. (MEDEIROS, 2012)

O Iterator é uma interface adotada nas Collections, que, de acordo com a apostila “Java e Orientação a Objetos” da Caelum, a define como uma “API robusta que possui classes que representam estruturas de dados avançadas”, como Listas (seja ArrayLists ou LinkedLists – listas duplamente encadeadas em Java) que permite elementos duplicados e ordenação, Conjuntos (Sets) que são coleções sem elementos duplicados e Mapas (Maps) que mapeia uma *chave* a um *valor*. A seguir, um exemplo de algoritmo que utiliza Iterator para percorrer uma Collection de Strings coleção1 para remover o conteúdo de uma coleção1 se o elemento estiver presente na Collection colecao2

```
private static void removerCores(Collection<String> colecao1, Collection<String> colecao2){
    Iterator<String> iterator = colecao1.iterator();
    while(iterator.hasNext()){
        if(colecao2.contains(iterator.next()))
            iterator.remove();
    }
}
```

Contudo, uma das principais diferenças entre Iterator e a Stream API é que, enquanto o Iterator manipula apenas um elemento da coleção por vez, a Stream API permite trabalhar com todos os elementos em sequência, através de **operações intermediárias** sobre estes dados de origem e termina com uma **operação terminal**, produzindo um resultado, através do encadeamento de métodos que o autor denomina **pipeline de fluxo**. Além disso, estes fluxos não possuem armazenamento (ao contrário das Coleções). Desta forma, depois que ele é processado, ele não poderá ser utilizado, porque não mantém uma cópia dos dados originais (Deitel, 2016). Na tabela 3 abaixo¹⁷, poderá ver algumas destas operações intermediárias:

¹⁷ A tabela 3 foi adaptada do livro “Java – Como Programar – 10ª ed, DEITEL & DEITEL - 2016

Operação	Descrição
filter	Retorna um fluxo com apenas os elementos que satisfaçam uma condição
distinct	Retorna um fluxo com apenas elementos não-duplicados
limit	Retorna um fluxo com apenas a quantidade de elementos estabelecidos como parâmetro desta operação a partir do fluxo original
map	Retorna um fluxo o qual todos os elementos do fluxo original são mapeados para um novo valor estabelecido pela expressão como parâmetro da função map.
sorted	Retorna um fluxo com os elementos do fluxo original ordenados

Tabela 4: Tabela com algumas operações intermediárias

As operações terminais de uma Stream API podem ser classificadas em operações de redução – retornando um único valor a partir de todos os valores do fluxo, operações mutáveis – criando um contêiner (como uma coleção ou um array) ou operações de pesquisa. Na tabela 4 na próxima página¹⁸, o leitor poderá ver a funcionalidade de algumas operações terminais:

Operação	Descrição	Classificação
forEach	Percorrer cada elemento de um fluxo, realizando determinado processamento.	-
average	Retornar a média numérica dos elementos de um fluxo	Redução
count	Retornar a quantidade de elementos de um fluxo	Redução
max	Retornar o maior valor numérico de um fluxo	Redução
min	Retornar o menor valor numérico de um fluxo	Redução
reduce	Reducir todos os elementos numéricos de um fluxo a um valor numérico estabelecido numa função de acumulação associativa	Redução
collect	Cria uma coleção de elementos com o resultado anterior de um fluxo	Mutável
toArray	Cria um array com os resultados anteriores de um fluxo	Mutável
findFirst	Localiza o primeiro elemento de um fluxo, terminando o processamento do pipeline	Pesquisa

¹⁸ A tabela 4 foi adaptada do livro “Java – Como Programar – 10ª ed, DEITEL & DEITEL - 2016

findAny	Localiza qualquer elemento de um fluxo de acordo com uma condição estabelecida, terminando o processamento do pipeline ao encontrá-lo	Pesquisa
anyMatch	Verifica se há algum elemento no fluxo que satisfaça a uma condição, terminando o processamento, caso o encontre.	Pesquisa
allMatch	Verifica se todos os elementos do fluxo satisfazem uma condição, terminando o processamento, caso positivo	Pesquisa

Tabela 5: Tabela com algumas operações terminais

Silva (2015) em seu artigo escrito no site da Oracle sobre as principais vantagens de uso da Stream API, cita:

A Streams API traz uma nova opção para a manipulação de coleções em Java seguindo os princípios da programação funcional. Combinada com as expressões lambda, ela proporciona uma forma diferente de lidar com conjuntos de elementos, oferecendo ao desenvolvedor uma maneira simples e concisa de escrever código que resulta em facilidade de manutenção e paralelização sem efeitos indesejados em tempo de execução. (SILVA,2016)

As expressões lambda atuam em diversas situações na programação em Java, tais como em Arrays e ArrayLists, leitura e gravação de arquivos textos, componentes gráficos e aplicações concorrentes, como será detalhado nos subtópicos abaixo.

4.1.1 Arrays e ArrayLists

De acordo com Deitel (2016), um array é um grupo de elementos que contém valores do mesmo tipo. Em Java é possível manipularmos um array unidimensional ou multidimensionais, quando é necessário representar os dados em linhas e colunas.

Para manipularmos este tipo de dado, contudo, a linguagem exige o uso de outra classe, a classe Arrays. Nela, contém métodos para organizarmos os dados do array em ordem crescente, métodos para fazer uma busca binária para encontrar um elemento, métodos para comparar arrays e métodos para preencher elementos do

array. No exemplo abaixo, ilustra a utilização de alguns destes métodos na classe Array:

```
1. public static void main(String[] args) {  
2.     int[] array = {3, 5, 9, 16, 21};  
3.  
4.     Arrays.sort(array); //retorna [3, 5, 9, 16, 21]  
5.     System.out.println(Arrays.toString(array));  
6.     System.out.println(Arrays.binarySearch(array, 9)); //retorna 2  
7.  
8.     int[] array2 = new int[5];  
9.     Arrays.fill(array2, 4); //retorna [4,4,4,4,4]  
10.    System.out.println(Arrays.toString(array2));  
11.  
12.    System.out.println(Arrays.equals(array, array2)); //retorna false  
13. }
```

Na linha 2, um novo array foi inicializado com cinco valores do tipo inteiro. Na linha 4, foi utilizado a classe Array que, de acordo com Deitel (2016), permite utilizar métodos *static* para manipular arrays comuns, sem precisar reinventar a roda. E nela foi chamado o método *sort()* que recebe um argumento, que é o objeto a ser ordenado. Na próxima linha apenas imprime o conteúdo do Array em formato de string com o método *.toString()*. Na linha 6, o método *binarySearch()* recebe dois parâmetros: o array e o elemento a ser procurado e retorna a posição do elemento, caso o encontre no vetor. Na oitava linha, inicializo um novo array e preencho todos os elementos do vetor com o valor 4 (quatro) através da função *fill()*. Em seguida, o array2 é impresso na tela e, na linha 2, verifica se os dois arrays são iguais graças ao método *equals()*. Retorna um valor booleano *true*, caso os dois arrays contenham os mesmos valores.

A linguagem JAVA possui o recurso de **coleções**, que são estruturas de dados predefinidas que permitem armazenar objetos relacionados na memória com

métodos que permite organizar, recuperar e armazenar os dados sem que conheçamos previamente como os dados são armazenados (Deitel, 2016).

A ArrayList é uma classe Java que, de acordo com Lanhellas (2013), é implementado como um Array que é dimensionado dinamicamente, uma grande vantagem com relação ao array comum com um tamanho fixo em tempo de execução, como aponta Deitel (2016). Ao inicializarmos uma arraylist no formato ArrayList<T>, sendo T uma classe, permite utilizar a mesma estrutura de dados para o armazenado de qualquer tipo de classe. A isso denomina-se **classes genéricas**.

A classe ArrayList permite o uso de alguns métodos que adiciona, remove, verifica se um elemento está contido na classe, como visto na tabela 5¹⁹ abaixo:

Método	Descrição
add	Adiciona um elemento no final do ArrayList
clear	Remove todos os elementos da ArrayList
contains	Retorna um valor booleano que responde se contém o elemento passado no parâmetro
get	Retorna o elemento do índice especificado no parâmetro
IndexOf	Retorna a posição da primeira ocorrência do elemento especificado
Remove	Remove da ArrayList, a primeira ocorrência do elemento especificado no parâmetro.
size	Retorna a quantidade de elementos da ArrayList

Tabela 6: Alguns métodos presentes na classe ArrayList

No algoritmo abaixo, demonstra-se o uso de alguns métodos da classe ArrayList:

```
public static void main(String[] args) {
    1. List<Integer> lista = new ArrayList<>();
    2. lista.add(5);
    3. lista.add(7);
    4. lista.add(12);
    5. System.out.println(lista.contains(7)); //retorna true
```

19 A tabela 5 foi adaptada do livro “Java – Como Programar – 10ª ed, DEITEL & DEITEL - 2016

```

6.     System.out.println(lista.get(2)); //retorna 12
7.     System.out.println(lista.indexOf(5)); //retorna 0
8.     lista.remove(2);
9.     System.out.println(lista.toString()); //retorna [5,7]
10.    lista.clear();
11.    System.out.println(lista.toString()); //retorna []
12. }
```

Na linha 1 foi inicializado uma lista vazia de elementos do tipo Integer. Nas linhas 2 a 4 foram inseridos três elementos com o método *add()*. Na linha 5 verifica se a lista possui um elemento cujo valor é 7 (sete), através do método *contain()*. Na linha 6, retorna o conteúdo do elemento que está na 3^a posição da lista (o primeiro elemento da lista está na posição zero na linguagem JAVA) com o método *get()*. Na linha 7 retorna a posição do elemento do conteúdo dentro do parâmetro. Na oitava linha, é removido o valor 2 da lista através do método *remove()*. Na nona linha imprime-se o conteúdo da lista. Na linha 10, removo todos os elementos da lista e, por fim, imprimo a lista novamente.

O Stream API *IntStream* permite simplificar operações com arrays e ArrayList utilizando o cálculo lambda, como pode ser comparado nos dois algoritmos abaixo. O primeiro utiliza o cálculo lambda para fazer diversas operações (se necessário, o leitor pode retornar as tabelas 3 e 4 para consultar as operações intermediárias e terminais de uma Stream API) e, no segundo algoritmo, as mesmas operações, sem o uso das expressões lambdas.

Algoritmo 1:

```

1. public static void main(String[] args) {
2.     int[] vetorDeNumeros = {16, 21, 46, 91, 18, 27, 33, 6, 12, 60, 73};
3.
4.     System.out.print("Valores: ");
5.     IntStream.of(vetorDeNumeros).forEach(numero -> System.out.printf("%d ", numero));
6.     System.out.println();
```

```
7.     System.out.println(IntStream.of(vetorDeNumeros).summaryStatistics());
8.     System.out.println("Produto      dos      valores:      "      +
IntStream.of(vetorDeNumeros).reduce((x, y) -> x * y).getAsInt());
9.
10.    System.out.print("Vetor em ordem crescente: ");
11.    IntStream.of(vetorDeNumeros).sorted().forEach(numero ->
12. System.out.printf("%d ", numero));
13.    System.out.println();
14.
15.    System.out.print("Valores impares do vetor em ordem crescente: ");
16.    IntStream.of(vetorDeNumeros).filter(numero -> numero % 2 > 0)
17.                  .sorted()
18.                  .forEach(numero -> System.out.printf("%d ", numero));
19.    System.out.println();
20.
21.    System.out.print("Vetores organizado contendo o dobro dos valores: ");
22.    IntStream.of(vetorDeNumeros).sorted().map(x -> x * 2).forEach(numero ->
23. System.out.printf("%d ", numero));
24.    System.out.println();
25.
26.    System.out.print("Os três maiores valores do vetor: ");
27.          IntStream.of(vetorDeNumeros).sorted().skip(8).forEach(numero ->
28. System.out.printf("%d ", numero));
29.    System.out.println();
30.    System.out.print("Numeros pares e maior que 20: ");
31.    IntPredicate par = valor -> valor % 2 == 0;
32.    IntPredicate maiorQue20 = valor -> valor > 20;
33.    IntStream.of(vetorDeNumeros).filter(par.and(maiorQue20)).forEach(numero
34.-> System.out.printf("%d ", numero));
35. }
```

Algoritmo 2:

```
1. public static void main(String[] args) {  
2.     int[] vetorDeNumeros = {16, 21, 46, 91, 18, 27, 33, 6, 12, 60, 73};  
3.     int qtdElementos = 0, maior = 0, soma = 0;  
4.     int menor = vetorDeNumeros[0];  
5.     double media = 0.0;  
6.  
7.     System.out.print("Valores: ");  
8.     imprimaVetor(vetorDeNumeros);  
9.  
10.    for (int numero : vetorDeNumeros) {  
11.        qtdElementos++;  
12.        if(numero < menor)  
13.            menor = numero;  
14.        if(numero > maior)  
15.            maior = numero;  
16.        soma += numero;  
17.    }  
18.  
19.    media = Double.valueOf(soma) / Double.valueOf(qtdElementos);  
20.    System.out.println("Quantidade de elementos: " + qtdElementos);  
21.    System.out.println("Menor valor: " + menor);  
22.    System.out.println("Maior valor: " + maior);  
23.    System.out.println("Soma dos valores: " + soma);  
24.    System.out.println("Média dos valores: " + media);  
25.  
26.    int produto = 1;  
27.    for (int i = 0; i < vetorDeNumeros.length; i++)  
28.        produto *= vetorDeNumeros[i];  
29.  
30.    System.out.println("Produto dos valores: " + produto);  
31.    System.out.println("Vetor ordem crescente: ");
```

```
32.    Arrays.sort(vetorDeNumeros);
33.    imprimaVetor(vetorDeNumeros);
34.    System.out.println("Valores impares do vetor em ordem crescente:");
35.    List<Integer> vetorTemp = new ArrayList<>();
36.        for (int numero : vetorDeNumeros) {
37.            if(numero % 2 > 0)
38.                vetorTemp.add(numero);
39.        }
40.    Collections.sort(vetorTemp); //organizando
41.    for (Integer numero : vetorTemp) {
42.        System.out.printf("%d ", numero);
43.    }
44.    System.out.println();
45.
46.    System.out.print("Vetores contendo o dobro dos valores: ");
47.    for (int numero : vetorDeNumeros) {
48.        System.out.printf("%d ", 2 * numero);
49.    }
50.    System.out.println();
51.
52.    System.out.print("Os três maiores valores do vetor: ");
53.    System.out.printf("%d %d %d ", vetorDeNumeros[vetorDeNumeros.length -
54.3],           vetorDeNumeros[vetorDeNumeros.length - 2],
55.vetorDeNumeros[vetorDeNumeros.length - 1]);
56.    System.out.println();
57.
58. }
59.
60. private static void imprimaVetor(int[] vetor) {
61.     for (int i : vetor) {
62.         System.out.printf("%d ", i);
63.     }
}
```

```

64.     System.out.println();
65. }
66.
67.}
```

Observando o código-fonte do algoritmo 1, a linha 5, com apenas o comando abaixo:

```
IntStream.of(vetorDeNumeros).forEach(numero -> System.out.printf("%d ", numero));
```

é possível imprimir todos os valores do vetor *vetorDeNumeros*, ao passo que no algoritmo 2 foi necessário criar um outro método *imprimaValor* que percorra cada elemento do vetor dentro de uma estrutura de repetição – no caso a instrução *for* aprimorada. O leitor pode verificar o código entre as linhas 60 a 65 do algoritmo 2.

Em seguida, com apenas uma instrução da linha 7 do algoritmo 1 abaixo foi possível retornar diversas estatísticas do vetor *vetorDeNumeros*, tais como a quantidade de elementos, a soma de todos os valores, o menor e o maior valor, além da média aritmética dos elementos

```
System.out.println(IntStream.of(vetorDeNumeros).summaryStatistics());
```

Ao rodar o algoritmo, aparece esta informação no console JAVA:

```
IntSummaryStatistics{count=11, sum=403, min=6, average=36,636364, max=91}
```

Em contrapartida, para realizarmos o mesmo comando no algoritmo 2 foi necessário inicializar cinco variáveis de controle *qtdElementos*, *maior*, *soma*, *menor* e *media* para poder armazenar estes dados, além do algoritmo compreendido entre as linhas 10 a 19.

Utiliza-se a operação de redução *reduce()* com a expressão lambda $(x, y) \rightarrow x * y$ para retornar o produto de todos os valores do vetor através de um simples comando na linha 8 do algoritmo 1.

Ao passo que, no algoritmo 2, teve que ser utilizada uma outra operação de looping para calcular o produto, como pode ser visto entre as linhas 26 e 28 do algoritmo 2.

A linha 16 do algoritmo 1 mostra um ótimo exemplo de uso do pipeline de fluxo para imprimir os valores ímpares do vetor na ordem crescente. A primeira operação intermediária *filter()* faz a seleção dos valores para permanecer no fluxo apenas os valores que atendem a expressão lambda $numero \rightarrow numero \% 2 > 0$. Em outras

palavras, os valores ímpares. Logo em seguida, o fluxo é ordenado com o comando *sort()* - outra operação intermediária para, em fim, serem impressos com a operação terminal *forEach()*, consumindo apenas uma linha de código:

```
IntStream.of(vetorDeNumeros).filter(numero -> numero % 2 > 0)
.sorted().forEach(numero -> System.out.printf("%d ", numero));
```

Para se realizar a mesma tarefa no algoritmo 2, tive que contar com uma Lista *vetorTemp* para armazenar temporariamente os valores ímpares – como não dá para prever a quantidade de valores, ela teria que ser dinamicamente alocada em tempo de execução. Por conta disto, a classe *ArrayList* era a mais adequada para a situação, como explicado no inicio deste tópico. Além disto, utilizou-se a classe *Collection* para chamar um método estático *sort()* para ordenar a lista temporária *vetorTemp*. E, por último, para imprimir o conteúdo da lista, foi utilizado a estrutura *for* aprimorada. O algoritmo ocupou as linhas 34 a 43.

Na linha 27 do algoritmo 1 abaixo retorna os três maiores valores com o método *skip(long n)*. Ele recebe um parâmetro do tipo *long* e retorna uma stream com os elementos restante após descarte dos primeiros *n* elementos passados como parâmetro. O trecho do código pode ser lido abaixo:

```
IntStream.of(vetorDeNumeros).sorted().skip(8).forEach(numero ->
28.System.out.printf("%d ", numero));
```

No algoritmo 2 utilizei uma manipulação dos índices do vetor para realizar a mesma função, uma vez que o tipo *int[]* tem um atributo *length* que retorna a quantidade de elementos do vetor. O trecho de código se encontra na linha 53 do algoritmo 2.

Entre as linhas 30 e 34 do algoritmo 1, foi utilizado a interface *IntPredicate* para definir a expressão lambda que retornaria verdadeiro se o valor fosse par e maiorQue20 através da operação lógica *par.and(maiorQue20)* passada como parâmetro da operação intermediária *filter()*. Enquanto esta operação pode ser realizada com apenas três linhas, no máximo, sem as Stream API e expressões lambdas é necessário duas estruturas de repetição para realizarmos a mesma operação, como pode ser visto abaixo:

```
1. for(int numero : vetorDeNumeros)
2.     if((numero % 2 == 0) && (numero > 20))
3.         vetorTemp.add(numero);
4. for (Integer numero : vetorTemp) {
5.     System.out.printf("%d ", numero);
6. }
```

4.1.2 Manipulação de arquivos textos

Expressões lambdas facilitam bastante as operações de leitura e gravação de arquivos texto na linguagem JAVA. De acordo com Deitel (2016), esta linguagem enxerga os arquivos como fluxo de bytes sequencial com um marcador representando o final do arquivo. Este fluxo podem ser, ainda segundo o autor, baseado em bytes – gerando arquivos binários – ou em caracteres (baseado em sequência de caracteres), que geram arquivos de texto.

No algoritmo ilustrado na figura 13 a seguir²⁰, Deitel (2016) utiliza lambdas e fluxos na manipulação de arquivos textos para contar o número de ocorrências de cada palavra dentro do arquivo. A isto, o autor citado denomina de concordância. No decorrer do código-fonte será explicado outros aspectos e expressões-lambdas não explicados nos exemplos anteriores.

20 A Figura 13 foi elaborada pelo autor

```

public class ContaPalavraLambda {
    public static void main(String[] args) throws IOException {
        // Regex que localiza um ou mais caracteres de espaço em branco consecutivos
        Pattern pattern = Pattern.compile("\\s+");

        try{
            //realiza a contagem de cada aparição da palavra
            Map<String, Long> contarPalavra = Files.lines(Paths.get
                ("arquivoTexto.txt"), StandardCharsets.ISO_8859_1)
                .map(linha -> linha.replaceAll("(?!')\\p{P}", ""))
                .flatMap(linha -> pattern.splitAsStream(linha))
                .collect(Collectors.groupingBy(String::toLowerCase,
                    TreeMap::new, Collectors.counting()));

            //exibe as palavras agrupadas pela letra inicial
            contarPalavra.entrySet().stream()
                .collect(Collectors.groupingBy(entrada -> entrada.getKey()
                    .charAt(0), TreeMap::new, Collectors.toList()))
                .forEach((letra, listaPalavras) -> {System.out.println(letra);
                    listaPalavras.stream()
                        .forEach(palavra ->
                            System.out.printf("%13s: %d%n",
                                palavra.getKey(), palavra.getValue()));
                });
        }catch(MalformedInputException e){
            e.printStackTrace();
        }
    }
}

```

Figura 13: Algoritmo que conta ocorrência de palavras num arquivo texto com Expressão lambda

Deitel (2016) utiliza a classe Pattern para dividir as linhas do texto em palavras individuais. Possui o intuito de localizar caracteres de espaço em branco consecutivo. Segundo Deitel(2016), esta classe representa uma expressão regular que possui o propósito para detectar padrões de pesquisa para validar a entrada e garantir que os dados estejam num determinado formato. Esta classe está presente no pacote java.util.regex. O parâmetro “\\s+” no método compile() é uma expressão regular que representa um espaço em branco e o sinal de + é um quantificador que procura uma ou mais ocorrências do padrão digitado.

O primeiro trecho de código logo após o “try” é responsável por contar as ocorrências de cada palavra. O resultado é retornado num mapeamento

representado pela classe Map, a qual a chave é a palavra do tipo String e o valor é o número de ocorrências da palavra.

O primeiro método lines() da classe Files cria uma Stream<String> para ler as linhas do texto. Em seguida, retira-se toda a pontuação do texto através da função map() e substituir por vazio (representado por abre e fecha dupla aspas - "") e a expressão regular \p{P} localiza qualquer carácter de pontuação – com exceção de apóstrofos. Depois, uma outra operação intermediária flatMap() é utilizada para dividir cada linha do texto em palavras individuais. De acordo com Silveira e Turini (2014), utiliza-se o método flatMap() quando queremos que o resultado de uma transformação seja reduzido a um Stream apenas, sem composição. Para ilustrar melhor o conceito, se tivesse escrito esta linha de código abaixo:

```
Map<String, Long> contarPalavra = Files.lines(Paths.get("arquivoTexto.txt"),
StandardCharsets.ISO_8859_1).map(linha -> linha.replaceAll("(?!)\p{P}", ""))
.map(linha -> pattern.splitAsStream(linha))
```

Teria como resultado um Stream<Stream<String>>. Ao usar o comando flatMap(), achatamos uma Stream de Streams para ter no final apenas um Stream<String>.

Por fim, o método collect() é responsável pela contagem das ocorrências de cada palavra e inseri-las num Treemap<String,Long>. Treemap é uma classe que implementa a interface Map – tipo de coleção em JAVA faz a associação entre uma “chave” e um “valor” - e armazena estes elementos em árvores, além de ordená-las, uma vez que Treemap implementa a interface SortedMap. As três referências de método convertem todas as strings para letra minúscula (String::toLowerCase), retorna uma coleção TreeMap que as mantém em ordem (TreeMap::new) e o Collector.counting() que realiza a contagem do número de ocorrências.

O segundo trecho do código é responsável por exibir o conteúdo do Map<String,Long>. Uma vez que o Map não tem suporte a Streams, o método entrySet() obtém um Set a partir dos objetos Map e a partir dele pode ser obtido uma stream através do método stream(). Em seguida é feito um “group by”, separando as palavras por letras do alfabeto e exibe na tela através da operação terminal forEach().

Em todo caso, com apenas duas linhas de comando subsequentes subdivididas em algumas linhas, por motivos de indentação e facilidade de leitura, foi o suficiente para pegarmos um enorme texto salvo em um arquivo de texto, fazer a contagem das ocorrências por palavras e exibir para o usuário de forma ordenada alfabeticamente. O algoritmo escrito por Deitel (2016), contudo, considerou as letras com acento (exemplo - “é”) como se fosse outra letra do alfabeto, como mostra a figura 14 abaixo²¹, que mostra o console ao executar o algoritmo, considerando que o conteúdo do arquivo “arquivoTexto.txt”, para efeito de testes, diz: “Este é o teste de um parágrafo que serve para contar a quantidade de ocorrências de cada palavra neste trecho de texto escrito”.

21 A figura 14 foi elaborada pelo autor

```

a
  a: 1
c
  cada: 1
  contar: 1
d
  de: 4
e
  escrito: 1
  este: 1
n
  neste: 1
o
  o: 1
  ocorrências: 1
p
  palavra: 1
  para: 1
  parágrafo: 1
q
  quantidade: 1
  que: 1
s
  serve: 1
t
  teste: 1
  texto: 1
  trecho: 1
u
  um: 1
é
  é: 1

```

Figura 14: Imagem do Console da IDE Netbeans ao executar o algoritmo da Figura 12

Um algoritmo equivalente ao demonstrado na figura 12, sem a utilização de Stream API e expressões lambdas, ficaria bem maior. Foi criado quatro métodos: um para abrir e ler o arquivo texto. Contém métodos como o `contaPalavras()`, responsável por contar as ocorrências de uma palavra num texto inteiro. Esta função auxilia o método `contaOcorrencias()`, que é o responsável de pegar cada palavra do texto e contar a ocorrência de cada palavra e mapeá-las em seguida.

Enquanto o algoritmo 1 ocupou apenas vinte e cinco linhas de código, este aqui ocupou por volta de setenta para realizar a mesma operação. Este algoritmo está

ilustrado na figura 15 na próxima página, com o intuito de facilitar a leitura do algoritmo.

```

public class ContaPalavra {

    private static Scanner input;
    private static String texto;
    private static char letraAtual;

    //conta as ocorrências de uma palavra só
    public static int contaPalavras(String palavra, String texto){
        int quant = 0;
        String [] palavras = texto.split(" ");
        for (String cadaPalavra : palavras) {
            if(cadaPalavra.equals(palavra))
                quant++;
        }
        return quant;
    }

    public static void main(String[] args) {

        /*pega o arquivo texto, lê linha por linha
        e retira acentos, pontos e coloca padronizado*/
        texto = ""; /*inicia o arquivo texto, evitando
        aparecer o "null" no texto*/
        abrirArquivo("arquivoTexto.txt");
        lerArquivo(input);
        String textoSemAcentosEPontosELetraMinuscula =
        texto.replaceAll("(?!')\\p{P}", "").toLowerCase();

        //conta as ocorrências
        TreeMap<String, Integer> mapa =
        contaOcorrencias(textoSemAcentosEPontosELetraMinuscula);

        //exibe a contagem (Ainda não em ordem)
        System.out.println("Palavra - Ocorrências");
        letraAtual = '#'; //tipo um "null" do letra atual
        for (Map.Entry<String, Integer> entry : mapa.entrySet()) {
            String key = entry.getKey();
            Integer value = entry.getValue(); //ocorrências
            //simulando um group by do algoritmo usando lambda
            if(key.charAt(0) != letraAtual){
                letraAtual = key.charAt(0);
                System.out.println(letraAtual);
            }
            System.out.printf("%20s\n", key + " - " + value); //20s - 20 espaços pra direita
        }
    }
}

```

Figura 15: Algoritmo equivalente ao da figura 12, sem cálculo lambda

A figura 16 abaixo²² é a continuação do algoritmo da figura 15. Contém as funções responsáveis por abrir e ler o arquivo texto – abrirArquivo() e lerArquivo() e o método contaOcorrencias() a qual conta a ocorrência das palavras do texto todo e mapeia numa estrutura TreeMap<String, Integer>

```
//conta ocorrências das palavras do texto todo
private static TreeMap<String, Integer> contaOcorrencias(String texto) {

    String [] palavras = texto.split(" ");
    TreeMap<String, Integer> mapa = new TreeMap<>();
    for (String cadaPalavra : palavras) {
        int ocorrencias = contaPalavras(cadaPalavra, texto);
        mapa.put(cadaPalavra, ocorrencias);
    }
    return mapa;
}

private static void abrirArquivo(String caminho) {

    try {
        input = new Scanner(new FileInputStream(caminho), "latin1");
    } catch (IOException e) {
        System.err.println("Erro ao abrir arquivo. Finalizando...");
        System.exit(1);
    }
}

private static void lerArquivo(Scanner input) {
    try{
        while(input.hasNext())
            texto += input.nextLine();
    } catch(NoSuchElementException e){
        System.err.println("Arquivo formatado de forma inválida. Terminando... ");
    } catch(IllegalStateException e){
        System.err.println("Erro ao ler o arquivo. Terminando... ");
    }
}
```

Figura 16: Continuação do algoritmo da figura 15

4.1.3. Componentes gráficos

Atualmente a interface gráfica está presente na maioria dos softwares. De acordo com Deitel (2016), “muitos aplicativos [...] utilizam uma interface de múltiplos documentos (multiple document interface, MDI) – uma janela principal [...] contendo outras janelas (frequentemente chamadas janelas filhas)”. Ela permite que torne mais amigável a operação de um programa de computador, com um aprendizado mais fácil e amigável e também permite que o usuário tenha um retorno visual das consequências ao realizar uma ação – como clicar um botão. Uma outra vantagem é permitir que sistemas operacionais operem em multitasking (ou seja, parelamente executar múltiplas instâncias de um programa ou múltiplos programas simultaneamente), resultando numa produtividade maior ao usuário.

A linguagem JAVA trabalha com duas bibliotecas gráficas: AWT e Swing. A Swing foi uma API que surgiu no Java 1.2 e que apresentou novas funcionalidades em relação a AWT. Esta biblioteca apresenta diversos componentes gráficos que permitem a interação com o usuário, tais como botões, entrada de texto, tabelas, abas, entre outros. Quando o usuário interage com tais componentes para realizar uma tarefa, ocorre um “evento”. Deitel (2016) denomina uma **rotina de tratamento de evento** o código-fonte responsável em responder a um evento gerado pelo usuário e o processo total de responder a eventos, como **tratamento de eventos**.

Em JAVA há duas entidades responsáveis neste tratamento de eventos. De acordo com Goldman (1999), os eventos são tratados em termos dos *event sources* e *event listeners*. O primeiro é o objeto que provoca o evento e o listener é o objeto que gostaria de ser informado quando um evento ocorre. Deitel (2016) afirma que para cada objeto que provoca o evento, há uma interface *listener* correspondente. Tanto que, ao ocorrer um evento, o componente o qual o usuário interagiu chama um método de tratamento de evento apropriado para aquele *listener*. Dentre estas interfaces *listener*, algumas são funcionais, como o ActionListener e o ItemListener, permitindo utilizar expressões lambdas para implementar rotinas de tratamentos de evento. Os próximos dois exemplos mostraremos como aplicar as expressões lambdas nos dois listeners citados acima.

O primeiro exemplo é um algoritmo que implementa um simples contador. Através da classe BtnListener, é possível detectar qual é o objeto (*source*) que provocou o evento e aplicar os incrementos ou decrementos de acordo com o botão que o usuário clicou. O código fonte apresenta-se na figura 17 a seguir²³:

```

BtnListener listener = new BtnListener();
btnCountUp.addActionListener(listener);
btnCountDown.addActionListener(listener);
btnReset.addActionListener(listener);

setTitle("AWT Counter");
setSize(400, 100);
setVisible(true);

}

public static void main(String[] args) {
    new InterfaceSemLambda();

}

private class BtnListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent evt) {
        Button source = (Button)evt.getSource(); //determina qual botão gerou o evento

        if (source == btnCountUp) {
            ++count;
        } else if (source == btnCountDown) {
            --count;
        } else {
            count = 0;
        }
        tfCount.setText(count + "");
    }
}

```

Figura 17: Código-fonte de uma interface gráfica sem expressão lambda

²³ A figura 17 foi elaborada pelo autor

Na figura 18 a seguir²⁴, o leitor poderá ver a interface gráfica em execução, com os botões *Count Up*, *Count down* e *Reset*. Quando o usuário clica em um dos botões, a classe *BtnListener* verificará qual botão gerou o evento. Se foi o botão *Count Up*, será incrementado o valor do contador (*Counter*). Caso tenha sido o *Count Down*, decrementará o valor e caso o usuário tenha clicado no botão *Reset*, o contador voltará ao estado original (valor zero).

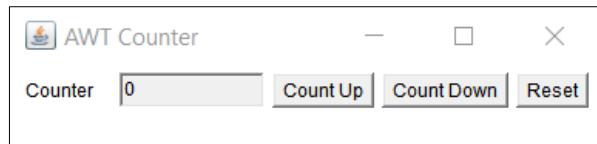


Figura 18: Janela do programa contador

De acordo com Deitel (2016), ao utilizar expressões lambdas, reduz significativamente a quantidade de linhas para ser escrita. Para corroborar o autor, ao adaptar o código, a classe *BtnListener* se tornou desnecessária, bastando implementar a funcionalidade dentro do método *addActionListener*, economizando uma boa quantidade de linhas ao código-fonte, como visto na figura 19, na página seguinte²⁵.

²⁴ A figura 18 foi elaborada pelo autor
²⁵ A figura 19 foi elaborada pelo autor

```

import java.awt.*;
import java.awt.event.*;

public class InterfaceLambda extends Frame {
    private TextField tfCount;
    private Button btnCountUp, btnCountDown, btnReset;
    private int count = 0;

    public InterfaceLambda(){
        setLayout(new FlowLayout());
        add(new Label("Counter"));
        tfCount = new TextField("0", 10);
        tfCount.setEditable(false);
        add(tfCount);

        btnCountUp = new Button("Count Up");
        add(btnCountUp);
        btnCountDown = new Button("Count Down");
        add(btnCountDown);
        btnReset = new Button("Reset");
        add(btnReset);
        addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent evt) {
                System.exit(0);
            }
        });

        btnCountUp.addActionListener(evt -> tfCount.setText(++count + ""));
        btnCountDown.addActionListener(evt -> tfCount.setText(--count + ""));
        btnReset.addActionListener(evt -> {count = 0;
            tfCount.setText(Integer.toString(count));
        });

        setTitle("AWT Counter");
        setSize(400, 100);
        setVisible(true);
    }

    public static void main(String[] args) {
        new InterfaceSemLambda();
    }
}

```

Figura 19: Código-fonte de uma interface gráfica com expressão lambda

No exemplo a seguir, Deitel (2016) mostra a aplicação das expressões lambas para tornar a rotina de tratamento de um ItemListener mais conciso. No algoritmo 1 mostra o código-fonte sem o uso de expressões lambdas e no algoritmo 2, o mesmo trecho de código é escrito com o uso de expressões lambdas.

Algoritmo 1:

```

1.imagesJComboBox.addItemListener(
2. new ItemListener()
3. {
4.     @Override
5.     public void itemStateChanged(ItemEvent event)
6.     {
7.         if (event.getStateChange() == ItemEvent.SELECTED)
8.             label.setIcon(Icons[
9.                 imagesJComboBox.getSelectedIndex()]);
10.    }
11. }
12.);
```

Algoritmo 2:

```

1.imagesJComboBox.addItemListener(event -> {
2.     if (event.getStateChange() == ItemEvent.SELECTED)
3.         label.setIcon(Icons[imagesJComboBox.getSelectedIndex()]);
4.});
```

4.1.4. Aplicações concorrentes

No tópico 4.1 foi mencionado que um dos objetivos da Stream API é facilitar a vida do programador ao paralelizar o seu código-fonte, permitindo que aproveite melhor os recursos que um computador multi-core fornece, principalmente quando lidamos com um processamento de um número muito grande de dados. No algoritmo da

figura abaixo, Deitel (2016) realiza algumas operações matemáticas num vetor de números aleatórios que, ao todo, constitui-se um *array* de 10 milhões de elementos. A classe Instant no código-fonte será responsável por cronometrar o tempo de inicio e fim das operações matemáticas. A primeira parte do algoritmo realiza a operação da contagem, soma, acha os menores e maiores valores e retorna a média aritmética de forma separada. Na segunda parte do algoritmo, utiliza-se a função `summaryStatistics()` utilizado no Algoritmo 1 do subtópico 4.1.1 sobre Arrays e ArrayLists, utilizando Streams de forma *sequencial*. E, na última parte do algoritmo, obtem-se um fluxo paralelo através do método `parallel()`, para poder tirar proveito dos processadores multinúcleo. O código-fonte do algoritmo está na figura 20 na próxima página²⁶:

²⁶ Figura elaborada pelo autor. Algoritmo retirado do livro JAVA – Como Programar (10^a Ed) – Deitel (2016)

```

public class ComparaçaoStreamStatistics {
    synchronized public static void main(String[] args) {
        SecureRandom random = new SecureRandom();

        long[] values = random.longs(100_000_000, 1, 1001).toArray();

        //PARTE 1: CALCULANDO SEPARADAMENTE
        Instant separateStart = Instant.now();
        long count = Arrays.stream(values).count();
        long sum = Arrays.stream(values).sum();
        long min = Arrays.stream(values).min().getAsLong();
        long max = Arrays.stream(values).max().getAsLong();
        double average = Arrays.stream(values).average().getAsDouble();
        Instant separateEnd = Instant.now();

        System.out.println("Calculations performed separately");
        System.out.printf(" count: %,d%n", count);
        System.out.printf(" sum: %,d%n", sum);
        System.out.printf(" min: %,d%n", min);
        System.out.printf(" max: %,d%n", max);
        System.out.printf(" average: %f%n", average);
        System.out.printf("Total time in milliseconds: %d%n%n",
            Duration.between(separateStart, separateEnd).toMillis());

        //PARTE 2: UTILIZANDO STREAM - FLUXO SEQUENCIAL
        LongStream stream1 = Arrays.stream(values);
        System.out.println("Calculating statistics on sequential stream");
        Instant sequentialStart = Instant.now();
        LongSummaryStatistics results1 = stream1.summaryStatistics();
        Instant sequentialEnd = Instant.now();

        displayStatistics(results1);
        System.out.printf("Total time in milliseconds: %d%n%n",
            Duration.between(sequentialStart, sequentialEnd).toMillis());

        //PARTE 3: UTILIZANDO STREAM - FLUXO PARALELO
        LongStream stream2 = Arrays.stream(values).parallel();
        System.out.println("Calculating statistics on parallel stream");
        Instant parallelStart = Instant.now();
        LongSummaryStatistics results2 = stream2.summaryStatistics();
        Instant parallelEnd = Instant.now();

        displayStatistics(results1);
        System.out.printf("Total time in milliseconds: %d%n%n",
            Duration.between(parallelStart, parallelEnd).toMillis());
    }

    private static void displayStatistics(LongSummaryStatistics stats) {
        System.out.println("Statistics");
        System.out.printf(" count: %,d%n", stats.getCount());
        System.out.printf(" sum: %,d%n", stats.getSum());
        System.out.printf(" min: %,d%n", stats.getMin());
        System.out.printf(" max: %,d%n", stats.getMax());
        System.out.printf(" average: %f%n", stats.getAverage());
    }
}

```

Figura 20: Algoritmo que compara cálculos em vetores utilizando fluxo sequencial e paralelo

O processador que executou o algoritmo possui 12 núcleos: seis reais e seis virtuais. De acordo com a figura 21 abaixo²⁷ que mostra os resultados finais, podemos constatar que o programa calculou de forma separada em 229 milissegundos. Utilizando fluxo sequencial, conseguiu realizar as mesmas operações em 30% do tempo. Ao ativar o paralelismo, na terceira parte do código fonte, o algoritmo foi 40% mais rápido comparado com o fluxo sequencial, totalizando em apenas 42 milissegundos.

```

Calculations performed separately
count: 10.000.000
sum: 5.005.860.636
min: 1
max: 1.000
average: 500,586064
Total time in milliseconds: 229

Calculating statistics on sequential stream
Statistics
count: 10.000.000
sum: 5.005.860.636
min: 1
max: 1.000
average: 500,586064
Total time in milliseconds: 69

Calculating statistics on parallel stream
Statistics
count: 10.000.000
sum: 5.005.860.636
min: 1
max: 1.000
average: 500,586064
Total time in milliseconds: 42

```

Figura 21: Desempenho final do algoritmo, comparando fluxo sequencial com fluxo paralelo

De acordo com Silveira e Turini (2014), é necessário que haja um tamanho de entrada muito grande para que possa tirar proveito do paralelismo de stream. Quando a coleção é muito pequena, pode ocorrer uma sobrecarga, tornando a execução mais lenta. Ao alterar a quantidade de elementos de 10 milhões para

²⁷ Figura elaborada pelo autor

apenas 100 mil, a diferença entre os três algoritmos se torna insignificante, como demonstra a figura 22 abaixo²⁸.

```

Calculations performed separately
count: 100.000
sum: 50.056.910
min: 1
max: 1.000
average: 500,569100
Total time in milliseconds: 15

Calculating statistics on sequential stream
Statistics
count: 100.000
sum: 50.056.910
min: 1
max: 1.000
average: 500,569100
Total time in milliseconds: 4

Calculating statistics on parallel stream
Statistics
count: 100.000
sum: 50.056.910
min: 1
max: 1.000
average: 500,569100
Total time in milliseconds: 4

```

Figura 22: Desempenho do algoritmo da figura 14 com 100 mil elementos do tipo long

De acordo com Schieck (2016), as Threads são o único mecanismo de concorrência suportado na plataforma principal. Este recurso permite que parte do código-fonte possa ser executado de forma independente com o programa principal. De acordo com Deitel (2016), programas Java podem ter várias threads em execução de forma concorrente. A esta capacidade, o autor chama de **multithreading**.

Uma das formas de criar Threads em Java é implementar a interface Runnable onde contém o método run() com o código-fonte a ser executado por aquela thread. Para executar a thread, basta adicionar o método start(), como mostra o algoritmo da próxima página:

²⁸ Figura elaborada pelo autor

Exemplo do algoritmo 1 de criação de Threads até o Java 8:

```
public class ExemploThread {
    public static void main(String[] args) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                //código para executar em paralelo
                System.out.println("ID: " + Thread.currentThread().getId());
                System.out.println("Nome: " + Thread.currentThread().getName());
                System.out.println("Prioridade: " + Thread.currentThread().getPriority());
                System.out.println("Estado: " + Thread.currentThread().getState());
            }
        }).start();
    }
}
```

A partir do Java 8, as expressões lambdas permitiu que se criasse Threads de uma forma mais concisa e legível e, de acordo com Schieck (2016), reduziria também o tempo de desenvolvimento. No exemplo abaixo mostra a criação de uma Thread utilizando o mesmo código do algoritmo 1 com a notação lambda:

```
public class ExemploThreadLambda {
    public static void main(String[] args) {
        new Thread(() -> {
            System.out.println("ID: " + Thread.currentThread().getId());
            System.out.println("Nome: " + Thread.currentThread().getName());
            System.out.println("Prioridade: " + Thread.currentThread().getPriority());
            System.out.println("Estado: " + Thread.currentThread().getState());
        }).start();
    }
}
```

Schieck (2016) afirma que é possível notar que, com a implementação implícita da interface Runnable, reduz-se o número de linhas de cinco para duas linhas para criar Threads.

Portanto, este tópico mostrou que, entre as vantagens do uso de expressões lambdas, uma delas é facilitar a implementação, tornando o código mais conciso e fácil de ser lido e compreendido. Manipular estruturas de dados como cálculos de vários elementos em Arrays em Java 8, permite uma performance bem superior em arquiteturas multiprocessadas ao utilizar o paralelismo fornecido pela Stream API. A implementação dos eventos em elementos gráficos torna-se mais simples, com poucas linhas de código, através dos métodos ActionListener() e ItemListener() que permitem notação lambda em Java 8 e através de sucessivos usos de métodos intermediários permitem filtrar e processar longas cadeiras de Strings de uma forma mais eficiente, como visto no exemplo que conta a ocorrência de palavras num texto.

4.2. Cálculo lambda em Kotlin

Como mencionado no subtópico 2.3.2, apesar de Kotlin ser considerada uma linguagem orientada-a-objetos, ela apresenta recursos de linguagens funcionais e expressões lambdas. De acordo com Castillo (2017), quando se tem uma possibilidade de passar funções como parâmetros de outras funções, ou conseguir retornar funções após um processamento qualquer, a linguagem se torna capaz de realizar técnicas de computação, como a *lazy evaluation*. Em outras palavras, ela adia a necessidade de computar um resultado até que seja necessário. Ferraz (2008) exemplifica com um dos poucos casos onde as linguagens interativas utilizam esta técnica no caso de operações ternárias em condicionais. Supondo o código abaixo:

$$y = (x > 10) ? (x * 2) : (x / 2)$$

Esta operação acima seria equivalente a estrutura *if-then-else* abaixo:

```
if (x > 10) then y = x * 2 else y = x / 2
```

Neste caso, a depender do valor de x, segundo o autor, apenas um dos blocos de código será executado e a outra parte seria ignorada, melhorando a performance geral do programa quando o cálculo é realizado apenas sob demanda.

Assim como já visto no subtópico 4.1 em Java, podemos aplicar a programação funcional através de sucessivas operações intermediárias e terminais em coleções (tais como listas). Jeverov e Isakova (2016) afirmam que algumas funções da biblioteca padrão em Kotlin permitem que escreva-se códigos mais simples de ser entendidos, evitando o hábito de implementar tudo por conta do programador, passo a passo.

Para reforçar o conceito acima, o exemplo abaixo mostra uma implementação de uma função que acha a pessoa mais velha numa lista de pessoas definida pela *data class* Pessoa, que possui como parâmetros o nome e a idade. Da maneira tradicional teríamos este algoritmo:

```
1.data class Pessoa(val nome: String, var idade: Int)
2.fun acheOMaisVelhoSemLambda(pessoas: List<Pessoa>){
3.    var maiorIdade = 0
4.    var oMaisVelho: Pessoa? = null
5.    for(cadaPessoa in pessoas){
6.        if(cadaPessoa.idade > maiorIdade){
7.            maiorIdade = cadaPessoa.idade
8.            oMaisVelho = cadaPessoa
9.        }
10.    }
11.    println(oMaisVelho?.nome)
12.}
```

A abordagem tradicional exige a criação de duas variáveis intermediárias (`maiorIdade` e `oMaisVelho`) para armazenar os valores, além de iterar a lista, atualizando os valores, o que torna o código propenso a erros. Caso não esteja acostumado a sintaxe de Kotlin, o ponto de interrogação ao lado do tipo `Pessoa` e do nome “`oMaisVelho`” é um teste de segurança para verificar se a variável aceita valores nulos.

Uma das funções presentes na biblioteca padrão Kotlin é a `maxBy`. Ela recebe apenas um parâmetro que é a função que define qual valor deve ser comparado para achar o maior elemento. Desta forma, doze linhas de código podem ser reduzidas para apenas uma linha de código abaixo:

```
println(listaDePessoas.maxBy{it.idade}?.nome)
```

O elemento de uma coleção inteira é referenciado através do `it`. O `idade` é o atributo que é o valor a ser comparado. Por debaixo dos panos, o `it.idade` é uma expressão lambda representada por:

```
listaDePessoas.maxBy{p: Pessoa → p.idade}
```

Que pode ser substituído pela referência de método (consultar tópico 4.1) abaixo:

```
listaDePessoas.maxBy(Pessoa::idade)
```

Jeverov e Isakova (2016) aconselha apenas o uso do “`it`” quando não se é claro inferir o contexto do argumento e que, na maioria dos casos, deve-se optar sempre em explicitar os parâmetros como nos códigos acima.

Outro método padrão da linguagem Kotlin – que também se encontra presente em Java – é o método `forEach()` - que é o formato mais conciso do que um tradicional `for` de iterar e manipular coleções. O leitor poderá ler mais detalhes sobre este tipo de iteração interna no subtópico 4.1 – Cálculo Lambda em Java. O que difere as duas linguagens, contudo, é o fato de que em Kotlin não se está limitado a manipular

constantes – ou como o Jeverov e Isakova (2016) denominam “variáveis finais”. Ou seja, é possível modificar valores de variáveis dentro de uma função lambda

O exemplo do contadorDeErros acima demonstra o uso de um dos conceitos da programação funcional chamado *closures*. Segundo Jeverov e Isakova (2016), a define como uma função que tem acesso a variáveis e parâmetros definidos no escopo acima ou fora da função que a chamou.

Para demonstrar isto, o exemplo abaixo quantifica quais erros são por parte do cliente e quais destes erros são responsáveis pelo servidor. A função recebe uma lista de *Strings* de mensagem de erros. Embora as variáveis *errosCliente* e *errosServidor* foram declaradas no contexto da função contadorDeErros, a função lambda *forEach* não é só capaz de ler estas variáveis mas também modificá-las. O código-fonte se encontra na figura 23 abaixo²⁹:

```
fun contadorDeErros(respostas: Collection<String>){
    var errosCliente = 0; var errosServidor = 0

    respondas.forEach{
        if(it.startsWith("4")) errosCliente++
        else if(it.startsWith("5")) errosServidor++
    }

    println("$errosCliente erros do cliente, $errosServidor erros do servidor")
}

fun main(args: Array<String>){

    val respondas = listOf("200 OK", "404 Not Found", "403 Forbidden", "500 Internal Server Error")
    contadorDeErros(respostas)

}
```

Figura 23: Função contadorDeErros em Kotlin

Já em Java 8 isto não seria possível e, para demonstrar esta limitação de escopo, ao tentar rodar o pequeno trecho de código abaixo:

1. void fn() {
2. int myVar = 42;
3. Supplier<Integer> lambdaFun = () -> myVar;
4. myVar++;
5. System.out.println(lambdaFun.get());
6. }

²⁹ Código-fonte adaptado do livro Kotlin in Action (2016) - JEMEROV, D., ISAKOVA, S., p. 126

De acordo com a figura 24 abaixo³⁰ apontará um erro de compilação, alertando ao desenvolvedor que apenas variáveis locais referenciadas em uma expressão lambda devem ser finais.

```

17 public class LambdaLimitations {
18
19     // Local variable 'myVar' is referenced from a lambda expression
20     Supplier<Integer> lambdaFun = () -> myVar;
21     myVar++;
22     System.out.println(lambdaFun.get());
23 }
24
25
26     public static void main(String[] args) {
27         fn();
28     }
29 }
```

Figura 24: Erro de compilação ao tentar incrementar a variável myVar

Mannino (2017) explica que Java apenas armazena os valores de variáveis livres para que sejam usadas dentro de expressões lambdas e mesmo que haja um incremento nesta variável, o resultado retornaria 42 da mesma maneira e isto contribui para que o compilador evite a criação de cenários incoerentes, como demonstra o exemplo abaixo:

```

1. public static Map<String, Supplier> createCounter(int initialValue) { // the enclosing
2. scope
3.     int count = initialValue;
4.     Map<String, Supplier> map = new HashMap<>();
5.     map.put("val", () -> count);
6.     map.put("inc", () -> count++);
7.     return map;
8. }
```

³⁰ Figura elaborada pelo autor. Código-fonte extraído do link <<https://dzone.com/articles/java-8-lambdas-limitations-closures>>

O objetivo do algoritmo acima é realizar um mapeamento de valor que liga a String “val” ao valor passado como parâmetro na variável *count*, que recebe como parâmetro um número inteiro *initValue*. E, em seguida, criar um outro mapeamento com o seu valor incrementado. Mannino (2017), contudo, alerta que uma vez que a linguagem Java armazena as variáveis de funções na pilha e que elas são eliminadas no término da função, se o compilador permitisse que a segunda expressão anônima lambda na linha 6 do algoritmo acima permitisse alterar a cópia da variável *count*, seria muito confuso e que, para tal, a linguagem deveria salvar o escopo na pilha para permitir que a variável permaneça sendo utilizada após o término da primeira expressão anônima lambda na linha 5.

4.2.1 Manipulação de Coleções em Kotlin

No tópico 4.1 – Cálculo Lambda em Java, foi explicado conceitos de operações intermediárias e terminais para a manipulação de coleções em Java. As tabelas 3 e 4 mostram as principais funções intermediárias e terminais que a linguagem Java permitia. Em Kotlin também possui diversas funções que tem como objetivo a simplificação do código quando trabalha com coleções, tais como *filter*, *map*, *all*, *any*, *count*, *find*, *group by*, entre outras. De acordo com Jeverov e Isakova (2016), estas funções não são invenções dos desenvolvedores da linguagem. Pelo contrário, elas estão presentes em todas as linguagens que suportam linguagem lambda. Por conta disto, a explicação delas será mais sucinta, uma vez que o leitor pode consultar os subtópicos 4.1.1 e 4.1.2 onde estas foram profundamente exploradas em Java 8.

As funções principais são a *filter* e *map*. A primeira tem o propósito de remover elementos indesejáveis dentro de uma coleção e a *map* aplica uma função em cada elemento e retorna uma nova coleção.

No exemplo abaixo temos uma lista de pessoas com seus respectivos nomes e idades e apenas imprime o nome das pessoas que possuem mais de 25 anos. Ainda estamos utilizando a *data class* Pessoa com os atributos nome e idade dos exemplos anteriores.

```

1.val listaDePessoas = listOf(Pessoa("Lucas", 25), Pessoa("Anderson", 29),
2.      Pessoa("Thiago", 16),
3          Pessoa("Ana", 30), Pessoa("Pedro", 27), Pessoa("Carlos", 31),
4.          Pessoa("Thiago", 16), Pessoa("Carla", 14), Pessoa("Igor", 36))
5. println(listaDePessoas.filter{it.idade > 25}.map{it.nome})

```

O mapeamento coleta cada objeto da classe Pessoa dentro da lista *listaDePessoas* e retorna apenas o atributo nome desta classe. Se eu, programador, quisesse criar uma função que se comportasse de forma similar a linha 5 do algoritmo acima, uma possível solução seria esta:

```

1.fun maisDe25AnosSemLambda(pessoas: Collection<Pessoa>){
2.    for(pessoa in pessoas){
3.        if(pessoa.idade > 25){
4.            println(pessoa.nome)}
5.    }
6.}

```

A função que retorna o nome de todas as pessoas que possui mais de 25 anos pode ser escrita de outra forma utilizando predicados em Kotlin. Estes predicados são passado como parâmetros dos métodos *all*, *any*, *count* e *find*. *All* retorna verdadeiro se todas os elementos da coleção satisfaz a condição do predicado. *Any* retorna verdadeiro se pelo menos um satisfazê-la. *Count* retorna a quantidade de elementos que o predicado for verdadeiro e, por fim, o *find* retorna o primeiro elemento que satisfaz a condição.

O algoritmo abaixo mostra o uso destas quatro funções citadas na página anterior:

```

1.val listaDePessoas = listOf(Pessoa("Lucas", 25), Pessoa("Anderson", 29),
2.Pessoa("Thiago", 16),
3.                  Pessoa("Ana", 30), Pessoa("Pedro", 27), Pessoa("Carlos", 31),
4.                  Pessoa("Thiago", 16), Pessoa("Carla", 14), Pessoa("Igor", 36))
5. val temMaisDe25Anos = {p: Pessoa -> p.idade > 25} //predicado
6.println(listaDePessoas.all(temMaisDe25Anos)) //retorna false
7.println(listaDePessoas.any(temMaisDe25Anos)) //retorna true
8.println(listaDePessoas.count(temMaisDe25Anos)) //retorna 5
9.println(listaDePessoas.find(temMaisDe25Anos)) //retorna Pessoa("Anderson", 29)

```

O *groupBy* é responsável em mapear os elementos em grupos de acordo com um parâmetro. O exemplo abaixo mostra o mapeamento de uma lista de Strings categorizados de acordo com a primeira letra da palavra, de forma parecida ao exemplo do contador de ocorrência de palavras da Figura 10.

```

1.val strings = listOf("ab", "a", "b", "c")
2.println(strings.groupBy(String::first)) //retorna {a=[ab, a], b=[b], c=[c]}

```

A função *flatMap* é responsável em mapear cada elemento em uma coleção além de combinar várias listas em uma só. Esta combinação é responsável pela função *flatten*. Para exemplificar, supondo que haja uma classe *Livros* com dois atributos: o título do livro (tipo *String*) e autor(es) do livro. Considerando que um livro pode ter um ou mais autores, o tipo mais adequado para representar seria uma lista de Strings (*List<String>*).

No algoritmo abaixo, o resultado esperado seria uma lista de Strings com o nome de todos os autores dos livros cadastrados na lista *listaDeLivros*.

```

1.val listaDeLivros = listOf(Livro("Kotlin in Action", listOf("Dmitty Jeremov", "Svetlana
2.Ivakova")),
3.    Livro("Programming Kotlin", listOf("Stephen Samuel", "Stefan Bocutiu")),
4.    Livro("Java 8 Prático", listOf("Paulo Silveira", "Rodrigo Turini")))
5.
6.    println(listaDeLivros.flatMap { it.autores }.toString())
7. /* [Dmitty Jeremov, Svetlana Iakovova, Stephen Samuel, Stefan Bocutiu, Paulo 8.
Silveira, Rodrigo Turini] */

```

A figura 25 abaixo³¹ demonstra o funcionamento do flatMap aplicado na linha 6 do algoritmo acima. Primeiramente ocorre um mapeamento para extrair o nome dos autores que estão armazenados no formato de lista de Strings. Em seguida, o flatten() combina as duas listas em uma só. E, por fim, a função toString() transforma esta lista única em formato de String, de onde provém o resultado final descrito – em comentários – na linha 7 do algoritmo acima.

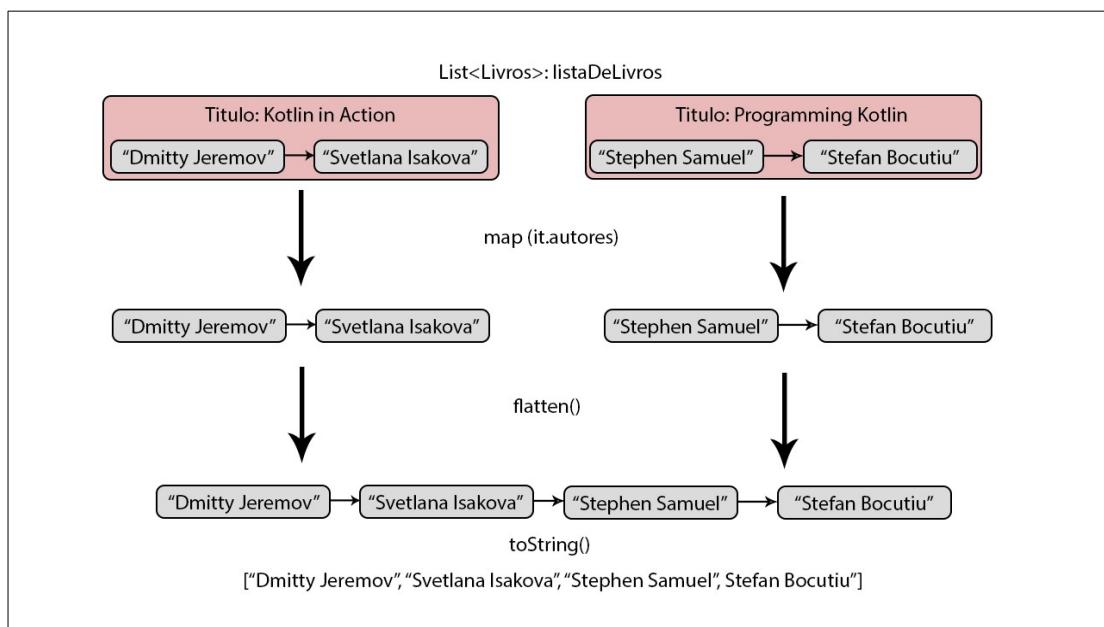


Figura 25: Funcionamento do método flatMap() em Kotlin

³¹ A figura foi elaborada pelo autor, adaptada da figura 5.6 presente no livro “Kotlin in Action (2016)” p. 137

Jeverov e Isakova (2016) afirma que há muitas outras funções que manipulam coleções em Kotlin e aconselha que deve-se consultar sempre a documentação da biblioteca que realize a transformação que o desenvolvedor deseja, em vez de tentar utilizar a abordagem da implementação manual. A documentação completa pode ser encontrada neste link: <https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/index.html>

4.2.2 Sequências em Kotlin

Até aqui, o leitor percebeu que foram utilizadas funções em cadeira, tal como o exemplo abaixo:

```
println(listaDePessoas.filter{it.idade > 25}.map{it.nome})
```

A cada uso consecutivo dos métodos *filter* e *map* geram uma lista intermediária onde o resultado destas operações são armazenadas. São, portanto, operações custosas e que pode ser um problema quando lidamos com uma coleção com milhões de elementos.

Kotlin oferece um recurso que soluciona este problema denominado **sequências**. Além de suportar as APIs de coleção com seus vários métodos vistos no tópico 4.2.1 e são operações preguiçosas (*lazy*), ou seja, não há o armazenamento temporário de listas em cada operação intermediária (consultar tabela 3). Jeverov e Isakova (2016) sugere que nos casos que tiver uma grande coleção e uma cadeia de operações intermediárias a utilizar sequências.

Para transformar a operação acima em sequência, utilizamos o método *asSequence()* - (1) - e listamos cada operação intermediária em uma quebra de linha (2). E, no final, devemos transformar a sequência de volta em coleção (3), principalmente quando utilizamos algum método API como visto abaixo:

```
listaDePessoas.asSequence() (1)
    .filter{it.idade >= 30} (2)
    .map(Pessoa::nome)
    .toList() (3)
```

A ordem a qual os métodos *filter* e *map* são chamados é fundamental para a performance do programa. Aconselha-se a utilizar o *filter* primeiro para limitar o número de transformações realizadas, como visto na figura 26 abaixo³²:

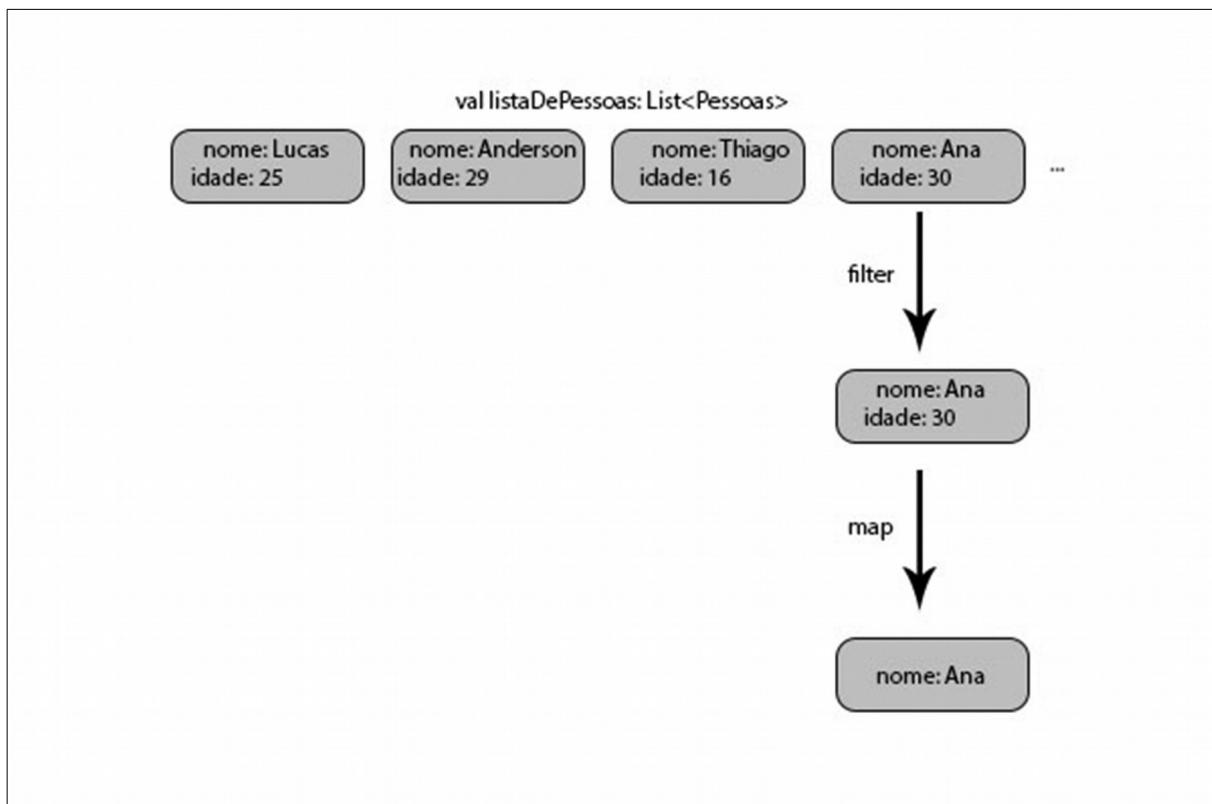


Figura 26: Utilização do método filter e depois o map

A figura 27 demonstra que caso invertesse a ordem da chamada dos métodos *filter* e *map*, teria uma quantidade bem maior de transformações a serem computadas.

³² As figuras 26 e 27 foram elaboradas pelo autor, adaptada da figura 5.9 do livro “Kotlin in Action” (2016) p. 142

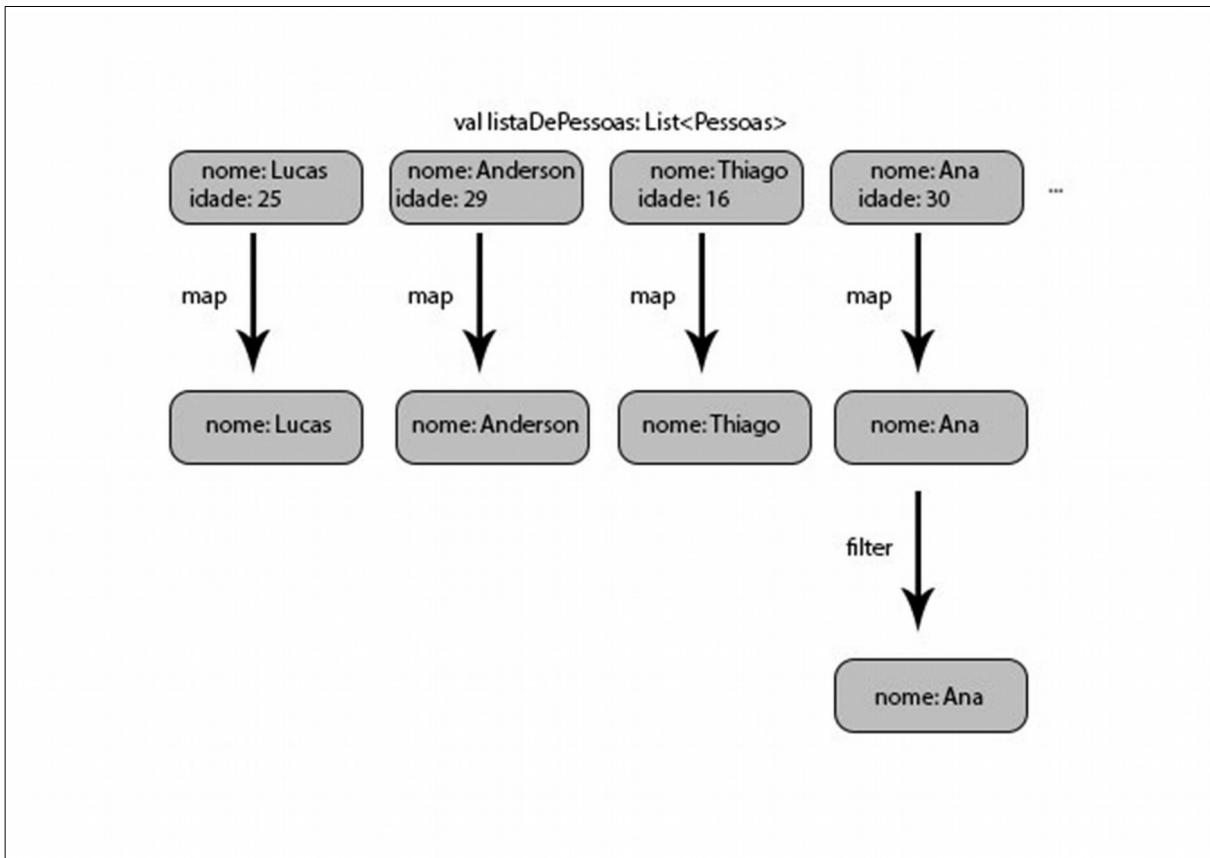


Figura 27: Utilização do método map antes do filter

As sequências em Kotlin funcionam de forma análoga a Stream API em Java 8. Assim, no caso de trabalhar com versões antigas do Java que não suportam este recurso (como programação Android), é interessante que se trabalhe com sequências, uma vez que Kotlin possui interoperabilidade com Java. A desvantagem das sequências em Kotlin é a incapacidade de realizar o paralelismo com as operações intermediárias.

4.2.3. Funções de Alta Ordem e Interface Gráfica em Kotlin

Uma função de alta ordem (*High Order Functions*) é uma função que tanto aceita uma função como parâmetro, retorna uma função como seu valor de retorno ou ambos os casos (SAMUEL e BOCUTIU, 2017).

O exemplo abaixo demonstra o caso quando utilizamos a função como parâmetro. Repare no corpo da função *funçãoPrimária*: ela recebe dois parâmetros: uma String

str é uma função, que recebe String como parâmetro e retorna uma String. Para testar se o método funciona, passo a função *toUpperCase()* como parâmetro. Ela converte todos os caracteres da string em letra maiúscula. A variável *x* será responsável em aplicar a função recebida no parâmetro da *funçãoPrimeira*.

```

1.fun funcaoPrimeira(str: String, funcao: (String) -> String){
2.    val x = funcao(str)
3.    println(x)
4.}
5. funcaoPrimeira("teste", { it.toUpperCase() } )

```

A linguagem Kotlin permite que, assim como podemos ter string literais apenas delimitando-a entre aspas duplas (exemplo: "Olá!"), é possível realizar a mesma coisa com funções. Aquelas que são limitadas entre chaves são conhecidas como **função literais**, como visto na linha cinco do exemplo acima.

Além disto, uma vez que há apenas um único parâmetro, o compilador da linguagem Kotlin é esperto suficiente para omití-las completamente, tornando-a implícita através da palavra reservada *it* (SAMUEL e BOCUTIU, 2017)

O próximo exemplo demonstra o que Samuel e Bocutiu menciona sobre uma função retornar uma outra função. Ao atribuir a função *funcaoSegunda()* a variável *y*, ela retornará uma função String que retorna String, como mostra o corpo da *funçãoSegunda()*. A função delimitada após as chaves é o corpo da função a ser retornada. De acordo com a documentação oficial da linguagem Kotlin, a função *removePrefix()* retornará uma cópia da String com o prefixo removido. Caso contrário, retornará a própria String. No exemplo abaixo, a String que iria retornar seria "computador"

```

1. fun funcaoSegunda(): (String) -> String = { s -> s.removePrefix("micro")}
2. val y = funcaoSegunda()
3. println(y("microcomputador"))

```

Com este conceito em mente, da mesma maneira vista no Java 8, na linguagem Kotlin é possível escrever ActionListeners de forma mais concisa e, inclusive, conforme Jemerov e Isakova (2016), Kotlin permite inclusive a utilizar lambdas na chamada de métodos Java que possui interfaces funcionais como parâmetros, como visto no trecho de código abaixo:

```
button.setOnClickListener({ view -> doSomething() })
```

Como pode ser visto abaixo, a função setOnClickListener recebe uma função listener, recebendo um View como parâmetro e um Unit como retorno. Este tipo entenda-se como um null ou void (ou seja, a função não retorna valor). Uma vez que a função setOnClickListener recebe uma função como parâmetro, é considerada uma função de alta ordem.

Fun setOnClickListener(listener: (View) → Unit)

Os parênteses antes das chaves podem ser removidos quando a função lambda é o único parâmetro da função, melhorando a legibilidade do código:

```
button.setOnClickListener { doSomething() }
```

Comparado com a estrutura sintática pré-Java 8, houve uma economia de seis linhas de código para apenas uma, como visto abaixo:

Estrutura básica da função setOnClickListener até Java 8:

```
1.button.setOnClickListener(new OnClickListener(){  
2.    @Override  
3.    public void onClick(View v){  
4.        doSomething();  
5.    }  
6.});
```

4.2.4: Currying e Memoization: Outros recursos da programação funcional em Kotlin

No capítulo 2 foi visto que nos anos 20, o matemático soviético Schönfinkel descobriu que poderia resolver funções de múltiplas variáveis através de sucessivas funções simples de uma só variável e, baseado nos estudos do Schönfinkel, o Haskell Curry a denominou como currying.

A linguagem Kotlin permite que transformemos uma função com múltiplos parâmetros como a função abaixo:

```
fun funcao(a: String, b:Int) : Boolean
```

Neste formato onde passo apenas uma variável como parâmetro da função, porém retorna uma função com os demais parâmetros (*Int* e *Boolean*).

```
fun funcao(a: String): (Int) → Boolean
```

Currying é uma técnica muito útil em permitir funções com vários parâmetros em trabalhar com outras funções que aceitam apenas um único parâmetro (SAMUEL e BOCUTIU, 2017).

Para exemplificar o conceito de uma forma simples, supomos que haja uma função soma escrita em Kotlin desta maneira:

```
1.fun soma(a: Int, b: Int): Int {  
2.    return a + b  
3.}
```

É uma função bastante simples, recebendo dois parâmetros do tipo Inteiro e retorna também um Inteiro – o resultado final da soma. De acordo com a definição mencionada acima, uma versão da mesma função utilizando a técnica do Currying, ficaria deste jeito na linguagem Kotlin:

```

1.fun somaCurrying1(a: Int): (Int) -> Int {
2.    return {b: Int -> a + b}
3.

```

A função *somaCurrying1* recebe apenas um parâmetro do tipo Inteiro e, em vez de retornar um valor inteiro, neste caso retorna uma outra função que recebe um inteiro e retorna outro valor do tipo inteiro, conseguindo, desta maneira, transformar uma função com múltiplas variáveis em várias funções simples de uma variável apenas. Porém, existe uma maneira mais enxuta de escrever o mesmo código acima em Kotlin, como demonstrado abaixo:

```
1. fun somaCurrying2(a: Int) = {b: Int -> a + b}
```

Comparando a maneira que chamamos o método *soma* com o método *somaCurrying2* há uma pequena alteração. Em vez de colocarmos todos os parâmetros dentro dos parênteses, cada parâmetro estará delimitado em um par de parênteses, como no exemplo abaixo, o qual desejo imprimir o valor da soma “7 + 15”

Utilizando o método *soma*:

```
1. println(soma(7,15))
```

Utilizando o método *somaCurrying2*:

```
1. println(somaCurrying2(7)(15))
```

Samuel e Bocutiu (2017) também afirmam que o conceito de função parcial (*Partial application*, em inglês) é relacionado com o conceito de currying e que é definido por aquele o qual alguns parâmetros são especificados previamente, retornando uma nova função com os parâmetros faltando.

Para demonstrar isto em Kotlin, se quiséssemos criar uma função que somasse dois ao valor inserido pelo usuário, a função teria este resultado abaixo:

```
val adicionar2 = {b: Int -> soma(2, b)}
```

A variável *adicionar2* é atribuída uma função que chama outra função já criada – a função *soma*, já atribuindo dois ao parâmetro *a*. Para utilizá-la no código, basta inserir como parâmetro o valor que deseja somar, como 7, por exemplo. O código que imprimirá o resultado se encontra abaixo:

1. `println(adicionar2(7)) //retornará 9`

Outra forma de realizar a mesma funcionalidade em Kotlin é através do uso da função auxiliar (*helper function*, em inglês). De acordo com o portal de Ciência da Computação da *Worchester Polytechnic Institute*, uma função auxiliar é aquela que realiza parte da computação de outra função e tem o objetivo de tornar os programas mais legíveis.

No exemplo abaixo, extraído do Stack Overflow, mostra como utilizar uma função auxiliar para demonstrar o conceito da função parcial.

```
1.fun <A, B, C> partial2(f: (A, B) -> C, a: A): (B) -> C {
2.    return { b: B -> f(a, b)}
3.
4.val adicionar2 = partial2(::soma, 2)
5.val resultado = adicionar2(7)
```

De acordo com Mike Hearn (2015), recursos como Currying e *Partial Application* que se encontram em outras linguagens funcionais como F# e Haskell, não possui suporte em Kotlin na sua biblioteca padrão. Porém, há uma biblioteca chamada funKtionale que suporta estes recursos à linguagem de uma forma natural.

Memoization é uma outra técnica da programação funcional que utiliza o cache para agilizar a computação em vez de recalcular o mesmo valor quando lhe é chamado. Uma forma de demonstrá-la é através da sequência de Fibonacci, que pode ser programada como uma função recursiva em Kotlin, como descrito abaixo:

```

1. fun fibonacci(k: Int): Long = when (k) {
2.     0 -> 1
3.     1 -> 1
4.     else -> fibonacci(k - 1) + fibonacci(k - 2)
5. }
```

Lipschutz e Lipson (2013) definem sequência de Fibonacci como aquela que seja os dois primeiros termos $F_0 = 0$ e $F_1 = 1$, cada termo seguinte é resultado da soma dos dois termos anteriores. Só que aplicando a função fibonacci acima terá várias vezes que será feito o cálculo do mesmo número várias vezes devido a sua aplicação de forma recursiva. Supondo $k = 5$, ao resolver $\text{fibonacci}(5)$, será chamado $\text{fibonacci}(4) + \text{fibonacci}(3)$. Para calcular $\text{fibonacci}(4)$ teria que calcular $\text{fibonacci}(3) + \text{fibonacci}(2)$, assim por diante.

A figura 28 a seguir³³ mostra as ramificações criadas para calcular $\text{fibonacci}(8)$. Perceba que a maioria dos ramos é para calcular o mesmo número várias vezes. Por exemplo, $\text{fibonacci}(2)$ foi calculado 13 vezes para chegar no resultado de $\text{fibonacci}(8)$, tornando este método ineficiente.

33 Figura adaptada do livro Programming Kotlin (2017) p.160

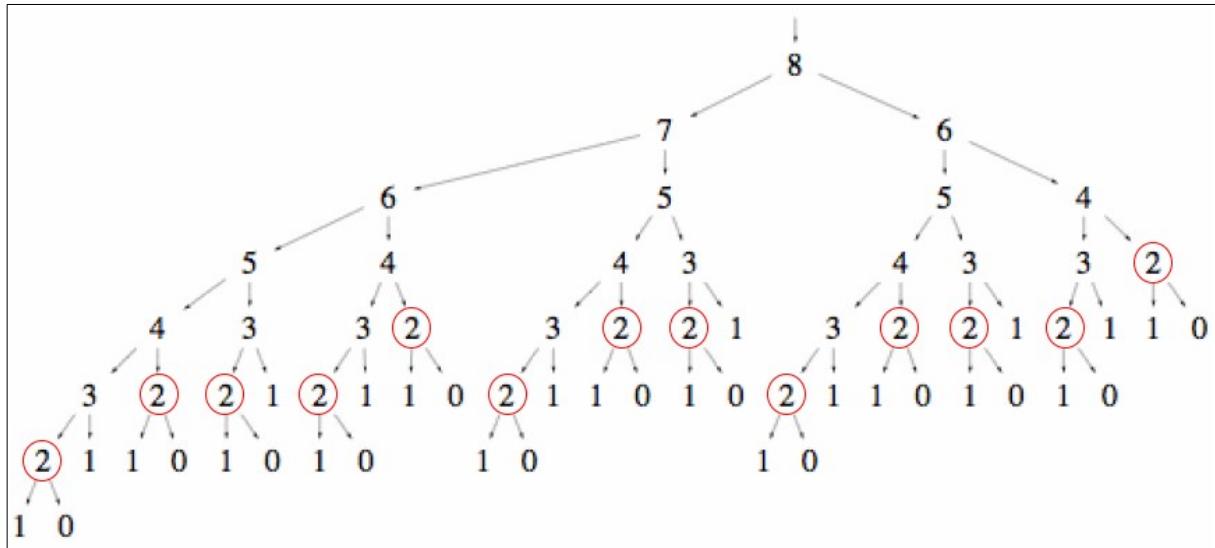


Figura 28: Ramificação após cálculo recursivo da Sequência de Fibonacci

A implementação abaixo utiliza uma variável *map* do tipo Map como a cache para armazenar resultados temporários da sequência de Fibonacci. A função memfibonacci já aplica a técnica do *memoization* para a realização destes cálculos.

```

1. val map = mutableMapOf<Int, Long>()
2. fun memfibonacci(k: Int): Long {
3.     return map.getOrPut(k) {
4.         when (k) {
5.             0 -> 1
6.             1 -> 1
7.             else -> memfibonacci(k - 1) + memfibonacci(k - 2)
8.         }
9.     }
10. }

```

Da mesma maneira que foi feita na figura 16 para calcular o tempo do algoritmo para provar que o paralelismo economizava bastante tempo em enormes cálculos de vetores de 10 milhões de elementos, o código-fonte abaixo utiliza-se as classes Instant e Duration de Java para calcular o tempo que a função fibonacci leva para retornar o resultado de fibonacci(50) em Kotlin para comparar o quanto tempo o

método *fibonacci* era ineficiente comparado ao método *memfibonacci*. A figura 29 abaixo³⁴ mostra o algoritmo usado:

```

fun main(args: Array<String>){

    fun fibonacci(k: Int): Long = when (k) {
        0 -> 1
        1 -> 1
        else -> fibonacci(k - 1) + fibonacci(k - 2)
    }

    val map = mutableMapOf<Int, Long>()

    fun memfibonacci(k: Int): Long {
        return map.getOrPut(k){
            when(k) {
                0 -> 1
                1 -> 1
                else -> memfibonacci(k - 1) + memfibonacci (k - 2)
            }
        }
    }

    var inicio: Instant = Instant.now()
    var resultado = fibonacci(50)
    var fim: Instant = Instant.now()
    println("Resultado de Fibonacci(50): $resultado")
    print(Duração do programa (sem Memoization):")
    println(Duration.between(inicio, fim).toMillis())

    var inicio2: Instant = Instant.now()
    var resultado2 = fibonacci(50)
    var fim2: Instant = Instant.now()
    println("Resultado de Fibonacci(50): $resultado2")
    print(Duração do programa (com Memoization):")
    println(Duration.between(inicio, fim).toMillis())

}

```

Figura 29: Algoritmo que calcula o tempo de processamento dos métodos da sequência de Fibonacci

A figura 30 a seguir mostra o resultado do processamento do algoritmo do código da figura 29 anterior. O hardware utilizado para a realização dos testes é o mesmo que do algoritmo Java8 da figura 20 (consultar subtópico 4.1.4 – Aplicações concorrentes). Percebe-se que a diferença de tempo é absurda. Um algoritmo

³⁴ As figuras 29 e 30 foram elaboradas pelo autor

demorou 77413 milissegundos (77 segundos ou 1:07 minutos) para processar o mesmo resultado que o outro algoritmo, com auxilio de cache, levou 1 milissegundo.

```
Resultado de Fibonacci (50) : 20365011074
Duração do programa (sem Memoization): 77413
Resultado de Fibonacci(50): 20365011074
Duração do programa (com Memoization): 1
```

Figura 30: Resultado do processamento do algoritmo da figura 29

Desta forma, corrobora o que Samuel e Bocutiu (2017) afirma que o método memoization é um comprometimento (*trade-off*) entre memória utilizada e velocidade.

Apesar de citado alguns recursos de linguagem funcional em Kotlin neste subtópico, para Jorge Castillo (2017), destaca que ela ainda peca em oferecer alguns recursos que, para ele, são fundamentais para que esta linguagem de programação seja considerada Programação Funcional, como *Type Classes* e *Higher Kind Types*, além de faltar alguns construtores e abstrações que, segundo o autor, sem estes recursos, Kotlin não poderia ser considerada uma linguagem de programação funcional de forma completa. Porém, ele confessa, durante seu artigo no site Medium, que o que define se uma linguagem é funcional ou não depende do ponto de vista de cada um ou do conteúdo o qual o leitor consome e que, ao adicionar estes recursos que permitem uma abstração de mais alto nível à linguagem, é possível que, no futuro, a linguagem Kotlin consiga atingir a este patamar de linguagem funcional.

4.3. Considerações finais sobre o capítulo 4

No decorrer do capítulo 4, o leitor percebeu que o paradigma declarativo funcional permite que realizemos mais tarefas com pouco código-fonte. Tanto na linguagem Java, quanto na linguagem Kotlin, as operações intermediárias substituem uma

quantidade considerável de código-fonte, principalmente em operações, tais como cálculo de média, valor máximo, valor mínimo, que exigem o uso de variáveis para armazenamento de valores intermediários, que podem acarretar em erros lógicos do programa, caso o desenvolvedor a inicialize de forma indevida. No exemplo do algoritmo do contador de ocorrências nas figuras 13, 15 e 16, com apenas dois comandos, este era capaz de retirar a pontuação do texto, contar as ocorrências, ordenar e agrupar as palavras de acordo com a sua letra inicial enquanto, no segundo algoritmo, sem o uso do cálculo lambda, houve a necessidade de criar um método para contar a ocorrência de uma palavra no texto (o método contaPalavras) e um outro método que chamaria o contaPalavras() para todas as palavras do texto (O método contaOcorrencias), que também adicionaria o resultado numa estrutura TreeMap. Criar componentes gráficos e adicionar um ActionListener tornaram-se menos verbosos, como visto no algoritmo da figura 19, em Java e também no tópico 4.2.3 – *Funções de Alta Ordem e Interface Gráfica em Kotlin*. Por fim, o recurso do paralelismo presente no Stream API em Java e o memoization em Kotlin permitem a manipulação de uma quantidade muito grande de dados com um melhor aproveitamento dos recursos dos processadores multinúcleos. Os resultados vistos na figura 21 ao rodar o algoritmo da figura 20 no hardware citado naquele tópico foram próximos aos encontrados pelo Deitel (2016), como visto na tabela 7 a seguir³⁵:

	Testes feitos pelo Hardware citado na página 79	Testes realizados por Deitel	Diferença em relação a Deitel
Cálculos realizados separadamente	229	173	+32%
Cálculos realizados em Stream sequencialmente	69	69	0%
Cálculos realizados em Stream de forma paralela	42	38	+10%

Tabela 7: Comparação de resultados obtidos por Deitel e pelo autor do trabalho

³⁵ Tabela criada pelo autor. Os resultados obtidos pelo Deitel encontram-se na obra “Java – Como Programar (10ª ed) p. 801

Além disto, o resultado obtido pela execução do programa de Fibonacci sem a técnica do memoization, visto na figura 30, confere com os valores obtidos pelo Samuel e Bocutiu (2016). Os autores cronometraram o tempo necessário para calcular a sequência de Fibonacci de diversos valores, como visto na tabela 8 abaixo³⁶:

Fib(5)	1ms
Fib(10)	1ms
Fib(15)	1ms
Fib(20)	1ms
Fib(25)	2ms
Fib(30)	5ms
Fib(35)	54ms
Fib(40)	667ms
Fib(45)	6349ms
Fib(50)	69102ms

Tabela 8: Tempo relativo após invocar a função Fibonacci para vários valores

Desta forma, conclui-se que expressões lambdas são recomendáveis para obter-se códigos mais eficientes, objetivos, com melhor legibilidade e maior desempenho ao computar uma entrada de dados muito grandes, sejam estes vetores, um banco de dados ou Big Data e que devemos aproveitar estes recursos, sempre que possível, no código-fonte dos softwares criados.

36 Tabela retirada da obra Programming Kotlin de Samuel e Bocutiu (2016) , p. 160

CAPÍTULO 5 – CONCLUSÃO

Neste trabalho nota-se que o cálculo lambda é uma linguagem que, apesar da gramática bastante enxuta, pode-se fazer diversas operações, tais como representação de algarismos e operações matemáticas e booleanas e, inclusive, condicionais. Os estudos de Alonzo Church foram alicerce para que outros cientistas, na segunda metade do século XX, criarem um novo paradigma de programação que permitiu avanços na área da matemática e na área de inteligência artificial. Tal paradigma permite ao programador não ter preocupações com detalhes de implementações, o que permite que o código consiga ser assertivo e não-propenso a erros, uma vez que uma das bases da programação funcional é a imutabilidade, ou seja, dada a mesma entrada de dados, espera-se a mesma saída toda vez que uma função é executada.

Além disto, os recursos funcionais adotados em Java8 e Kotlin permitem uma manipulação e ordenar dados de forma bastante simples através das funções filter() e map(), o qual é possível extrair novas informações a partir de um conjunto de dados de entrada, sem que o programador se preocupe como isto deve ser feito, de uma forma mais natural, similar ao que é utilizado em banco de dados na linguagem SQL. A criação de componentes gráficos tornou-se bastante simples com um código bem mais conciso e quando lidamos com um conjunto muito grande de dados, através do paralelismo suportada pela Stream API do Java torna o algoritmo muito mais rápido em processamento de cálculos. Deste modo, mostra que para estruturas de dados como Arrays, listas e outros tipos de coleção em Java e Kotlin, o uso os recursos funcionais é benéfico no sentido de melhor performance e melhor legibilidade do código-fonte, permitindo que faça várias operações com poucas linhas de código.

A melhora de performance também é notável em funções recursivas, como visto através do recurso de memoization na linguagem Kotlin, onde ao aproveitar a cache, o cálculo da sequência recursiva de Fibonacci torna-se muito mais eficiente. Uma vez que os resultados já estarão armazenados na cache, evita-se que o cálculo de um valor numérico seja feito várias vezes. Ou seja, para cálculos recursivos de

números extensos, compensa a troca de espaço consumido em memória para um ganho de velocidade no cálculo do valor.

Há contudo vários outros aspectos da programação funcional a ser explorados, como o uso de *with* e *apply* em Kotlin, além de estudar a biblioteca funKtionale que adiciona a linguagem Kotlin outros recursos de programação funcional de forma natural. Além disto, há outras operações possíveis em cálculo lambda, como achar o antecessor de um número, subtração e divisão, comparação de número e até mesmo a representação de funções recursivas. Esta tarefa pode deixar a cargo caso surja algum outro interessado para prosseguir com a pesquisa sobre o assunto, mas, ainda sim, espero que através deste trabalho, tenha consigo cumprir a minha missão de apresentar o mundo da programação funcional e que o incentive a utilizá-lo nos programas a serem desenvolvidos.

REFERÊNCIAS

ALAMA, Jesse; KORBMACHER, Johannes. "The Lambda Calculus", The Stanford Encyclopedia of Philosophy (Summer 2018 Edition), Edward N. Zalta (ed.). Disponível em <<https://plato.stanford.edu/archives/sum2018/entries/lambdacalculus/>>. Acesso em 18/08/2018

CARDONE, Felice; HINDLEY, J. Roger. "History of Lambda-calculus and Combinatory Logic". 2006. Disponível em <<http://hope.simons-rock.edu/~pshields/cs/cmpt312/cardone-hindley.pdf>>. Acesso em 18/08/2018

HINDLEY, J.Roger; SELDIN, Jonathan P. "Lambda-Calculus and Combinators – an Introduction". 2^a ed. Cambridge University Press, 2008.

DA SILVA, Ricardo Dutra; "Aula 13: Lógica de Predicados", DAINF-UTFPR, 2014. Disponível em <http://www.dainf.ct.utfpr.edu.br/~rdutra/courses/2014-2/logica_aulas/aula13I.pdf>. Acesso em 19/08/2018

WU, Steinway. "Free Variables, Functions and Closures". Disponível em <<https://steinwaywu.ghost.io/free-variables-functions-and-closures/>> Acesso em 19/08/2018.

VARELA, Carlos. "Lambda Calculus alpha-renaming, beta reduction- applicative and normal evaluation orders, Church-Rosser theorem, combinators". Rensselaer Polytechnic Institute. 2010. Disponível em <<http://www.cs.rpi.edu/academics/courses/spring10/proglang/handouts/LambdaCalculus.pdf>>. Acesso em 20/08/2018

WANGENHEIM, Aldo von. "A semântica operacional do Cálculo Lambda". INE/CTC/UFSC. Disponivel em <<http://www.inf.ufsc.br/~aldo.vw/func/aula2.pdf>> Acesso em 20/08/2018.

_____ . “Cálculo lambda”. UFSC. Disponível em <<http://www.inf.ufsc.br/~j.barreto/PF/CalLambda.htm>> Acesso em 20/08/2018

_____ . “Natural Numbers as Church Numerals”. Department of Computer Science. University of North Carolina. Disponivel em <<http://www.cs.unc.edu/~stotts/723/Lambda/church.html>> Acesso em 21/08/2018

DE ARAUJO, Vitor. “A tese de Church-Turing”. UFMG. 2014. Disponível em <<http://homepages.dcc.ufmg.br/~nvieira/cursos/tl/a17s2/church-turing.pdf>>. Acesso em 21/08/2018

_____ . “Paradigma Funcional: Conceitos básicos”. UEM. Disponível em <<http://din.uem.br/ia/ferramentas/lisp/lisp3.htm>>. Acesso em 22/08/2018

SEBESTA, Robert W. “Conceitos de Linguagem de Programação”. 5^a ed. Bookman, 2006

SCHUBERT, Lenhart K.; “A Quick Introduction to Common Lisp”. University of Rochester, 2008. Disponível em <<https://www.cs.rochester.edu/~schubert/247-447/lisp-intro.pdf>>. Acessado em 23/08/2018

MEDEIROS, Adelardo; “Linguagem Lisp”. Disponível em <<https://www.dca.ufrn.br/~adelardo/lisp/>>. Acesso em 24/08/2018

LANDIN, P.J. “A Correspondence Between ALGOL 60 and Church’s Lambda-Notation: Part 1”. Communications of the ACM. 1965. Disponivel em <https://fi.ort.edu.uy/innovaportal/file/20124/1/22-landin_correspondence-between-algol-60-and-churchs-lambda-notation.pdf>. Acesso em 27/08/2018.

BONO, V., SALVO, I. “A CuCh Interpretation of an Object-Oriented Language”. Università degli Studi di Torino. 2001. Disponível em <https://ac.els-cdn.com/S1571066104001719/1-s2.0-S1571066104001719-main.pdf?_tid=8c0f0332-13ff-4e62-801f-f7095d670259&acdnat=1535582807_82e353ca458f1740e31068ef4dafcb4c>. Acesso em 29/08/2018

MACHADO, R. “Uma introdução ao cálculo lambda e a linguagens de programação funcionais”. UFRGS, 2013. Disponivel em <<http://www.inf.ufrgs.br/~rma/documentos/minicurso-weit2013.pdf>>. Acesso em 29/08/2018

DEITEL P., DEITEL H. “Java, Como programar – 10ª edição”. 10ª edição, Editora Pearson, 2016

_____. “JAVA SE versions history”. Disponivel em <<https://www.codejava.net/java-se/java-se-versions-history>>. Acesso em 31/08/2018.

_____. “A brief history of the Green Project”. Disponivel em <<https://tech-insider.org/java/research/1998/05-a.html>>. Acesso em 31/08/2018

GUIMARÃES, G. “A história da Linguagem Java”. Disponivel em <<https://tech-insider.org/java/research/1998/05-a.html>>. Acesso em 31/08/2018

JEMEROV, D., ISAKOVA, S.; “Kotlin in Action”. 11ª edição. Manning Publications. 2016

SAMUEL, S., BOCUTIU S.; “Programming Kotlin”. 1ª edoção. Packt Publishing. 2017

AQUILLES, A. “**Iteração Interna com Java 8 e métodos default**”. Disponível em <<https://alexandreaquiles.com.br/2014/03/24/iteracao-interna-com-jav...-default/>>. Acesso em 09/09/2018

LANHELLAS, R. “**Diferença entre ArrayList, Vector e LinkedList em Java**”. Disponível em <<https://www.devmedia.com.br/diferenca-entre-arraylist-vector-e-linkedlist-em-java/29162>>. Acesso em 09/09/2018

MEDEIROS, H. “**Padrão de Projeto Iterator em Java**”. Disponivel em <<https://www.devmedia.com.br/padrao-de-projeto-iterator-em-java/26733>>. Acesso em 09/09/2018

ARAUJO, C. “**Java Collections: Como utilizar Collections**”. Disponivel em <<https://www.devmedia.com.br/java-collections-como-utilizar-collections/18450>>. Acesso em 09/09/2018

LANGER, A. “**Java performance tutorial – How fast are the Java 8 streams?**” Disponivel em <<https://jaxenter.com/java-performance-tutorial-how-fast-are-the-jav...-streams-118830.html>>. Acesso em 09/09/2018.

SILVEIRA P., TURINI, R; “**Java 8 Prático: Lambdas, Streams e os novos recursos da linguagem**”. 1ª edição. Casa do Código, 2014.

_____. “**Class Arrays**”. Disponivel em <<https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html>>. Acesso em 14/09/2018.

COHEN, M. “**Genéricos em Java**”. PUCRS. 2013. Disponivel em <https://www.inf.pucrs.br/flash/alpro2/present/U03_projeto/01b-genericos/handout.html#gen%C3%A9ricos> Acesso em 14/09/2018.

SILVA, C. “**Java 8: Iniciando o desenvolvimento com a Streams API**”. 2016. Disponível em <<https://www.oracle.com/technetwork/pt/articles/javastreamsapi-javas-8-3410098-ptb.html>>. Acesso em 20/09/2018

_____. “**GUI Definition**”. The Linux Information Project. 2004. Disponível em <<http://www.linfo.org/gui.html>>. Acesso em 24/09/2018

ADAMS, C. “**Benefits of the Graphical User Interface**”. 2016. Disponível em <<https://www.thoughtco.com/benefits-of-graphical-user-interface-1206357>>. Acesso em 24/09/2018

CAELUM. “**Interfaces gráficas com Swing**”. Disponível em <<http://www.caelum.com.br/apostila-java-testes-xml-design-patterns/interfaces-graficas-com-swing/#5-1-interfaces-graficas-em-java>>. Acesso em 24/09/2018

GOLDMAN, K. “**CS102: The Java Event Model**”. 1999. Disponível em <<https://www.cse.wustl.edu/~kjg/cs102/Notes/EventModel/>>. Acesso em 24/09/2018

CHUAN, C.. “**Java Programming Tutorial – Programming Graphical User Interface (GUI)**”. Nanyang Technological University. 2018. Disponível em <https://www3.ntu.edu.sg/home/ehchua/programming/java/j4a_gui.html> Acesso em 24/09/2018

SCHIECK, R.; “**Threads: paralelizando tarefas com diferentes recursos do Java**”. Disponível em <<https://www.devmedia.com.br/threads-paralelizando-tarefas-com-os-diferentes-recursos-do-java/34309>>. Acesso em 28/09/2018

BOTTILHO, D.; “**Kotlin by examples: Methods and Lambdas**”. Medium. 2017. Disponível em <<https://medium.com/@dbottillo/kotlin-by-examples-methods-and-lambdas-25aef7544365>>. Acesso em 11/10/2018

CASTILLO, J.; “**Kotlin Functional Programming: Does it make sense?**”. Disponível em <<https://medium.com/@JorgeCastilloPr/kotlin-functional-programming-does-it-make-sense-36ad07e6bacf>>. Acesso em 14/10/2018

FERRAZ, R. M.; “**Conceitos de Programação: Lazy Evaluation**”. Disponível em <<http://logbr.reflectionsurface.com/2008/01/25/conceitos-de-programacao-lazy-evaluation/>>. Acesso em 14/10/2018

MANNINO, M.; “**Java 8 Lambda Limitations: Closures**”. Java Zone. 2017. Disponível em <<https://dzone.com/articles/java-8-lambdas-limitations-closures>>. Acesso em 14/10/2018

LORIMAN, R.J.; “**Currying and Partial Application in Kotlin**”. RealJenius. 2017. Disponível em <<https://realjenius.com/2017/08/24/kotlin-curry/>> Acesso em 17/10/2018

_____.; “**What’s a helper function?**”. WPI. Sem data. Disponível em <<https://web.cs.wpi.edu/~cs1101/a05/Docs/creating-helpers.html>>. Acesso em 18/10/2018

_____.; “**Does Kotlin support partial application?**”. Stack Overflow. Disponível em <<https://stackoverflow.com/questions/52711621/does-kotlin-support-partial-application>>. Acesso em 18/10/2018

_____. “**Tiope Index for October 2018**”. Disponível em <<https://www.tiobe.com/tiobe-index/>> Acesso em 22/10/2018

HEARN, M.; “**Functional programming in Kotlin, a new language from JetBrains**”. Medium. 2015. Disponível em <<https://blog.plan99.net/kotlin-fp-3bf63a17d64a>> Acesso em 18/10/2018

LIPSCHUTZ S., LIPSON, M.; “**Matemática discreta – Terceira edição**”. 3^a ed. Coleção Schaum. 2013.

VAREJÃO, F.; “**Linguagens de Programação Java, C e C++ e Outras**”. 1^a ed. Campus. 2004.

GUIMARÃES, A., LAGES, N. “**Introdução à Ciência da Computação**”. 1ª ed. LTC – Livros técnicos e Científicos Editora S.A. 1985

ROJAS, R.; “**A Tutorial Introduction to the Lambda Calculus**”. FU Berlin. 1997. Disponível em <<https://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf>>. Acesso em 24/10/2018

HUDAK, P.; “**A brief and informal introduction to the Lambda Calculus**”. Yale University. 2008. Disponível em <<http://www.cs.yale.edu/homes/hudak/CS201S08/lambda.pdf>>. Acesso em 24/10/2018

_____.; “**How to multiply in Lambda Calculus?**”. Disponível em <<https://math.stackexchange.com/questions/595518/how-to-multiply-in-lambda-calculus>>. Acesso em 24/10/2018.

HORWITZ, S.; “**Lambda Calculus (Part II)**”. University of Wisconsin-Madison. 2013. Disponível em <<http://pages.cs.wisc.edu/~horwitz/CS704-NOTES/2.LAMBDA-CALCULUS-PART2.html>>. Acesso em 25/10/2018

VERNA, P. “**Functional Programmers: Why we call them first-class citizens?**”. LinkedIn. 2016. Disponível em <<https://www.linkedin.com/pulse/functional-programmers-why-we-call-them-first-class-citizens-verma/>>. Acessos em 29/10/2018