PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Programa de Pós-Graduação em Engenharia de Software

Rodrigo Lazoti da Silva

VANTAGENS DE USO DA PROGRAMAÇÃO FUNCIONAL DENTRO DA PLATAFORMA JAVA

Belo Horizonte

2013

Rodrigo Lazoti da Silva

VANTAGENS DE USO DA PROGRAMAÇÃO FUNCIONAL DENTRO DA PLATAFORMA JAVA

Monografia apresentada ao Curso de Pós-Graduação em Engenharia de Software da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para obtenção do título de Especialista em Engenharia de Software.

Orientador: Prof. Carlos Barreto Ribas

Belo Horizonte

2013

Rodrigo Lazoti da Silva

VANTAGENS DE USO DA PROGRAMAÇÃO FUNCIONAL DENTRO DA PLATAFORMA JAVA

Monografia apresentada ao Curso de Pós- Graduação em Engenharia de Software da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para obtenção do título de Especialista em Engenharia de Software.

Prof. Carlos Barreto Ribas (Orientador) – PUC Minas

Prof. Humberto Torres Marques Neto – PUC Minas

Belo Horizonte, 10 de dezembro de 2013

À minha esposa Andressa pelo seu apoio incondicional, ao meu filho Rafael pela sua compreensão e paciência e à minha mãe por todo o seu apoio.

AGRADECIMENTOS

À todos que de alguma forma contribuíram para esta construção deste trabalho.

Aos grandes colegas e amigos do curso de Pós-Graduação em Engenharia de Software da Pontifícia Universidade Católica De Minas Gerais pela excelente amizade consolidada neste período.

Ao corpo docente do curso de Pós-Graduação em Engenharia de Software da Pontifícia Universidade Católica De Minas Gerais pelo apoio e pelas sugestões que muito me auxiliaram na elaboração deste trabalho.

Seu tempo é limitado, então não o desperdice vivendo a vida de outra pessoa. Não fique preso pelo dogma - que é viver pelos resultados do que outras pessoas pensam. Não deixe o ruído da opinião dos outros afogar a sua voz interior. E o mais importante, tenha a coragem de seguir seu coração e sua intuição. Eles de alguma forma já sabem o que você realmente quer se tornar. Tudo o mais é secundário.(JOBS, 2013)

RESUMO

Esta monografia realizou um experimento sobre o uso das linguagens de programação Clojure e Scala na máquina virtual Java. O objetivo foi analisar através de métricas o uso dessas linguagens que utilizam o paradigma da programação funcional na escrita de determinados tipos de algoritmos e que foram comparados com algoritmos escritos com a linguagem padrão da máquina virtual Java que é a linguagem de programação Java, e que faz uso do paradigma de programação orientado a objetos. A avaliação comparativa utilizou como métricas para comparação a quantidade de linhas de cada algoritmo, e o tempo em milissegundos de execução gasto por cada algoritmo.

Após análise dos dados coletados pelo experimento proposto nesse trabalho, os resultados mostram que utilizando linguagens de programação funcional como Clojure ou Scala na máquina virtual Java, consegue-se obter um melhor desempenho em algoritmos que precisam fazer cálculos ou iterar sobre listas de objetos e também de escrever menos código para se obter os resultados desejados.

Palavras-chave: Programação funcional, Programação Orientada a Objetos, Java, Scala, Clojure, máquina virtual Java.

ABSTRACT

This monograph conducted an experiment on the use of programming languages Clojure and Scala in the Java virtual machine. The objective was to analyze metrics through the use of these languages using the paradigm of functional programming in writing certain types of algorithms and that were compared to the algorithms written in the standard language of the Java virtual machine which is the Java programming language that it makes use of the paradigm of object-oriented programming. The benchmarking used as metrics to compare the number of lines of each algorithm and the execution time in milliseconds spent by each algorithm.

After analyzing the data collected by the experiment proposed in this work, the results show that using functional programming languages like Clojure or Scala in the Java virtual machine achieves better performance in algorithms that need doing calculations or iterate over lists of objects, and also to write less code to achieve the desired results.

Keywords: Functional Programming, Object Oriented Programming, Java, Scala, Clojure, Java Virtual Machine.

LISTA DE TABELAS

TABELA 1 – *Tempo (ms) de Processamento Gasto Por Cada Programa* 51

TABELA 2 – *Linhas de Código Utilizada Por Cada Programa* 52

LISTA DE GRÁFICOS

TABELA 1 – Tempo de processamento gasto por cada linguagem de programação . 52
TABELA 2 – Linhas de código por cada linguagem de programação 53

LISTA DE ABREVIATURAS E SIGLAS

EPFL – *École Polytechnique Fédérale de Lausanne*

FP – *Functional Programming - Programação Funcional*

JDK – *Java Developer Kit*

JVM – *Java Virtual Machine - Máquina Virtual Java*

Lisp – *List Processing - Processador de listas*

LoC – *Lines of Code - Linhas de Código*

OOP – *Object Oriented Programming - Programação Orientada a Objetos*

opcode – *Operation Code - Código de instrução*

TCO – *tail call optimization*

TI – *Tecnologia da Informação*

SUMÁRIO

1 INTRODUÇÃO **25**
1.1 Justificativa 25
1.2 Objetivos 25
1.2.1 Objetivo Geral 25
1.2.2 Objetivos Específicos 25
1.3 Problema 26
1.4 Metodologia 26
1.5 Organização do Documento 26

2 PARADIGMAS DE PROGRAMAÇÃO **28**
2.1 Programação Funcional 28
2.1.1 Cálculo Lambda 29
2.2 Programação Orientada a Objetos 30

3 PLATAFORMA JAVA **32**
3.1 História do Java 32
3.2 Máquina Virtual Java 33
3.3 Bytecode Java 34
3.4 Coletor de Lixo 34
3.5 Interoperabilidade Entre Linguagens na JVM 35

4 LINGUAGENS DE PROGRAMAÇÃO NA JVM **36**
4.1 Clojure 36
4.1.1 S-expressions 36
4.1.2 Funções de Ordem Superior 36
4.1.3 Transparência Referencial 37
4.1.4 Recursividade 37
4.2 Scala 38
4.2.1 Funções de ordem superior 39
4.2.2 Imutabilidade 39

4.2.3 Recursão 40
4.3 Java 40
4.3.1 Orientada a Objetos 41
4.3.2 Compilada e Interpretada 41
4.3.3 Herança 41
4.3.4 Encapsulamento 42
4.3.5 Polimorfismo 42

5 EXPERIMENTOS PROPOSTOS **44**
5.1 Somar Números Inteiros 44
5.1.1 Java 44
5.1.2 Clojure 44
5.1.3 Scala 45
5.2 Manipulação de Listas 46
5.2.1 Java 46
5.2.2 Clojure 46
5.2.3 Scala 47
5.3 Cálculo Fatorial 47
5.3.1 Java 47
5.3.2 Clojure 48
5.3.3 Scala 49

6 TESTES E RESULTADOS **50**
6.1 Compilando os Programas 50
6.2 Executando os Programas 50
6.3 Resultados 51
6.3.1 Tempo de Processamento 51
6.3.2 Linhas de Código 52

7 CONCLUSÃO **54**

REFERÊNCIAS **55**

1 INTRODUÇÃO

Desde o surgimento da máquina virtual Java, a linguagem de programação Java vem sendo utilizada para desenvolver sistemas para as mais diversas finalidades utilizando o paradigma de programação orientada a objetos, porém nos últimos anos surgiram outras linguagens de programação que conseguem gerar bytecode Java e que por sua vez podem ser executadas dentro da máquina virtual Java. Algumas dessas linguagens tem como base o paradigma da programação funcional, sendo que as que mais se destacam são as linguagens Clojure e Scala.

Um ponto interessante a se considerar entre as linguagens de programação Clojure e Scala e a própria linguagem de programação Java é que mesmo utilizando paradigmas de programação distintos ainda existe interoperabilidade entre elas e que com isso é possível compartilhar algoritmos e invocar códigos escritos entre estas linguagens de programação.

1.1 Justificativa

Embora a primeira vista pareça interessante utilizar em um mesmo projeto linguagens funcionais como Scala ou Clojure em conjunto com a linguagem Java, faz-se necessário um estudo para avaliar e descobrir em quais circunstâncias algoritmos escritos através da programação funcional possuem vantagem sobre algoritmos que produzem os mesmos resultados, mas que foram escritos através da programação orientada a objetos.

A principal justificativa para a realização desta pesquisa é demonstrar o uso da programação funcional dentro da máquina virtual Java através das linguagens Clojure e Scala, e as vantagens que pode-se obter ao optar pelo uso dessas linguagens.

1.2 Objetivos

1.2.1 Objetivo Geral

O principal objetivo desse trabalho é introduzir o paradigma da programação funcional, sua essência e a sua utilização dentro da plataforma Java, assim como demonstrar a viabilidade de uso do paradigma funcional na escrita de algoritmos e também expor sua simplicidade, melhor legibilidade e desempenho para determinadas tarefas.

1.2.2 Objetivos Específicos

- Contextualizar os princípios do paradigma da programação funcional e da programação orientada a objetos.

- Introduzir o uso da programação funcional dentro da plataforma Java por meio das linguagens Clojure e Scala;
- Introduzir o uso da interoperabilidade entre linguagens de programação dentro da plataforma Java;
- Como aplicar testes comparativos de desempenho e de como avaliar a legibilidade em algoritmos escritos em Clojure, Java e Scala;

1.3 Problema

Observa-se a necessidade do mercado pelo rápido desenvolvimento de aplicações e em conjunto uma grande dificuldade dos desenvolvedores para acompanhar essa velocidade na implementação de algoritmos que proporcionem ao mesmo tempo simplicidade na codificação e ganho de desempenho. Algumas linguagens de programação não oferecem uma flexibilidade na codificação de algoritmos, gerando muito mais código e consequentemente dificultando o seu entendimento e a sua facilidade de manutenção.

1.4 Metodologia

A modalidade de trabalho proposta para esta pesquisa é utilizar o formato de experimento, para assim conseguir realizar testes com a utilização de dois paradigmas de programação distintos e de mudanças em como algoritmos são escritos mesclando o uso de diferentes paradigmas de programação em uma única plataforma.

O método de pesquisa aplicado será uma avaliação comparativa entre dois paradigmas de programação.

O método de abordagem aplicado nesta pesquisa será uma abordagem indutiva e qualitativa, para que a pesquisa explore e apresente uma análise com base nos dados obtidos pela avaliação comparativa.

1.5 Organização do Documento

O primeiro capítulo contém uma breve introdução sobre esta monografia, a descrição das motivações, interesses e objetivos utilizados para compor esta monografia.

No segundo capítulo apresenta-se o que são paradigmas de programação e descreve-se também o paradigma de programação funcional e o paradigma de programação orientada a objetos.

O terceiro capítulo apresenta a máquina virtual Java, o bytecode Java e a interoperabilidade entre linguagens de programação na plataforma Java.

No quarto capítulo apresenta-se as linguagens de programação Clojure, Scala e Java.

O quinto capítulo apresenta os programas utilizados nos experimentos.

O sexto capítulo apresenta a execução dos programas apresentados no capítulo anterior, assim como os resultados obtidos por eles.

O sétimo capítulo apresenta a conclusão obtida através do experimento realizado nesse trabalho.

2 PARADIGMAS DE PROGRAMAÇÃO

"Paradigma é um termo com origem no grego ̈paradeigmaq̈ue significa modelo, padrão. No sentido lato corresponde a algo que vai servir de modelo ou exemplo a ser seguido em determinada situação. São as normas orientadoras de um grupo que estabelecem limites e que determinam como um indivíduo deve agir dentro desses limites." (SIGNIFICADOS, 2013).

Paradigma de programação pode ser definido como um uma idéia que o desenvolvedor possui ao estruturar e escrever um programa. Metodologias diferentes são propostas através de diferentes linguagens de programação, sendo que estas apresentam diferentes paradigmas. Um paradigma de programação está relacionado à maneira de pensar do desenvolvedor e na forma em que ele escreve uma solução para os problemas. É através do paradigma que permite-se ou proíbe-se o uso de determinadas técnicas de programação.

Paradigmas de programação são classificados com base em seus conceitos que podem ser: estruturado, funcional, imperativo, lógico e orientado a objetos. Cada paradigma define uma forma particular de tratar problemas e de definir possíveis soluções. Fora isso, uma linguagem de programação pode fazer uso de mais de um paradigma para melhorar as análises e soluções.

2.1 Programação Funcional

Segundo Fogus "Programação funcional é o uso de funções que transformam valores em unidades de abstração, subsequentemente usadas para construir sistemas de software."(FOGUS, 2013, tradução nossa).[1]

O Paradigma de programação funcional basicamente cria algoritmos escritos em linguagem definida por expressões, declarações e funções, considerando a computação como uma avaliação de funções matemáticas. Com o uso da *Functional Programming - Programação Funcional* (FP) existe também a possibilidade de escrever algoritmos onde podemos determinar o que se pretende criar e não como será criado. O paradigma funcional não utiliza o conceito de atribuição pelo fato de que os programas são formados por definições de funções. Uma de suas características determinantes é a aplicação de funções à dados imutáveis e sem estados.

Existem inúmeras características que cada linguagem funcional utiliza, mas geralmente as linguagens de programação funcionais tem as seguintes características:

- Funções de ordem superior;

[1] Functional programming is the use of functions that transform values into units of abstraction, subsequently used to build software systems.

- Transparência referencial;
- Recursão;
- Imutabilidade;

Na área da matemática e da ciência da computação, funções de ordem superior são funções que podem receber funções como argumentos, assim como produzir funções como resultado de sua computação. Por ser um modelo simples, o cálculo lambda permite demonstrar alguns conceitos importantes de linguagens de programação, como por exemplo ligação, escopo, ordem de avaliação, computabilidade, sistemas de tipos e etc.(WAMPLER, 2011).

Na matemática, funções nunca possuem efeitos colaterais. Por exemplo, não importa o que aconteça internamente em um método chamado $cos(x)$ que calcule o co-seno de x, o resultado será sempre o mesmo para um dado valor de x, sem nenhuma mudança no estado externo do programa que invocou este método. Por exemplo, dado a função $y = fx$ e a função $g = hyy$, é possível substituir a definição de g pela função $g = h(fx)(fx)$ e obter o mesmo resultado. Estar apto a substituir a chamada de uma função por um conjunto de parâmetros com o valor esperado é chamada de transparência referencial e torna possível conduzir o raciocínio equacional no código.(HASKELL, 2013).

Recursão é um recurso amplamente utilizado na programação funcional como a principal forma de iteração. Linguagens de programação funcionais frequentemente irão oferecer otimizações para garantir que a execuções de recursões grandes não consumam muita memória sem necessidade.(HASKELL, 2013).

Programas funcionais puros tipicamente operam através de dados imutáveis, dados que não possuem mudança de valor. Ao invés de alterar valores existentes dos dados, são criadas cópias alteradas e os valores originais são mantidos. Desde que a estrutura desses dados não possam ser modificadas, eles podem frequentemente ser compartilhados entre cópias novas e velhas armazenadas em memória.

2.1.1 Cálculo Lambda

O calculo lambda também conhecido pelo símbolo λ, foi inventado por Alonzo Chruch em 1930 e publicado em 1941 em resposta ao problema de decisão de David Hilbert (Hilbert's Entscheidungsproblem) proposto no ano de 1928. O problema de decisão inspirou outro modelo computacional conhecido como máquina de Turing.(ZEROTURNAROUND, 2013)

O calculo lambda é um dos pilares da ciência da computação e pode ser considerado a primeira linguagem de programação funcional, embora nunca tenha sido projetada para ser realmente executada em um computador. Ele talvez seja o mais famoso, por servir

como base para a linguagem de programação LISP, inventada por John McCarthy em 1958.(ZEROTURNAROUND, 2013)

O cálculo lambda é um modelo matemático, e pode ser pensado como uma linguagem de programação pura, baseada na definição e aplicação de funções, e o seu método de iteração é através da recursão. Esse modelo permite a representação de qualquer algoritmo, e é pura no sentido de que as funções recebem e retornam dados, que podem ser inclusive funções, e não podem ser alterados pela função.

Funções em cálculo lambda são muito diferentes de funções em linguagens de programação imperativas como Java e C. Em uma linguagem de programação imperativa a avaliação de uma função pode ter efeitos colaterais, afetando avaliações futuras de uma função para outras funções. No cálculo lambda uma função não pode retornar um resultado baseado em seus parâmetros, as invés disso seus parâmetros são reduzidos para chegar em um resultado que matematicamente é equivalente a questão.(MICHAELSON, 1989).

O conceito central em cálculo lambda são as expressões ou termos, sendo que existem três tipos de expressões:

- Variável: x;
- Abstração ou Função: λx.e;
- Aplicação: x y;

Variáveis são expressas por meio de identificadores alfanuméricos, com por exemplo a letra *x*. Abstrações representam a função que retorna o valor *e* quando recebe o parâmetro formal *x*, sendo que o . na expressão separa o argumento da função de seu corpo. Aplicações representam a aplicação da expressão *x* para *y*.

2.2 Programação Orientada a Objetos

Paradigma de orientação a objetos é um conceito que foi criado devido a necessidade de ultrapassar os problemas que foram encontrados com o uso de técnicas como a programação estruturada. Enquanto a programação estruturada tem colocado ênfase na lógica e ações, a programação orientada a objetos tomou uma direção completamente diferente colocando ênfase em objetos e informações. Com a programação orientada a objetos, um problema vai ser dividido em várias unidades que são chamados de objetos.

Existem vantagens ao utilizar a programação orientada a objetos como a manutenção simplificada, uma análise avançada de programas complexos e reutilização de código. Existe uma série de linguagens de programação que usam *Object Oriented Programming - Programação Orientada a Objetos* (OOP), como o Java, C++ e Smalltalk. Um con-

ceito importante na OOP é a modelagem de dados, pois antes de construir um sistema orientado a objetos, é necessário encontrar os objetos dentro do sistema e determinar as relações entre eles.

Uma classe é uma unidade que armazena dados e funções que irão realizar operações sobre os dados. Um objeto é uma instância de classe e ele pode receber e enviar mensagens para outros objetos. Os objetos que existem dentro de programas são muitas vezes baseados em objetos do mundo real, e que irão se comportar da mesma maneira.

Existem duas coisas que são encontradas em todos os objetos que existem no mundo real, essas duas coisas são comportamentos e estados. Enquanto comportamentos e estados são encontrados em objetos do mundo real, eles também podem ser codificados para objetos no programa.

As principais características das linguagens de programação orientadas a objetos são:

- Herança;
- Encapsulamento;
- Polimorfismo;

Herança é um aspecto da OOP que permite que subclasses possam herdar os traços e as características de sua superclasse. A subclasse herdará todos os membros, exceto aqueles que foram definidos como sendo privado. Uma subclasse pode usar o comportamento dos membros que herdou da superclasse e que também pode adicionar novos membros e comportamentos. Existem duas grandes vantagens no uso de herança que são a implementação de dados abstratos e reutilização de comportamentos.

Encapsulamento é responsável por proteger os dados dentro de uma classe de objetos externos. Ele só irá revelar a informação funcional. No entanto, a implementação será ocultada. O encapsulamento é um conceito que promove a modularidade e é também crucial para esconder informações que não precisam ser expostas para outros objetos.

Polimorfismo é a capacidade de diferentes objetos para responder à mesma mensagem de diferentes maneiras. Ele permite que um único nome ou o operador a ser associada com diferentes operações, dependendo do tipo de dados que tenha passado, e dá a possibilidade de redefinição de um método dentro de uma classe de derivados. (JR., 2010)

3 PLATAFORMA JAVA

3.1 História do Java

Com o Java, a Sun Microsystems criou a primeira linguagem de programação que não estava vinculada a nenhum sistema operacional específico ou microprocessador. As aplicações escritas em Java podiam ser executadas em qualquer lugar, eliminando um dos maiores problemas para os usuários de computador, a incompatibilidade entre sistemas operacionais e versões de sistemas operacionais.

O Java foi desenvolvido a partir de um desejo de construir software para produtos eletrônicos de consumo como aparelhos eletrônicos e eletrodomésticos. Tudo começou em 1991, quando uma equipe de pesquisadores da Sun desenvolveu alguns conceitos dando uma nova direção para alta tecnologia, os consumidores que necessitavam de computadores estavam por toda a parte e foram a força motriz por trás de muitos dos produtos voltados para a casa como o videocassete, o forno de micro-ondas e o sistema de som. No entanto, cada produto necessitava da sua própria interface. Em outras palavras, para controlar três dispositivos, os consumidores tiveram que ter três controles remotos e compreender o manuseio para os três dispositivos. Além do fato de que a Sun estava ficando para trás de seus concorrentes, este foi um forte motivador para a Sun lançar um novo projeto, que mais tarde se tornaria o Java.(ORACLE, 2013b).

Uma equipe chamada Green Team foi formada para trabalhar na criação de um dispositivo simples que controlava uma variedade de produtos eletrônicos de uso para o dia-a-dia. A equipe foi composta por dois programadores, Patrick Naughton e James Gosling, e o engenheiro Mike Sheridan. Gosling percebeu que o que eles precisavam era de uma nova linguagem de programação.(ORACLE, 2013a).

Até o momento, as linguagens de programação existentes como o C++ tinha sua ênfase na velocidade, e não na confiabilidade. No setor de eletroeletrônicos, a confiabilidade é mais importante que a velocidade. Com este propósito Gosling e Naughton conseguiram realizar o trabalho em conjunto e criar uma nova linguagem que eles chamaram de Oak e isso aconteceu em agosto de 1991. Um ano depois, o Green Team desenvolveu um dispositivo portátil, sem teclado, sem botões e com uma tela minúscula. Bastava um toque para ativá-lo e controlar a ação na tela com a ponta do dedo, e com esse dispositivo tornou-se possível programar o gravador de videocassete apenas movendo o dedo ao longo da tela. Ainda assim, esta tecnologia não emplacou devido a vários motivos como a fabricação dos chips que eram muito caros.(HORSTMANN; CORNELL, 2010).

O nome Oak teve de ser alterado devido ao fato de que era muito próximo ao de uma linguagem de programação já existente, consequentemente Oak foi rebatizado

como Java. Em 1994, a maioria das pessoas utilizam o Mosaic, um navegador Web não-comercial e em meados de 1994 os desenvolvedores da Sun viram uma oportunidade para a linguagem Java com o surgimento da World Wide Web. Sua idéia era liberar o Java de graça na Internet liberando também o seu próprio navegador para uso não comercial, para que assim eles se tornassem um padrão.(HORSTMANN; CORNELL, 2010).

3.2 Máquina Virtual Java

A máquina virtual Java, conhecida também pela sigla *Java Virtual Machine - Máquina Virtual Java* (JVM) que é um acrônimo para Java Virtual Machine, é o principal componente da plataforma Java. Ela é a tecnologia responsável por tornar programas escritos em Java independentes de sistemas operacionais e de hardware. A JVM é chamada de "virtual", pois fornece uma interface que não depende de sistema operacional subjacente ou da arquitetura de hardware da máquina. Esta independência de hardware e de sistema operacional é a base do termo "Write Once, Run Anywhere"que traduzido significa "escreve uma vez, execute em qualquer lugar"(LINDHOLM et al., 2013). Existem versões da JVM para diversas arquiteturas e sistemas operacionais.

A maioria das linguagens de programação compilam o código fonte diretamente para código de máquina, que é projetado para ser executado em uma arquitetura de microprocessador específico ou sistema operacional, como o Windows ou Unix. No caso da JVM, ela permite que o bytecode Java possa ser executado como ações ou chamadas de sistema operacional em qualquer processador, independentemente do sistema operacional. Portanto, a JVM não tem conhecimento sobre a linguagem Java, porque ela entende somente o bytecode. Desta forma, qualquer linguagem que possar ser compilada e ser capaz gerar bytecode Java poderá ser executada na JVM.

Algumas das características mais importantes da JVM são:

- Baseia-se numa pilha de avaliação, o qual pode ser manipulada pelos bytecodes Java. Os argumentos do método são enviados para a pilha antes da invocação de um método, e quando completado o valor de retorno está localizado na pilha.
- É fortemente tipada.
- Manipulação de ponteiro não é permitido.
- Possui coleta automática de lixo, onde os objetos não referenciados são automaticamente liberados da memória.

3.3 Bytecode Java

Arquivos *.class* são o resultado da compilação de programas escritos com a linguagem Java. Os arquivos *.class* contém bytecode Java, sendo que bytecode Java são instruções que a máquina virtual Java consegue executar, cada instrução ou bytecode a ser executado é um código de operação (*Operation Code - Código de instrução* (opcode)) com o tamanho de um byte, seguido por zero ou mais operandos fornecendo argumentos ou dados que são usados na operação. Atualmente existem 256 opcodes possíveis, embora nem todos estejam sendo utilizados e mais 51 opcodes estão reservados para uso futuro.(LINDHOLM et al., 2013). Muitas instruções não tem operandos e consiste apenas de um único opcode.

Segundo Lindholm et al. (2013), essas instruções são agrupadas da seguinte forma:

- Operações para checagem de tipos;
- Operações de carga e armazenamento;
- Operações aritméticas;
- Operações para conversão de tipos;
- Operações para criação e manipulação de objetos;
- Operações para gerenciamento de pilha;
- Operações para controle de fluxo;
- Operações para invocação de método;
- Operações lançamento de exceções;
- Operações para sincronização;

Um programador Java não precisa entender os bytecodes Java para ser proficiente na linguagem, da mesma forma que um programador de qualquer linguagem de alto nível compilada para linguagem de máquina não precisa conhecer a linguagem de montagem do computador hospedeiro para escrever bons programas naquela linguagem.

3.4 Coletor de Lixo

A máquina virtual Java possui coleta de lixo automática, ou seja, caso não exista mais nenhuma referência a um objeto que tenha sido criado, o coletor de lixo destrói o objeto e libera a memória ocupada por ele. Quando a JVM percebe que o sistema diminuiu a utilização do processador, a JVM faz com que o coletor de lixo execute, vasculhando a memória em busca de algum objeto criado e não mais referenciado.

Diferente de outras linguagens de programação, em Java não é possível liberar explicitamente a memória de objetos. Embora a máquina virtual fornece dois métodos que podem (não exista garantia que isso irá ocorrer) executar de forma instantânea o

coleta de lixo. Os métodos para isso são *Runtime.gc*() e *System.gc*().

O coletor de lixo do Java é uma grande vantagem para desalocação de memória, que é um grande inconveniente para programadores que trabalham com ponteiros e necessitam liberar o espaço alocado, visto que é o próprio sistema que se encarrega desta limpeza, evitando erros de desalocação de objetos ainda em uso.

3.5 Interoperabilidade Entre Linguagens na JVM

Interoperabilidade é a capacidade que componentes dentro de uma infraestrutura de *Tecnologia da Informação* (TI) tem de conversar entre si. Assim, com interoperabilidade garante-se que aplicações possam conversar de forma a trocar e processar dados geridos por outras aplicações. Seguir um padrão de intercâmbio de informações é fundamental para que se consiga atingir interoperabilidade. O uso de padrões abertos e públicos permite que diversos fabricantes possam fornecer formas de acesso para outras aplicações usando protocolos e formatos de arquivos padronizados.

Interoperabilidade entre linguagens é a capacidade de duas linguagens de programação diferentes interagirem nativamente e operarem a mesma estrutura de dados. Na plataforma Java, existem compiladores para outras linguagens de programação que geram bytecode Java e por isso podem ser executados na máquina virtual Java. Algumas das diversas linguagens de programação que geram bytecode são:

- Clojure - um dialeto *List Processing - Processador de listas* (Lisp);
- Groovy - uma linguagem de script;
- JRuby - uma implementação da linguagem Ruby;
- Jython - uma implementação da linguagem Python;
- Rhino - uma implementação da linguagem Javascript;
- Scala - uma linguagem híbrida que utiliza os conceitos de programação orientada a objetos e programação funcional;

4 LINGUAGENS DE PROGRAMAÇÃO NA JVM

4.1 Clojure

Clojure é uma linguagem de programação criada por Rich Hickey e classificada como um dialeto Lisp que funciona na JVM. Clojure utiliza o paradigma de programação funcional e ela é construída em S-expressions que são uma notação para listas de dados aninhadas. Lisp é uma das mais velhas linguagens de programação, criada por John McCarthy in 1958. Além de Clojure, muitos outros dialetos do Lisp foram criados, como Scheme e Common Lisp.

Clojure é uma linguagem dinâmica assim como o Lisp. Isto significa que muitas coisas são determinadas em tempo de execução de um programa e não quando o código-fonte do programa vai ser compilado. Isso permite que programas sejam escritos de forma que não seria possível em linguagens estáticas.(RATHORE, 2012).

Devido a sua interoperabilidade com o Java, Clojure pode utilizar qualquer biblioteca Java, assim como bibliotecas escritas em Clojure podem ser usadas no Java. Uma aplicação Clojure pode ser compilada e ser executada como uma aplicação Java comum.

Clojure é classificada como uma linguagem de programação funcional e um exemplo disso está na forma de tratar todas as estruturas de dados como imutáveis, além disso Clojure incentiva o uso de funções de ordem superior. Suas principais estruturas de dados utilizam uma técnica chamada laziness (preguiça), que significa que sua execução acontece somente quando necessário. Um exemplo de uso para laziness é a capacidade de definir e utilizar sequências infinitas.(RATHORE, 2012).

4.1.1 S-expressions

S-expression ou expressão simbólica são estruturas de dados baseadas em listas que representam dados semi-estruturados, seu uso se tornou mais comum devido ao seu uso na família de linguagens de programação baseadas no Lisp. Uma S-expression pode ser uma lista quem contém outras S-expressions. Eles normalmente são representados com texto entre parênteses, sequências de caracteres separadas por espaços brancos, como por exemplo em $(= 3(1 + 2))$, que representa a expressão booleana normalmente escrita em Java como $1 + 2 == 3$.

4.1.2 Funções de Ordem Superior

Funções escritas em Clojure são funções de primeira classe e isso significa que funções podem ser passadas em forma de parâmetro para outras funções, podem ser

criadas dinamicamente e podem ser usadas como retorno de outras funções. Em Clojure funções são como estruturas de dados, números ou um conjunto de caracteres (strings).

Dada as funções *soma* e *calcular* a seguir:

```
1 (defn soma [a b] (+ a b))
2 (defn calcular [fn a b]  (format "Resultado: \%d" (fn a b)))
```

Exemplo de função de ordem superior passando a função *soma* como argumento para a função *calcular*:

```
1 (calcular soma 10 15)
2 "Resultado: 25"
```

4.1.3 Transparência Referencial

Funções referencialmente transparentes sempre retorna o mesmo resultado quando chamadas com o mesmo argumento. A fim de alcançar este objectivo, elas só dependem de seus próprios argumentos e de valores imutáveis para determinar o seu valor de retorno.

Ao utilizar uma função referencialmente transparente, nunca é necessário considerar quais as possíveis condições externas podem afetar o valor de retorno da função. Isto é especialmente importante se a função é usada em vários lugares ou se está aninhada em uma cadeia de chamadas de função. Por exemplo, dada a função a seguir, não importa quantas vezes ela seja invocada com os mesmos parâmetros, a função sempre vai retornar o mesmo valor.

```
1  (defn somar [numeros] (reduce + numeros))
2
3  (somar [1 2 3])
4  6
5
6  (somar [1 2 3])
7  6
8
9  (somar [1 2 3])
10 6
```

4.1.4 Recursividade

Segundo Halloway e Bedra (2012), Clojure realiza um ótimo trabalho em unir o poder da programação funcional com a realidade da JVM, e um exemplo disso é o uso de *tail call optimization* (TCO) explícitos através de *loop/recur*. Uma das características

em Clojure que um programador acostumado com OOP pode sentir falta é a ausência de laços para realizar tarefas repetitivas como percorrer todos os elementos de uma lista.

A seguir um exemplo de como criar uma função para calcular o valor de x elevado à y para demonstrar o uso de recursividade em Clojure.

```
1  (defn exponencial
2    ([x y]
3      (if (= y 0)
4        1
5        (* x (exponencial x (- y 1))))))
6
7  (exponencial 2 3)
8  8
```

4.2 Scala

Scala é uma abreviação para Scalable Language e é uma linguagem de programação de tipagem estática classificada como uma linguagem híbrida ou multi-paradigma que incorpora algumas recursos da programação orientada a objetos e outros recursos da programação funcional.

O desenvolvimento da linguagem Scala foi iniciada por Martin Odersky em 2001 e teve a primeira versão liberada para o uso em 2003. Martin é professor na Escola de Ciências da Computação e Comunicação na *École Polytechnique Fédérale de Lausanne* (EPFL). Ele passou seus anos de pós-graduação trabalhando no grupo liderado por Niklaus Wirth. Martin trabalhou na linguagem Pizza, uma linguagem funcional precoce na JVM. Mais tarde, ele trabalhou no GJ, um protótipo do que mais tarde se tornou genéricos em Java, com Philip Wadler. Martin foi contratado pela Sun Microsystems para produzir a implementação de referência do *javac*, o compilador Java que é distribuído com o *Java Developer Kit* (JDK).(TATE, 2010).

Scala é executada na JVM e interopera de forma integrada com todas as bibliotecas Java, assim como programas escritos em Scala também podem ser utilizados por programas ou bibliotecas Java.

Scala estende o sistema de tipos do Java com objetos genéricos mais flexíveis. A inferência de tipo utilizada pela linguagem ajuda de forma automática na assinatura de tipos, para que o programador não tenha que fornecer informações sobre tipagem manualmente.

Segundo Subramaniam (2009), as principais características dessa linguagem são:

- Possui suporte a modelo de concorrência baseado em eventos;
- Possui suporte ao estilo de programação imperativa e funcional;
- É puramente orientada à objetos;
- Possui ótima interoperabilidade com Java;
- Impõe tipagem estática;
- É concisa e expressiva;
- É altamente escalável, e isso significa escrever menos código para criar programar com grande desempenho;

4.2.1 Funções de ordem superior

Em Scala é possível passar funções anônimas para uma outra função. Funções que podem receber outras funções como parâmetros são chamadas de funções de ordem superior (high-order function). Na matemática, dois exemplos de funções de ordem superior são as derivadas e integral.

Dada as funções *soma* e *calcular* a seguir:

```
1  def soma(a: Int, b: Int) = a + b
2  def calcular(funcao: (Int, Int) ⇒ Int, a: Int, b: Int) = "
       Resultado: " + funcao(a, b)
```

Exemplo de função de ordem superior passando a função *soma* como argumento para a função *calcular*:

```
1  calcular(soma, 10, 20)
2  "Resultado: 30"
```

4.2.2 Imutabilidade

Imutabilidade é outra consequência da matemática. Na expressão matemática $y = sin(x)$, uma vez que se saiba o valor de x, o resultado de y será sempre o mesmo. Como outro exemplo, se for feito a soma dos números inteiros 3 e 4, o valor resultante 7 desse cálculo é novo número e não um número modificado.(WAMPLER; PAYNE, 2009).

Em Scala, quando o valor de uma variável é definido, este não pode mais ser alterado. Para demonstrar, este é um exemplo de como criar uma variável em Scala:

```
1  class Teste {
2    val resultado = 3 + 4
3    resultado = resultado + 1
4  }
```

Quando essa classe é compilada, o compilador Scala informa o seguinte erro de compilação:

```
1  scalac Teste.scala
2  Teste.scala:3: error: reassignment to val
3    resultado = resultado + 1
4                ^
5    one error found
```

Segundo Wampler e Payne (2009), a imutabilidade tem enormes benefícios para a concorrência. Quase toda a dificuldade de programação concorrente está em sincronizar o acesso a dados compartilhados, também conhecido como estado mutável. Se remover a mutabilidade de um programa concorrente, então os problemas com estado mutável não irão existir. É a combinação de funções referencialmente transparentes e valores imutáveis que compõem a programação funcional como uma melhor forma de escrever software concorrente.

4.2.3 Recursão

Recursão desempenha um papel mais importante na programação funcional pura do que na programação imperativa, em parte por causa da restrição de que as variáveis precisam ser imutáveis. Uma forma de implementar *loop* de uma forma puramente funcional é com recursividade.

A seguir um exemplo de como criar uma função para calcular o valor de x elevado à y para demonstrar o uso de recursividade em Scala.

```
1  def exponencial(x: Int, y: Int): Int = if (y == 0) 1 else x *
     exponencial(x, (y - 1))
2
3  val resultado = exponencial(2, 3)
4  println(resultado) //imprime 8
```

4.3 Java

Embora Java não seja a única linguagem de programação disponível para a plataforma Java, ela é a linguagem de programação padrão. A linguagem Java foi criada na década de 90 na empresa Sun Microsystem por um equipe de programadores liderada por James Gosling. Java é uma linguagem de programação orientada a objetos, e isso significa que o foco da linguagem é sobre os dados que representam estados do objeto e dos métodos que servem para manipular os dados e alterar o estados dos objetos.

4.3.1 Orientada a Objetos

O Paradigma de orientação a objetos permite que o programador foque o desenvolvimento no dado, ou no objeto. Java não é uma linguagem puramente orientada a objetos como Smaltalk, onde qualquer elemento é um objeto. Em Java há os tipos primitivos de dados que não são objetos, mas foram criados e incorporados ao Java para permitir uma melhor forma de utilização da linguagem pelos programadores. Outra característica importante da linguagem Java em relação à linguagem C++, é que Java não suporta herança múltipla.(CLARO; SOBRAL, 2008).

4.3.2 Compilada e Interpretada

Um programa desenvolvido em Java necessita ser compilado, gerando um bytecode. Para executá-lo é necessário então, que um interpretador leia o código binário, o bytecode e repasse as instruções ao processador da máquina específica. Esse interpretador é conhecido como JVM (Java Virtual Machine). Os bytecodes são conjuntos de instruções, parecidas com código de máquina. É um formato próprio do Java para a representação das instruções no código compilado.(CLARO; SOBRAL, 2008).

4.3.3 Herança

Ao programar classes em Java, muitas vezes diferentes classes tem características comuns entre elas, por este motivo, ao invés de se criar um nova classes com todas essas características, usa-se as características de uma classe já existente através da herança.

A linguagem de programação Java suporta apenas herança simples, que significa que uma classe pode possuir apenas uma superclasse diretamente. A palavra-chave ***extends*** é utilizada para indicar que uma classe herda de outra classe.

Como exemplo, dada uma classe chamada *Veiculo* que possui um método *buzinar*() e duas classes *Carro* e *Moto*. As classes *Carro* e *Moto* herdam o método *buzinar*() da classe *Veiculo*, e automaticamente se tornar subclasses da superclasse *Veiculo*.

```
1     public class Veiculo {
2       public String buzinar() {
3         return "BiBiBi";
4       }
5     }
6
7     public class Carro extends Veiculo { }
8
9     public class Moto extends Veiculo { }
```

4.3.4 Encapsulamento

Encapsulamento é utilizado para não expor detalhes internos do objeto, tornando o objeto mais independente. Em outras palavras, utilizando o encapsulamento é possível separar os aspectos externos de um objeto e torná-los acessíveis para outros objetos, enquanto os detalhes internos desse objeto permanecem ocultos para os outros objetos.

Uma grande vantagem de uso do encapsulamento é permitir que a implementação de um objeto possa sofrer modificações sem que os outros objetos ou aplicações que fazem uso desse objeto sejam afetados. Isso ajuda na manutenção dos programas, pois uma mudança em um objeto não afeta outros objetos de uma aplicação.

O uso do encapsulamento também evita que dados específicos de uma classe possam ser acessados ou usados diretamente. Em Java esse controle é feito por modificadores de acesso que restringem ou não o acesso à métodos e atributos de uma classe.

Como exemplo, temos a classe *Pessoa* que possui um atributo marcado como privado através da palavra chave *private* e com esse modificador significa que nenhum outro objeto tem acesso à ele). Os métodos *getNome* e *setNome* utilizar o modificador de acesso *public*, que significa que esses métodos são públicos, ou seja, qualquer objetos pode invocar estes métodos.

```
1   public class Pessoa {
2     private String nome;
3
4     public void setNome(String nome) {
5       this.nome = nome;
6     }
7
8     public String getNome() {
9       return nome;
10    }
11  }
```

4.3.5 Polimorfismo

Polimorfismo é o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação ou assinatura mas possuem comportamentos distintos, especializados para cada classe derivada, usando para tanto uma referência a um objeto do tipo da superclasse. A decisão sobre qual o método que deve ser selecionado, de acordo com o tipo da classe derivada, é tomada em tempo de execução, através do mecanismo de ligação tardia.(RICARTE, 2013)

O Polimorfismo pode ocorrer de forma estática ou dinâmica. Polimorfismo estático ocorre quando existe um mesmo método implementado várias vezes na mesma classe. A escolha de qual método será invocado depende da assinatura dos métodos sobrecarregados. O Polimorfismo dinâmico acontece na herança, quando a subclasse sobrepõe o método original. Agora o método escolhido se dá em tempo de execução e não mais em tempo de compilação. A escolha de qual método será chamado depende do tipo do objeto que recebe a mensagem.

No exemplo a seguir, foi definido uma interface chamada *Animal* com um método chamado *fazerBarulho*() que representa qualquer tipo de animal que possa emitir algum tipo de som. As classes *Cachorro* e *Gato* são classes derivadas dessa interface e cada classe possui sua própria implementação do método *fazerBarulho* conforme cada tipo de animal.

```
1   public interface Animal {
2     abstract String fazerBarulho();
3   }
4
5   public class Cachorro implements Animal {
6     public String fazerBarulho() {
7       return "Au au Au";
8     }
9   }
10
11  public class Gato implements Animal {
12    public String fazerBarulho() {
13      return "Miau Miau";
14    }
15  }
16
17  Animal animal = new Cachorro();
18  animal.fazerBarulho(); //retorna "Au au Au"
```

5 EXPERIMENTOS PROPOSTOS

Neste capítulo, são apresentados os programas utilizados no experimento, sendo estes escritos nas linguagens de programação Java, Clojure e Scala seguindo esta mesma sequência. Todos os programas serão seguidos de uma breve explicação e comentário. Todos os programas geram o tempo de execução em milissegundos.

5.1 Somar Números Inteiros

Este primeiro programa tem por finalidade somar os primeiros 1000000 números inteiros.

5.1.1 Java

Implementação em Java:

```
1  public class SomarNumeros {
2
3    public static void main(String[] args) {
4      double tempoInicial = System.nanoTime();
5
6      long somatoria = 0;
7      for (long i = 1; i < 1000000; i++) {
8        somatoria += i;
9      }
10
11     double tempoFinal = System.nanoTime();
12
13     System.out.println("Resultado: " + somatoria);
14     System.out.println("Tempo (ms): " + ((tempoFinal -
           tempoInicial) / 1000000));
15   }
16
17 }
```

5.1.2 Clojure

Implementação em Clojure:

```
1  (ns somarnumeros.core
2    (:gen-class))
```

```
3
4       (defn -main
5         [args]
6
7         (def tempoInicial (double (. System (nanoTime))))
8
9         (def resultado (loop [somar (long 0) inicio (long 1)]
10          (if (< inicio 1000000) (recur (+ somar inicio) (inc inicio))
              somar)))
11
12        (def tempoFinal (double (. System (nanoTime))))
13
14        (println (format "Resultado: %d" resultado))
15        (println (format "Tempo (ms): %.6f" (/ (- tempoFinal
            tempoInicial) 1000000))))
```

5.1.3 Scala

Implementação em Scala:

```
1       object SomarNumeros extends App {
2
3         val tempoInicial: Double = System.nanoTime
4
5         def somar(inicio: Long, total: Long, resultado: Long = 0): Long
              =
6           if (inicio >= total) resultado else somar(inicio + 1, total,
              inicio + resultado)
7
8         val resultado = somar(1, 1000000)
9         val tempoFinal: Double = System.nanoTime
10
11        println(s"Resultado: $resultado")
12        println(s"Tempo (ms): ${((tempoFinal - tempoInicial) / 1000000)
            }")
13
14      }
```

5.2 Manipulação de Listas

Este programa tem por finalidade gerar um lista com números inteiros de 1 a 1000000, em seguida o programa deve filtrar todos os números pares dessa lista.

5.2.1 Java

Implementação em Java:

```
1  import java.util.ArrayList;
2  import java.util.List;
3
4  public class FiltrarNumerosPares {
5
6    public static void main(String[] args) {
7      double tempoInicial = System.nanoTime();
8
9      List<Long> resultado = new ArrayList<Long>();
10
11     for (long numero = 1; numero <= 1000000; numero++) {
12       if (numero % 2 == 0) {
13         resultado.add(numero);
14       }
15     }
16
17     double tempoFinal = System.nanoTime();
18
19     System.out.println("Resultado: " + resultado.size());
20     System.out.println("Tempo (ms): " + ((tempoFinal -
           tempoInicial) / 1000000));
21   }
22
23 }
```

5.2.2 Clojure

Implementação em Clojure:

```
1  (ns manipulacaodelistas.core
2    (:gen-class))
3
4  (defn -main
```

```
5         [args]
6
7         (def tempoInicial (double (. System (nanoTime))))
8
9         (def numeros (range 1 1000001))
10        (def resultado (filter #(= 0 (rem % 2)) numeros))
11
12        (def tempoFinal (double (. System (nanoTime))))
13
14        (println (format "Resultado: %d" (.size resultado)))
15        (println (format "Tempo (ms): %.6f" (/ (- tempoFinal
              tempoInicial) 1000000))))
```

5.2.3 Scala

Implementação em Scala:

```
1     object FiltrarNumerosPares extends App {
2
3       val tempoInicial: Double = System.nanoTime
4
5       val numeros = 1 to 1000000
6       val resultado = numeros.view.filter(_ % 2 == 0)
7
8       val tempoFinal: Double = System.nanoTime
9
10      println(s"Resultado: ${resultado.size}")
11      println(s"Tempo (ms): ${((tempoFinal - tempoInicial) / 1000000)
            }")
12
13    }
```

5.3 Cálculo Fatorial

Este programa tem por finalidade calcular o fatorial do número 50.

5.3.1 Java

Implementação em Java:

```
1     public class CalcularFactorial {
```

```
2
3       public BigInteger factorial(BigInteger f, BigInteger n) {
4         if (n.compareTo(new BigInteger("1")) == 0)
5           return f;
6         else
7           return factorial(f.multiply(n), n.subtract(new BigInteger("
              1")));
8       }
9
10      public static void main(String[] args) {
11        double tempoInicial = System.nanoTime();
12
13        CalcularFactorial c = new CalcularFactorial();
14        BigInteger resultado = c.factorial(new BigInteger("1"), new
            BigInteger("50"));
15
16        double tempoFinal = System.nanoTime();
17
18        System.out.println("Resultado: " + resultado);
19        System.out.println("Tempo (ms): " + ((tempoFinal -
            tempoInicial) / 1000000));
20      }
21
22    }
```

5.3.2 Clojure

Implementação em Clojure:

```
1     (ns calculofatorial.core
2       (:gen-class))
3
4     (defn -main
5       [args]
6
7       (def tempoInicial (double (. System (nanoTime))))
8
9       (defn factorial [f n]
10        (if (= n 1)
11          f
12          (factorial (* f n) (dec n))))
13
```

```
14      (def resultado (factorial 1N 50N))
15
16      (def tempoFinal (double (. System (nanoTime))))
17
18      (println (format "Resultado: %d" (biginteger resultado)))
19      (println (format "Tempo (ms): %.6f" (/ (- tempoFinal
          tempoInicial) 1000000))))
```

5.3.3 Scala

Implementação em Scala:

```
1   object CalcularFactorial extends App {
2
3     val tempoInicial: Double = System.nanoTime
4
5     def factorial(f: BigInt, n: BigInt): BigInt = {
6       if (n == 1) f else factorial((f * n), (n - 1))
7     }
8
9     val resultado = factorial(1, 50)
10    val tempoFinal: Double = System.nanoTime
11
12    println(s"Resultado: $resultado")
13    println(s"Tempo (ms): ${((tempoFinal - tempoInicial) / 1000000)
        }")
14
15  }
```

6 TESTES E RESULTADOS

Neste capítulo, são apresentados os resultados obtidos pelos programas propostos no capítulo anterior. Os dados medidos para cada programação são o tempo de execução, sendo que cada programa foi executado dez vezes para assim se obter os três menores tempos e a quantidade de *Lines of Code - Linhas de Código* (LoC).

6.1 Compilando os Programas

Os programas em Java foram compilados com seu compilador padrão *javac*, conforme demonstrado a seguir:

```
1 javac SomarNumeros.java
2 javac FiltrarNumerosPares.java
3 javac CalcularFactorial.java
```

Os programas em Scala foram compilados com seu compilador padrão *scalac*, conforme demonstrado a seguir:

```
1 scalac SomarNumeros.scala
2 scalac FiltrarNumerosPares.scala
3 scalac CalcularFactorial.scala
```

Os programas em Clojure foram compilados através da ferramenta de compilação disponível para programas escritos em Clojure chamada Leiningen. Para compilar cada programa Clojure é necessário acessar a sua pasta e executar o comando:

```
1 lein compile
```

O tempo de compilação do compilador Java leva vantagem quando comparado com os compiladores das linguagens Clojure e Scala. Por este motivo essas linguagens tem colocado muitos esforços para evoluir seu processo de compilação e torná-los mais rápidos. Um recurso que vem sendo incorporado pela linguagem Scala através da ferramenta SBT e pelo Clojure através da ferramenta LEIN, é a compilação incremental. Onde após uma primeira compilação completa do programa, as próximas compilações são realizadas apenas em arquivos que sofreram alterações.

6.2 Executando os Programas

Os programas em Java foram executados com seu interpretador padrão *java* na mesma pasta onde cada programa foi compilado, conforme demonstrado a seguir:

```
1  java SomarNumeros
2  java FiltrarNumerosPares
3  java CalcularFactorial
```

Os programas em Scala foram executados com seu interpretador padrão *scala* na mesma pasta onde cada programa foi compilado, conforme demonstrado a seguir:

```
1  scala SomarNumeros
2  scala FiltrarNumerosPares
3  scala CalcularFactorial
```

Os programas em Clojure foram executados através da ferramenta Leiningen. Para executar cada programa Clojure é necessário acessar a sua pasta e executar o comando:

```
1  lein run
```

6.3 Resultados

6.3.1 Tempo de Processamento

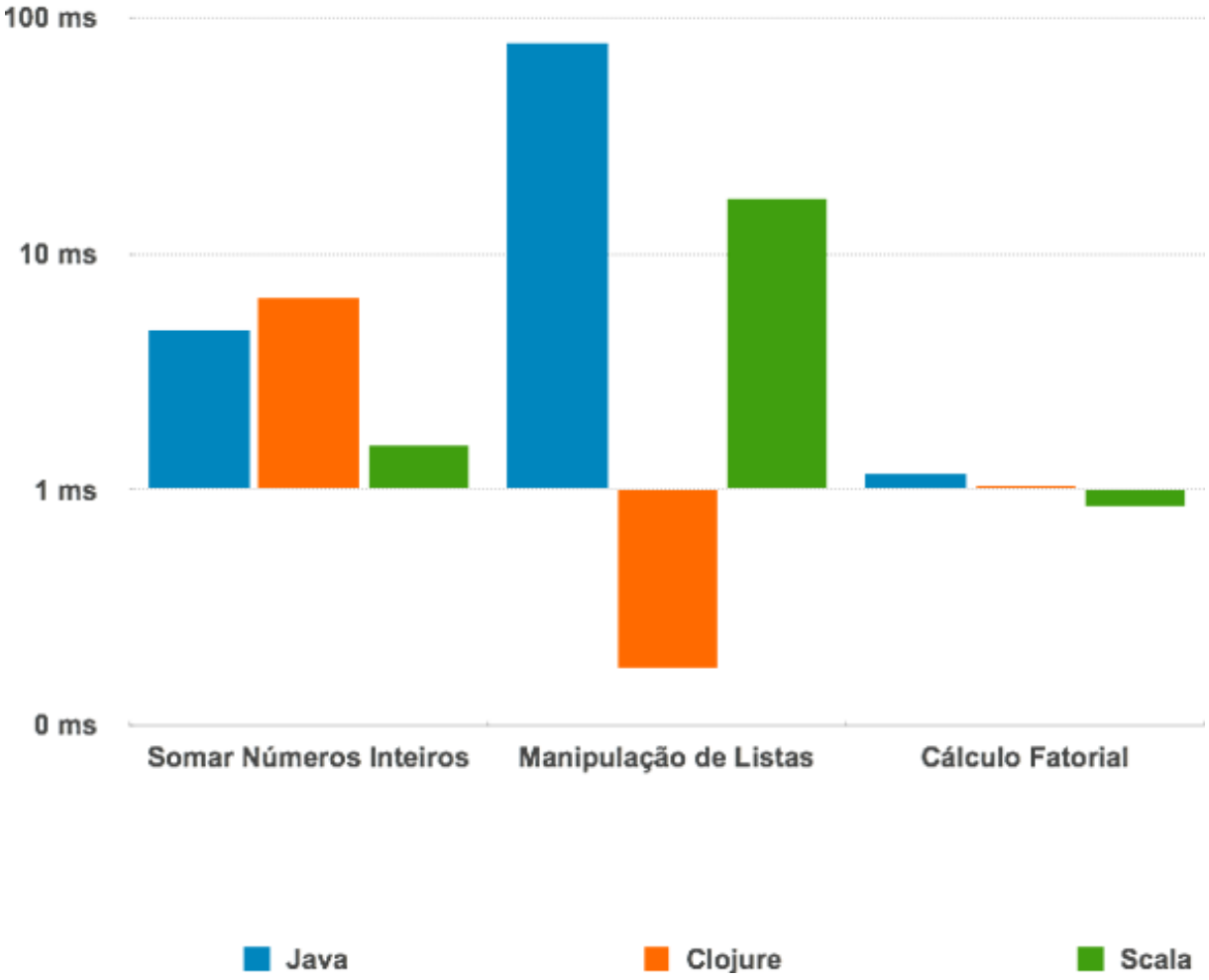
Todos os programas apresentaram os mesmo resultados, e o tempo de processamento utilizado foram os 3 menores de de 10 coletados para cada programa.

Tabela 1 – *Tempo (ms) de Processamento Gasto Por Cada Programa*

Programa (Tentativa)	Java	Clojure	Scala
Somar Números Inteiros (1)	4.815872 ms	6.564096 ms	1.544192 ms
Somar Números Inteiros (2)	4.71808 ms	6.727168 ms	1.579008 ms
Somar Números Inteiros (3)	4.827136 ms	6.550016 ms	1.607936 ms
Manipulação de Listas (1)	78.750976 ms	0.175872 ms	17.15712 ms
Manipulação de Listas (2)	79.156992 ms	0.180992 ms	17.278208 ms
Manipulação de Listas (3)	78.694144 ms	0.178944 ms	17.127936 ms
Cálculo Fatorial (1)	1.179136 ms	1.050112 ms	0.84608 ms
Cálculo Fatorial (2)	1.181952 ms	1.041920 ms	0.866048 ms
Cálculo Fatorial (3)	1.180928 ms	1.049088 ms	0.851968 ms

Os melhores (menores) tempos de cada programa em cada linguagem de programação foram utilizados para compor o gráfico a seguir.

Gráfico 1 – Tempo de processamento gasto por cada linguagem de programação



Fonte: Dados da pesquisa

6.3.2 Linhas de Código

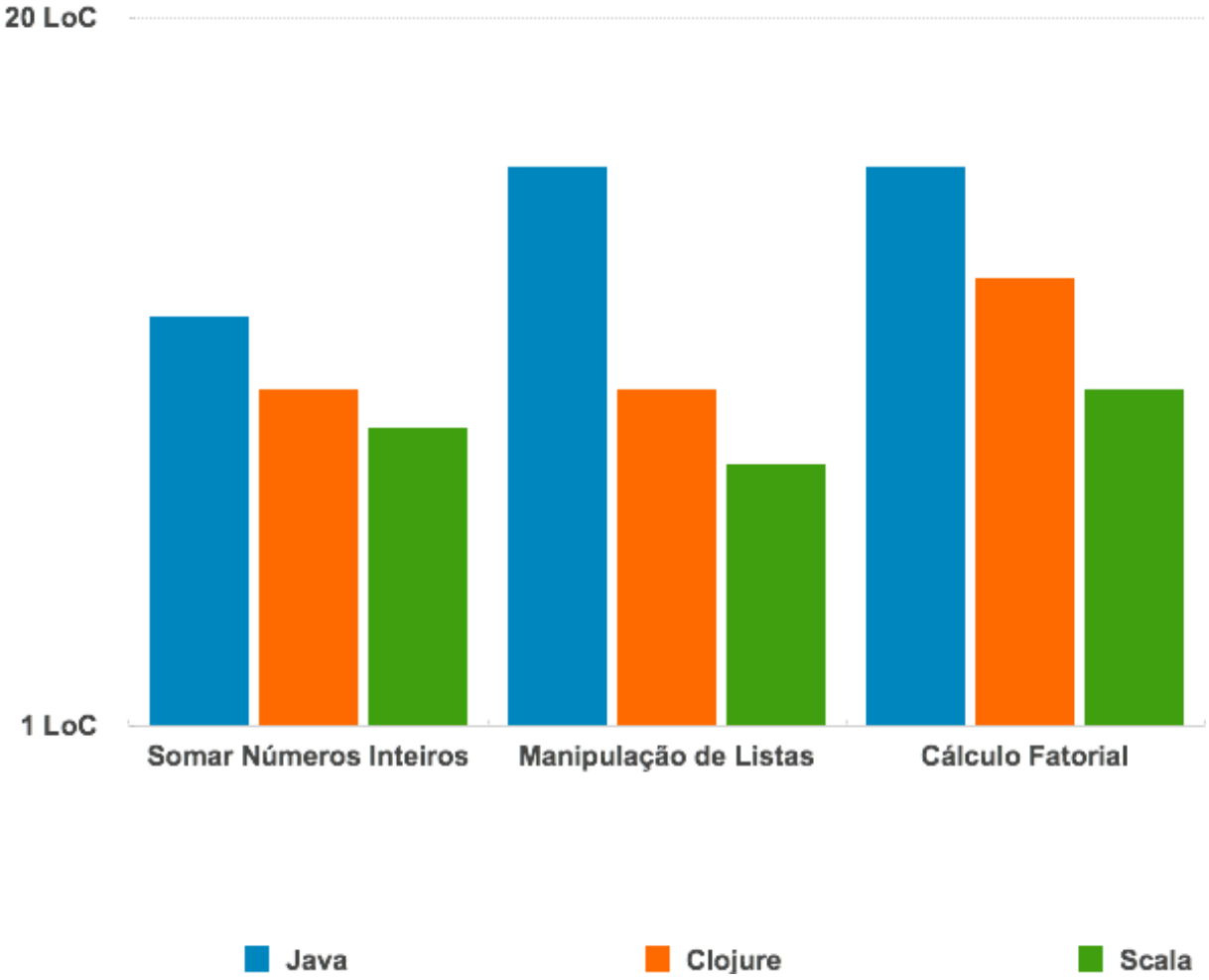
Para analisar a legibilidade de cada programa foi utilizado a quantidade de linhas de código como uma medida. As linhas que não continham nenhum caractere não foram acrescentadas a tabela a seguir.

Tabela 2 – *Linhas de Código Utilizada Por Cada Programa*

Programa	Java	Clojure	Scala
Somar Números Inteiros	12	10	9
Manipulação de Listas	16	10	8
Cálculo Fatorial	16	13	10

Os melhores (menores) tempos de cada programa em cada linguagem de programação foram utilizados para compor o gráfico a seguir.

Gráfico 2 – Linhas de código por cada linguagem de programação



Fonte: Dados da pesquisa

7 CONCLUSÃO

A linguagem de programação Java é uma linguagem muito popular e muito utilizada para se construir programas das mais diversas finalidades. Com o passar do tempo, mesmo com a evolução tanto da linguagem Java como da máquina virtual Java, é possível notar que em determinados tipos de algoritmos a linguagem Java atrapalha ou impede o desenvolvedor de escrever programas de forma mais concisa e mantendo um bom desempenho, principalmente em códigos executados de forma concorrente ou que precisam iterar sobre coleções de objetos realizando agrupamentos ou filtros.

Para desenvolvedores com bom conhecimento sobre a linguagem de programação Java e que não possuem conhecimento sobre nenhum dialeto Lisp, aprender e entender um programa escrito em Scala torna-se mais fácil e intuitivo, pois embora Scala faça um forte uso do paradigma funcional, ela ainda utiliza alguns conceitos do paradigma de orientação a objetos. Um outro ponto importante é que Scala também é estaticamente tipada assim como a linguagem Java, além de que quando compara com a linguagem Clojure, Scala ainda possui uma sintaxe mais próxima da linguagem Java.

Ao contrário da linguagem Java, as linguagens Clojure e Scala são pouco conhecidas e ainda pouco utilizadas, embora seu uso vem sendo encorajado devido a grandes empresas como Twitter, LinkedIn, Facebook e Netflix anunciarem que vem utilizando essas linguagens com grande êxito em seus sistemas.

Em face de tudo o que foi exposto anteriormente, conclui-se que é possível obter mais desempenho com programas mais concisos fazendo uso de linguagens que utilizam o paradigma de programação funcional dentro da máquina virtual Java, assim como foi visto nos capítulos introdutórios sobre o paradigma funcional e sobre as linguagens Clojure e Scala.

REFERÊNCIAS

CLARO, D. B.; SOBRAL, J. B. M. **Programação em JAVA**. 1. ed. Florianópolis, SC.: In Copyleft Pearson Education, 2008.

FOGUS, M. **Functional Javascript**: Introducting functional programming with underscore.js. United States of America: O'Reilly Media, 2013. ISBN 1449360777.

HALLOWAY, S.; BEDRA, A. **Programming in Clojure**. 2. ed. United States of America: Pragmatic Bookshelf, 2012.

HASKELL. **What is functional programming?** Disponível em http://www.haskell.org/haskellwiki/Functional_programming. Acesso em 23 set. 2013.

HORSTMANN, C. S.; CORNELL, G. **Core Java**: Volume i - fundamentos. 8. ed. São Paulo: Pearson Pretice Hall, 2010.

JOBS, S. **Frase de Steve Jobs em discurso de iniciação em Stanford, 2005**. Disponível em http://www.quemdisse.com.br/frase.asp?frase=95623. Acesso em 01 set. 2013.

JR., H. E. **Engenharia de Software na Prática**. 1. ed. Brasil: Novatec, 2010.

LINDHOLM, T. et al. **The Java Virtual Machine Specification**: Java se 7 edition. 1. ed. United States of America: Addison-Wesley Professional, 2013. ISBN 0133260445.

MICHAELSON, G. **An Introduction to Functional Programming Through Lambda Calculus**. 1st. ed. England: Addison-Wesley, 1989.

ORACLE. **Java History**. Disponível em http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html. Acesso em 01 out. 2013.

ORACLE. **Java Timeline**. Disponível em http://oracle.com.edgesuite.net/timeline/java/. Acesso em 01 out. 2013.

RATHORE, A. **Clojure in Action**: Elegant applications on the jvm. 1. ed. United States of America: Manning Publications Co, 2012.

RICARTE, I. L. M. **Polimorfismo**. Disponível em http://dca.fee.unicamp.br/cursos/PooJava/polimorf/. Acesso em 21 set. 2013.

SAFALRA. **Formal Description Of Lambda Calculus**. Disponível em http://safalra.com/science/lambda-calculus/formal-description/. Acesso em 20 set. 2013.

SIGNIFICADOS. **Significado de Paradigma**. Disponível em http://www.significados.com.br/paradigma/. Acesso em 01 out. 2013.

SUBRAMANIAM, V. **Programming Scala**: Tackle multi-core complexity on the java virtual machine. 1. ed. United States of America: Pragmatic Bookshelf, 2009.

TATE, B. A. **Seven Languages In Seven Weeks**: A pragmatic guide to learning programming languagens. 1. ed. United States of America: Pragmatic Bookshelf, 2010.

WAMPLER, D. **Functional Programming for Java Developers**: Tools for better concurrency, abstraction, and agility. 1. ed. United States of America: O'Reilly Media, 2011.

WAMPLER, D.; PAYNE, A. **Programming Scala**. 1. ed. United States of America: O'Reilly Media, 2009.

ZEROTURNAROUND. **What is Lambda Calculus and should you care?** Disponível em http://zeroturnaround.com/rebellabs/what-is-lambda-calculus-and-why-should-you-care/. Acesso em 21 set. 2013.