
**UNIVERSIDADE FEDERAL DE CAMPINA GRANDE**
**CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA**
**UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO**
**CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO**

**ANTUNES DANTAS DA SILVA**

# UM ESTUDO SOBRE DIFERENTES LINGUAGENS DE PROGRAMAÇÃO PARA INTRODUÇÃO DA PROGRAMAÇÃO FUNCIONAL

**CAMPINA GRANDE - PB**

**2019**

**ANTUNES DANTAS DA SILVA**

# UM ESTUDO SOBRE DIFERENTES LINGUAGENS DE PROGRAMAÇÃO PARA INTRODUÇÃO DA PROGRAMAÇÃO FUNCIONAL

**Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.**

**Orientador: Professor Dr. Everton Leandro Galdino Alves.**

**CAMPINA GRANDE - PB**

**2019**


S586e Silva, Antunes Dantas da.
Um estudo sobre diferentes linguagens de programação para introdução da programação funcional. / Antunes Dantas da Silva. – 2019.

14 f.

Orientador: Prof. Dr. Everton Leandro Galdino Alves.
Trabalho de Conclusão de Curso - Artigo (Curso de Bacharelado em Ciência da Computação) - Universidade Federal de Campina Grande; Centro de Engenharia Elétrica e Informática.

1. Programação. 2. Linguagens de programação. 3. Programação funcional. 4. Estudo de programação. I. Alves, Everton Leandro Galdino. II. Título.

CDU:004(045)

**Elaboração da Ficha Catalográfica:**

Johnny Rodrigues Barbosa
Bibliotecário-Documentalista
CRB-15/626

**ANTUNES DANTAS DA SILVA**

# UM ESTUDO SOBRE DIFERENTES LINGUAGENS DE PROGRAMAÇÃO PARA INTRODUÇÃO DA PROGRAMAÇÃO FUNCIONAL

**Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.**

## BANCA EXAMINADORA:

**Professor Dr. Everton Leandro Galdino Alves**
**Orientador – UASC/CEEI/UFCG**

**Professor Dr. Wilkerson de Lucena Andrade**
**Examinador – UASC/CEEI/UFCG**

**Professor Dr. Tiago Lima Massoni**
**Disciplina TCC – UASC/CEEI/UFCG**

**Trabalho aprovado em: 25 de novembro 2019.**

**CAMPINA GRANDE - PB**

# Um Estudo sobre Diferentes Linguagens de Programação para Introdução da Programação Funcional

Trabalho de Conclusão de Curso

Antunes Dantas da Silva
antunes.silva@ccc.ufcg.edu.br
Universidade Federal de Campina Grande
Campina Grande, Paraíba

Everton L. G. Alves
everton@computacao.ufcg.edu.br
Universidade Federal de Campina Grande
Campina Grande, Paraíba

## RESUMO

Um paradigma de programação determina a visão que o programador possui sobre a estruturação e execução do programa. As linguagens são classificadas por paradigmas. Dentre os principais paradigmas, o funcional tem crescido nos últimos anos. Muitas instituições de ensino tem incorporado seu estudo e algumas tem, inclusive, adotado como paradigma de ensino introdutório à programação. Há diversas linguagens de programação que incorporam o paradigma funcional e escolher uma delas como base de ensino requer atenção quanto à como os principais recursos são implementados. Tendo esse problema em vista, o objetivo deste trabalho é analisar duas linguagens funcionais (Haskell e Clojure) e uma multiparadigma (Javascript) buscando observar o processo de aprendizado de programação funcional nessas linguagens. Desenvolvendo um mesmo sistema nessas três linguagens, é esperado que ao final tenha-se um balanço geral quanto à facilidade de aprendizado, velocidade de implementação e legibilidade do código nas três linguagens.

## 1 INTRODUÇÃO

A Ciência da Computação é uma área relativamente nova, tendo seu maior desenvolvimento a partir da década de cinquenta [12]. Nesse sentido, o ensino da arte da programação ainda está avançando. O comunicar ao computador o que deve ser feito para que este resolva um problema específico (programação) é uma atividade que pode ser realizada de diversas formas possíveis. O ensino/aprendizagem de programação é ainda uma atividade bastante complexa. É comum que alunos tenham grandes dificuldades durante o processo de aprendizado. Diversos estudos ressaltam problemas na compreensão de estruturas de dados, no uso de variáveis, dentre outros. Elementos estes que são a base da programação imperativa [8] [15]. Outros estudos mostram que, apesar de compreenderem em teoria os conceitos básicos dos paradigmas e linguagens de programação, os alunos sentem dificuldade em aplicá-los na prática [9].

O paradigma de uma linguagem de programação pode ser definido como a abordagem a qual a comunicação será realizada, ou seja, o estilo que guia a escrita do programa [22]. Existem diversos paradigmas de linguagem de programação: imperativo, orientado à objetos, funcional, lógico, orientado à eventos, dentre outros. Eles variam de acordo com o nível de abstração, maneira como lidam com os estados da aplicação, valores que podem ser armazenados, etc. O paradigma imperativo, por exemplo, é baseado na descrição de um passo-a-passo do que deve ser feito para resolver um problema, isto se dá através do uso de variáveis, comandos e procedimentos. O paradigma orientado à objetos eleva o nível de abstração ao permitir a criação de entidades que encapsulam lógicas específicas e podem ser reutilizados em diferentes contextos. A programação lógica baseia-se no fornecimento de fatos e regras que são apresentados ao computador. De posse dos fatos, a linguagem deve inferir respostas para o problema [13].

Já o paradigma funcional, busca emular funções matemáticas. De acordo com [13], este paradigma enfatiza a aplicação de funções, que são tratadas como valores importantes e que podem ser utilizadas em qualquer parte da aplicação. Isso permtite uma grande reusabilidade de código, bem como um projeto com maior legibilidade. Linguagens funcionais tendem a não permitir alteração de estados (variáveis), o que fornece segurança contra modificações inesperadas (efeitos colaterais).

Comumente, o ensino de programação é introduzido utilizando linguagens que seguem o paradigma imperativo [18]. Porém, aprender diferentes paradigmas é importante, uma vez que determinados problemas são mais facilmente resolvíveis utilizando um determinado paradigma. A transparência referêncial (ausência de efeitos colaterais) da programação funcional, por exemplo, pode ser valiosa para aplicação do mercado financeiro, que preza por segurança.

Contudo, é comum que um estudante, que foi introduzido à programação através do estilo imperativo, tenha grandes dificuldades quando se depara com um novo paradigma. Pois, além de uma nova sintaxe (nova linguagem), este precisa compreender diversos conceitos novos: as características do paradigma, o modelo de organização do código, a estratégia de modelagem dos problemas, o fluxo de execução e estado da aplicação, etc. São inúmeros os aspectos que podem confundir um aprendiz quando focando em um paradigma que não está familiarizado.

Dentre estes aspectos que podem afetar o aprendizado do novo paradima, uma das mais importantes é a linguagem de programação escolhida. A linguagem de programação escolhida implementa de certa forma os conceitos teóricos do paradigma que se deseja aprender: estruturas de dados, definição de função, entrada e saída de dados, iteratividade, sistema de tipos, etc. Apesar de semelhantes, linguagens de um mesmo paradigma normalmente implementam diferentemente tais características, o que pode influenciar diretamente à curva de aprendizado do aluno.

Uma alternativa seria o uso de linguagens multiparadimas. Porém, linguagens que implementam múltiplos paradigmas levar o aprendiz a cometer vícios (e.g., usar construções imperativas). Tais vícios podem acabar por desvirtuar o processo de aprendizado.

Diversas linguagens implementam o paradigma funcional. Normalmente, aprender programação funcional depois que já se conhece a programação imperativa tende a ser um processo difícil. Pois, exige uma mudança da forma como o programador pensa/desenvolve suas soluções. Por exemplo, a utilização e variáveis e a alteração do valor destas durante a execução do problema, não é algo aplicável na programação funcional pura. Além disso, a recursão é uma ferramenta amplamente utilizada na programação funcional, sendo o principal mecanismo para se iterar sobre coleções de elementos. O pensar algoritmicamente de maneira recursiva também pode ser complexo, uma vez que não se é exigido na programação imperativa. Além disso, há linguagens que levam o conceito de pureza muito forte, separando completamente operações que causam efeito colateral (como operações de entrada e saída) das operações que não alteram estado nem possuem efeito colateral. Essa separação muitas vezes acaba por confundir o aprendiz.

Nesse contexto, a disciplina de Paradigmas de Linguagem de Programação (PLP) da UFCG permite que seus alunos tenham o primeiro contato com dois novos paradigmas (Funcional e Lógico). Isto se dá através da combinação de aulas teóricas, e aulas práticas onde os alunos realizam exercícios de programação utilizando as linguagens Haskell (funcional) e Prolog (lógico). Os alunos já chegam em PLP com conhecimento pévio do paradigma Imperativo. Historicamente, os alunos de PLP têm uma maior dificuldade com a programação funcional. Nesse sentido, faz-se necessário investigar mecanismos para reduzir tal dificuldade. Nas aulas teóricas, os alunos são introduzidos aos principais conceitos do paradigma juntamente com exemplos em Haskell. Depois, eles são levados a exercitar os conceitos aprendendidos em laboratórios (exercícios de programação) e através de um projeto simples desenvolvido nos paradigmas lógico e funcional. Muitos alunos, porém, sentem dificuldade com Haskell. Assim, existe a necessidade de investigar a possibilidade da introdução de outras linguagens para o ensino desse paradigma.

Este trabalho tem o objetivo de avaliar um conjunto de linguagens de programação com o intuito de observar qual a melhor opção para introdução de conceitos básicos do paradigma funcional. Para tal, foram escolhidas as linguagens Haskell, Clojure e Javascript e os seguintes conceitos de análise: estrutura de dados, sistema de tipos, declaração de função, entrada e saída de dados e iteratividade. Além disso, foi realizado um estudo com os alunos que estão cursando a disciplina atualmente para que estes avaliassem implementações nas linguagens estudadas considerando aspectos como legibilidade, manutenibilidade e corretude da implementação.

A execução do estudo trouxe resultados interesssantes. O primeiro autor, mesmo com anos de experiência em programação (majoritariamente imperativa) ainda teve dificuldades para aprender o paradigma funcional. Porém, foi notado que fazer isso com linguagens puramente funcionais foi melhor. Javascript, por ser multiparadigma, permite muitos vícios de programação. Haskell é uma excelente linguagem, mas sua maneira de lidar com funções com efeito colateral (entrada e saída) acaba por atrapalhar quem está aprendendo o paradigma no momento. Clojure, apesar de possuir uma sintaxe menos clara, foi uma grata surpresa por permitir acesso à métodos já conhecidos do mundo Java, além de ser maleável com relação a entrada e saída. Clojure possui uma sintaxe para casamento de padrões mais complexa, mas ainda assim se mostrou ser uma ótima linguagem para quem está aprendendo programação funcional e mais indicada para primeira linguagem.

## 2 PROGRAMAÇÃO FUNCIONAL

Paradigmas de linguagem de programação são os diversos estilos que o desenvolvedor tem para resolver um problema [2]. Um paradigma diz respeito ao conjunto de características que uma linguagem implementa. Existem diversos paradigmas, sendo os mais comuns o imperativo, funcional, lógico, orientado à objetos. Muitas linguagens atuais são multiparadigma, ou seja, implementam características de mais de um paradigma. Java, por exemplo, é imperativa e orientada à objeto, bem como vem implementando conceitos funcionais em suas últimas versões, como o uso de expressões lambda [19].

É possível classificar os paradigmas em dois grandes grupos: imperativo e declarativo. As linguagens do primeiro grupo são mais descritivas, onde o programador precisa descrever todos os passos necessários para resolver o problema. Programas segundo este paradigma são comumente associados à uma “receita de bolo”. Já as linguagens declarativas, possuem construções mais abstratas e focam na descrição do problema.

As linguagens funcionais são declarativas e promovem a construção de software baseado em funções matemáticas [14]. Programas funcionais tendem a tomar como base a decomposição do problema em pequenas funções que se agregam a funções mais genéricas para compor a solução. Linguagens que seguem o paradigma funcional normalmente tratam funções como valores de primeira ordem [10]. Logo, podem ser armazenadas em variáveis, passadas como parâmetro de funções e serem retorno de outra função. Essa capacidade amplia o leque de possibilidades para resolução de problemas.

O paradigma funcional não é algo novo. Tem suas origens no cálculo lambda [17]. Em 1958 foi criada a primeira linguagem funcional, Lisp [23], que influenciou diversas outras linguagens, como Clojure [5].

No inicio, linguagens funcionais se restringiam apenas ao ambiente acadêmico. Porém, nos últimos anos, o interesse por programação funcional tem crescido bastante. Dados do Google mostram que as buscas pelo termo "Functional Programming" dobrou nos últimos 10 anos. O gráfico abaixo mostra o crescimento das buscas pelo referido termo no período de 2009 - 2019.

Além disso, o crescimento do uso de conceitos funcionais em linguagens populares como Java [19] e o crescimento de Javascript [6] fizeram com que a comunidade tivesse outros olhos para a programação funcional. A fintech brasileira Nubank, por exemplo, adotou Clojure como a linguagem para seus serviços e tem obtido muito sucesso [1]. Universidades tem adotado programação funcional nas suas grades curriculares [4].

Os benefícios do paradigma são inúmeros. Dentre eles, destaco a transparência referêncial, funções como valores de primeira ordem. A primeira diz respeito ao valor de uma expressão, que sempre será o mesmo independente da ordem em que foi executado já

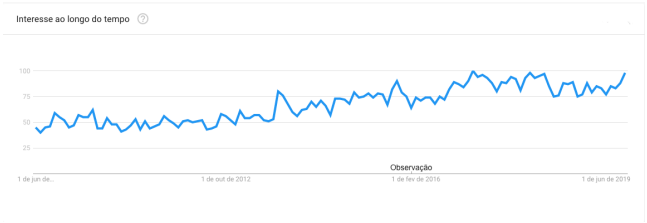


**Figure 1: Crescimento da procura por Functional Programming.**

que funções funcionais puras não realizam efeito colateral. Isso torna a programação muito menos suscetível à erros causados por, especialmente, mudanças de estado que não são atômicas. Já o uso de funções como valores de primeira ordem nos permite abstrair um problema em diversas funções que se agregam para resolvê-lo. Isso permite o desenvolvedor a construir uma solução mais desacoplada, reutilizável e simples.

Para ter-se ideia do poder da programação funcional e de como a programação declarativa permite escrever código de maneira mais objetiva e legível, segue a implementação do algoritmo de ordenação Quicksort em duas linguagens, C (Listing 1), que é imperativa, e Clojure (Listing 2), funcional. O código Clojure possui 6 linhas, enquanto o código para o mesmo algoritmo em C tem 22 linhas. O quicksort é um ótimo exemplo de como a programação funcional tende a ser menos verborágica e melhorar a legibilidade do código.

**Listing 1: Implementação do algoritmo Quicksort em C.**

```
void quicksort(int a[], int lo, int hi) {
  int h, l, p, t;
  if (lo < hi) {
    l = lo;
    h = hi;
    p = a[hi];
    do {
      while ((l < h) && (a[l] <= p))
          l = l+1;
      while ((h > l) && (a[h] >= p))
          h = h-1;
      if (l < h) {
          t = a[l];
          a[l] = a[h];
          a[h] = t;
      }
    } while (l < h);
    a[hi] = a[l];
    a[l] = p;
    quicksort( a, lo, l-1 );
    quicksort( a, l+1, hi );
  }
}
```

**Listing 2: Implementação do algoritmo Quicksort em Clojure.**

```
(defn quick-sort [[pivot & coll]]
  (when pivot
    (concat (quick-sort (filter #(< % pivot) coll))
            [pivot]
            (quick-sort (filter #(>= % pivot) coll))))
    )
```

## 3 PROJETO DO ESTUDO

Nesta seção, iremos definir e discutir os elementos de projeto do nosso estudo investigativo para comparação de linguagens de programação para a introdução de conceitos básicos da programação funcional.

### 3.1 Linguagens

O primeiro ponto importante de referência para nossa pesquisa é a decisão acerca das linguagens a serem investigadas. Foram escolhidas três: Haskell, Clojure e Javascript. A primeira foi escolhida por ser a linguagem atualmente utilizada na disciplina Paradigmas de Linguagem de Programação da UFCG para introdução do paradigma funcional. Além disso, esta é uma das linguagens mais referenciadas no ensino de programação funcional [21] [3]. Já Clojure é uma linguagem baseada no dialeto de Lisp, uma das mais antigas linguagens funcionais. Seu uso tem crescido bastante nos últimos anos e o mercado vem adotando a linguagem, especialmente em instituições financeiras [1]. O fato de ser executada sob a JVM (Java Virtual Machine) garante interoperabilidade com classes feitas em Java e familiaridade do programador para com o ambiente. Por útimo, Javascript foi escolhida por ser uma linguagem multiparadigma, amplamente utilizada no desenvolvimento web e na criação de microsserviços dada a sua sintaxe simples e poderosa. Esta, implementa o paradigma funcional, possibilitando o uso de função como valores de primeira classe. Javascript foi uma das grandes responsáveis pelo crescimento do interesse no paradigma funcional nos últiumos anos, devido à implementação de funções anônimas, funções como valores de primeira ordem e sua facilidade por ser flexível o suficiente para permitir que programadores acostumados com linguagens procedurais pudessem se aventurar, quando confortáveis, em um pensamento mais declarativo.

### 3.2 Tópicos de Análise

Uma vez escolhidas as linguagens, o próximo passo foi definir quais tópicos da programação funcional seriam estudados no contexto linguagens. As linguagens citadas oferecem mecanismos para ajudar na implementação desses tópicos. Porém, como elas o fazem é o que as diferencia. Para elaboração deste trabalho, foi pensado em um estudo que levasse em conta os principais tópicos estudados e exercitados durante a introdução da programação funcional na disciplina de PLP. Assim, os autores enumeraram eles de acordo com a ementa da disciplina. Foram selecionados cinco pontos importantes a serem explorados. Esses pontos ajudarão o primeiro autor à analisar melhor cada linguagem do estudo com um olhar voltado ao aprendizado do paradigma.

- **Entrada/Saída**: Todas as atividades práticas realizadas durante o ensino de programação funcional para os alunos requerem o uso de entrada e saída de dados através do canal

padrão do dispositivo (terminal). Isso pode ser um grande entrave para os alunos pois requer a implementação com efeito colateral, o que viola um dos princípios da programação funcional. Assim, algumas linguagens tratam essa funcionalidade de maneira diferente, separando ela como uma parte "impura" da linguagem, o que pode acabar gerando confusão.

- **Sistema de Tipo**: No contexto de linguagem de programação, sistema de tipos é o conjunto de regras que define uma propriedade tipo para as construções da linguagem como valor de retorno de uma função, parâmetro de função, variável, dentre outros. Na programação, é comum a utilização e definição de tipos. Nas linguagens funcionais, esse é um ponto importante a ser estudado pois, em sua maioria, elas são fortemente tipadas [20], não permitindo operações entre tipos incompatíveis (e.g., somar um valor booleano com um inteiro). Além disso, a presença ou ausência do conceito de "variáveis" poder ser algo a ser analisar ainda neste tópico.
- **Estruturas de Dados**: Construções muito importantes nas linguagens de programação, pois permitem ao programador lidar com coleções de dados a fim de armazená-los e processá-los. Esse, inclusive, é um ponto muito interessante a ser analisado, já que o estado dos objetos são imutáveis na programação funcional, logo a maneira como alterações são feitas na estrutura podem impactar fortemente no desenvolvimento de aplicações. Nas linguagens funcionais, as estruturas de dados tem um formato diferente das linguagens imperativas. Elas são imutáveis, e algumas linguagens usam um carregamento preguiçoso, o que permite a ideia de estruturas infinitas.
- **Iteratividade**: Repetir operações por um determinado, ou não, número de vezes é uma prática rotineira no desenvolvimento de sistemas. Algumas linguagens permitem iterar utilizando laços, como o *for*. Outras, permitem iterar sobre coleções, cadeias de caracteres e propriedades conhecidas como iteradores. O foco neste tópico, então, é analisar quais as diferentes ferramentas que as linguagens oferecem para permitir iterar sobre um código e sua complexidade.
- **Definição de Funções**: Uma função é uma estrutura de código que encapsula uma lógica. Para operar, ela pode receber argumentos que vão ser utilizados como base para gerar um valor final. Declaração de função envolve diversos pontos, como nomeação, polimorfismo, comportamento para certos valores de entrada, possibilidade de valores a serem retornados, fechamento, dentre outros. Analisar como as linguagens implementam essas possibilidades é ponto crucial deste trabalho visto que o software em si é uma função decomposta em diversas outras funções.

## 3.3 Metodologia

Para avaliar cada um desses tópicos, o autor realizou um estudo comparativo onde implementou um mesmo sistema exemplo, que consiste em um controle de estoque e vendas, utilizando as três linguagens. A especificação desse sistema foi dividida em quatro casos de uso, onde cada um deles aborda fortemente pelo menos uma das características citadas acima. A especificação do sistema foi coletada na disciplina Laboratório de Programação 2 do curso de Ciência da Computação da UFCG. Este sistema foi escolhido por sua adequação ao objetivo da pesquisa, bem como estar no nível de complexidade exigido para alunos do terceiro período do curso de Ciência da Computação da UFCG, período sugerido para alunos cursem a disciplina PLP. No decorrer do desenvolvimento, o autor coletou métricas e fez avaliações subjetivas sobre a experiência no desenvolvimento de cada caso de uso em cada uma das linguagens. As métricas serão discutidas posteriormente. É importante destacar que a implementação dos casos de uso foi realizada pelo primeiro autor. Porém, para evitar ganhos de tempo referentes ao melhor entendimento da especificação, a ordem de uso das linguagens de programacão foi alternada a cada novo caso de uso. Além disso, o autor pouco, ou nenhum, conhecimento prévio sobre as linguagens, o que emula o comportamento padrão dos alunos de PLP. A seguir, é apresentado um exemplo de caso de uso utilizado no contexto do estudo. Os demais casos de uso podem ser encontrados no nosso site [1]:

Caso de Uso 1 - Menu de Entrada:

- Pré-condição: o sistema estar sendo executado para interação via linha de comando.
- Passo 1: Ao iniciar a aplicação, um menu será carregado no terminal do usuário contendo as opções disponíveis no sistema.
- Passo 2: O usuário digita uma das opções e a funcionalidade escolhida por ele é apresentada na tela.
- Fluxo Alternativo: O usuário digita uma opção inválida. O sistema deve mostrar voltar para o Passo 1.
- Pós Condição: A tela do usuário deve mostrar o conteúdo da opção escolhida.

Analisando os casos de uso, é possível identificar a aplicação dos tópicos de programação discutidos anteriormente. Por exemplo, o primeiro caso de uso trata-se de um menu onde há interação com o usuário para obtenção de dados referentes à execução do problema. No segundo, é visto a necessidade de explorar o sistema de tipos da linguagem para a criação de tipos compostos. A implementação desses casos de uso pode levar a um maior uso de algum dos pontos citados. Por exemplo, é claro no primeiro caso de uso a necessidade de lidar com entrada e saída de dados. Essa funcionalidade das linguagens será muito utilizada para implementar tal caso. Tendo isso em vista, foi montada uma tabela que relaciona cada caso de uso com os principais pontos da linguagem que serão utilizado para implementá-lo e que serão levado mais em conta na análise da implementação de cada caso de uso. O segundo caso de uso explora muito o sistema de tipos da linguagem e as estruturas de dados da linguagem. O terceiro caso de uso faz necessidade de utilizar repetição para buscar elementos em uma lista e realizar cálculos. O último caso de uso reúne a necessidade de iterar e estruturas de dados, combinando essas duas lógicas para prover a sumarização de valores. Definição de função é um tópico que está intrínseco à todos os casos de uso. Por isso, não aparece na tabela.

[1] https://github.com/antunesdantass/tcc

| | I/O | Sistema de Tipo | Est. de Dados | Iteratividade |
|---|---|---|---|---|
| UC 1 | X | | | |
| UC 2 | | X | X | |
| UC 3 | | | | X |
| UC 4 | | | X | X |

Durante a implementação dos casos de uso, as seguintes métricas foram consideradas a fim de fornecer uma análise objetiva para a avaliação do uso das linguagens:

- **Tempo de Implementação**: tempo gasto para implementar cada caso de uso em cada linguagem. É uma métrica importante para avaliar a dificuldade de implementação naquela linguagem, já que é possível inferir que o tempo de implementação foi alto por que houveram dificuldades durante o processo.
- **Tamanho (em linhas de código)**: tamanho final do artefato gerado, importante para análise de reusabilidade e simplicidade da linguagem.
- **Quantidade de builds realizadas**: a obtenção de muitos erros de compilação durante o desenvolvimento pode significar um entendimento errado da sintaxe da linguagem por parte do desenvolvedor/aluno. Linguagens com sintaxe complexa tendem a ter um elevado número de erros de compilação nas primeiras tentativas.
- **Consulta à documentação**: a quantidade de vezes que a documentação da linguagem foi consultada ou foi pesquisado algum aspecto puro da linguagem, como sintaxe ou mensagem de erro. É uma métrica que ajuda a identificar a complexidade de entender a construção da linguagem.
- **Corretude da implementação**: cada caso de uso foi acompanhado por um conjunto de testes de aceitação. Nessa métrica iremos avaliar a capacidade de criação de uma implementação que cumpra todos os requisitos descritos na especificação e verificados pelos testes.

Além da análise quantitativa, como o participante do estudo busca emular o comportamento de estudantes iniciantes em programação funcional, também foi realizada uma análise qualitativa a fim de entender melhor as possíveis dificuldades práticas do uso das linguagens para o aprendizado nesse contexto

## 4 RESULTADOS E DISCUSSÕES

O estudo de implementação do sistema nas três linguagens teve uma duração total de 13 horas e 25 minutos. A tabela abaixo contém as métricas capturadas durante o estudo. A seguir, discutiremos os resultados considerando cada caso de uso. A implementação dos casos de uso considerando todas as linguagens estão disponíveis em nosso site[2].

[2]https://github.com/antunesdantass/tcc

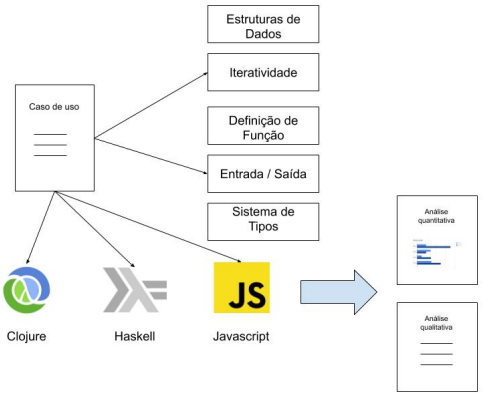


**Figure 2: Diagrama do trabalho**

| Javascript | Tempo(min) | LOC | Builds | Doc. | Corretude (%) |
|---|---|---|---|---|---|
| UC 1 | 62 | 35 | 16 | 8 | 100 |
| UC 2 | 103 | 33 | 11 | 4 | 100 |
| UC 3 | 16 | 26 | 3 | 1 | 100 |
| UC 4 | 3 | 10 | 1 | 0 | 100 |
| Haskell | | | | | |
| UC 1 | 37 | 5 | 11 | 8 | 100 |
| UC 2 | 232 | 12 | 74 | 26 | 100 |
| UC 3 | 74 | 27 | 39 | 14 | 100 |
| UC 4 | 4 | 10 | 1 | 0 | 100 |
| Clojure | | | | | |
| UC 1 | 65 | 7 | 15 | 17 | 100 |
| UC 2 | 90 | 23 | 17 | 12 | 100 |
| UC 3 | 106 | 34 | 25 | 17 | 100 |
| UC 4 | 11 | 14 | 4 | 2 | 100 |

### 4.1 Caso de Uso 1

O primeiro caso de uso explorava a interação com o usuário através do terminal, importante para avaliar como a linguagem lida com entrada e saída de dados. Antes de iniciar o desenvolvimento, o autor pensou em qual estratégia utilizaria para a arquitetura da aplicação. Assim, foi proposto uma arquitetura de módulos, onde o sistema teria um módulo de menu principal que delegaria as chamadas para os módulos de cadastro de produto, venda e balanço do sistema.

A primeira linguagem a ser utilizada foi Haskell. Ela, por ser puramente funcional, separa as operações "puras" (operações que não causam efeito colateral) de operações que causam. Tipicamente, operações de entrada e saída representam efeito colateral (impressão de dados na tela, por exemplo). Para essa distinção, Haskell utiliza o *tipo IO* para identificar que aquela operação não é pura. O menu inicial é um módulo IO, que imprime o texto de boas vindas e recebe do usuário qual a ação que ele deseja executar. O tratamento disso foi simples, com a criação de uma função que recebe como

parâmetro a ação escolhida do usuário e redireciona para o modulo correspondente. O uso de casamento de padrões nesse caso de uso foi fantástico pois o código ficou simples e direto.

A segunda linguagem foi Javascript. A sintaxe é simples. O autor se ateve a um maior uso de funções e evitar o uso de variáveis. Para guardar o texto de apresentação do menu, por exemplo, foi utilizada uma função que não recebe parâmetro algum e retorna o texto, mesma estratégia usada em Haskell. Para a função principal da aplicação, não foi possível utilizar a estratégia de casamento de padrão que há em Haskell, pois Javascript não dá suporte. Isso fez com que a implementação não fosse tão simples, sendo necessário o uso de uma estrutura como *switch* para simular o efeito do casamento de padrões. Empobrecendo a criação de função em Javascript. O autor teve dificuldades com a exportação e importação de módulos da linguaguem, que é um pouco confusa e necessita de um compilador para converter a maneira de importação introduzida no ECMA 2016 para vanilla Javascript. A linguagem não trata entrada e saída como sendo uma parte impura, então lidar com isso foi direto e simples.

Já Clojure causou uma leve estranheza inicial pelo uso de notação prefixada. O autor teve dificuldades em entender como se dava a declaração de novos módulos no sistema e de como referenciá-los em outros arquivos. A escrita do código em si foi tranquila, especialmente lidar com entrada e saída. Clojure não pede nenhum tratamento especial para isso, diferente de Haskell, o que apresentou uma fluidez para o trabalho. O uso de casamento de padrão é um pouco diferente. Em Clojure, há o conceito de definir múltiplas funções que serão chamadas de acordo com uma função de despacho. Foi complexo de entender no início mas depois foi tão poderoso e eficiente quanto em Haskell, o que ajuda bastante na legibilidade. A chamada de função também é simples e clara e a leitura com a notação prefixada se torna direta e fluida. Logo, para esse caso de uso, o uso de Clojure se mostrou a melhor opção. Apesar da análise quantitativa ter mostrado um maior tempo de implementação, a facilidade para lidar com entrada e saída na linguagem apresenta um ganho de compreensão. Após a dificuldade inicial com relação a sintaxe do casamento de padrões, foi simples desenvolver nela, as chamadas aos procedimentos da JVM era fácil e por não precisar de nenhum tratemento para entrada e saída, livrou o programador de ter que realizar alguma alteração no código para isso. A título de exemplificação, os Listings 3, 4 e 5 apresentam as implementações do menu principal em Haskell, Javascript e Clojure, respectivamente. As demais implementações estão disponíveis no nosso site.

**Listing 3: Implementação do menu principal em Haskell.**

```
runTask :: Integer -> IO ()
  runTask 1 = createProduct
  runTask 2 = sell
  runTask 3 = checkout
  runTask 4 = putStrLn ""
  runTask _ = main
```

**Listing 4: Implementação do menu principal em Javascript.**

```
const main = (produtosCadastrados, totalArrecadado)
    ↪ =>{
  const opcao =readLine.question(main_presentation())
  switch (parseInt(opcao)) {
    case 1:
      main(product(produtosCadastrados),
          ↪ totalArrecadado);
      break;
    case 2:
      main(produtosCadastrados, sale(
          ↪ produtosCadastrados, totalArrecadado));
      break;
    case 3:
      checkout(produtosCadastrados, totalArrecadado);
      main(produtosCadastrados, totalArrecadado);
      break;
    case 4:
      process.exit();
    default:
      return main(produtosCadastrados);
  };};
```

**Listing 5: Implementação do menu principal em Clojure.**

```
(defmulti goto-choice identity)
(defmethod goto-choice "1" [_] (create-product))
(defmethod goto-choice "2" [_] (sell-product))
(defmethod goto-choice "3" [_] (checkout))
(defmethod goto-choice "4" [_] (System/exit 0))
```

## 4.2 Caso de Uso 2

Neste, havia um maior trabalho com relação ao sistema de tipos da linguagem, pois seria necessário criar estruturas para representar elementos do produto. Seguindo a estratégia de mudar a ordem das implementações, a primeira utilizada para implementar esse caso de uso foi Javascript. Para esse caso de uso, o autor pensou em uma estrutura onde a aplicação principal tivesse as estruturas de dados contendo os produtos cadastrados. Logo, a função principal do módulo de produtos recebe como parâmetro essa estrutura e retorna uma estrutura atualizada com os novos produtos. Foi escolhido separar a consulta de cada propriedade em funções diferentes e fazer com que a criação do produto final fosse uma composição dessas funções. O resultado final foi uma função simples e de fácil legibilidade.

Apesar disso, houve certa dificuldade ao ler os dados da entrada por que a biblioteca utilizada para tal fim era assíncrona. Foi gasto muito tempo para identificar isso, porém depois de identificado e trocado por uma operação síncrona, a implementação foi realizada sem grandes problemas. A linguagem é dinamicamente tipada, então não foi possível definir um tipo esperado para as propriedades. Importante mencional que em Javascript é possível definir variáveis e atualizá-las, então o programador deve manter-se atento para não realizar comportamentos do tipo, já que ferem o princípio de transparência referencial. Além disso, a estrutura para definição de novos tipos é baseada em protótipo, onde um objeto é representado por um mapa de chave e valor. Simples, porém não muito seguro visto que não há uma compilação para validar se as propriedades estão sendo utilizadas corretamente. Javascript também dispõe de

diversas estruturas de dados para uso, como lista, mapa, conjunto. Utilizá-las foi muito simples e se mostraram ferramentas poderosas.

A próxima linguagem foi Haskell. Diferente do primeiro caso de uso, essa foi a que apresentou maior dificuldade. Devido à sua natureza de separar a parte impura do código, acabou por levar quase quatro horas, mais que o dobro que Clojure, por exemplo, por diversos erros de compilação referentes a operações impuras misturadas com operações puras. Resolvê-las foi muito complexo e desestimulante. Apesar de ter um sistema de tipos estático e uma interface simples para criação de tipos compostos, a implementação teve que mudar de arquitetura e a separação em módulos teve que ser transformada em um só arquivo, visto que todos os módulos lidavam com entrada e saída de dados.

Haskell oferece uma estrutura de dados em lista muito poderosa. Ela é separada em cabeça e corpo e o desenvolvedor é encorajado sempre a utilizá-la de maneira recursiva. É uma estrutura pura que não permite modificações, sendo necessário criar uma nova lista à cada modificação.

Por último, Clojure foi uma grata surpresa. Como não tem os problemas de IO que Haskell tem, foi bastante simples desenvolver este caso de uso. Até mesmo a criação dos tipos foi simples. A linguagem, assim como Javascript, é dinamicamente tipada, oferecendo os mesmos riscos que a primeira. Oferece um bom conjunto de estruturas de dados, tendo uma arquitetura semelhante à Haskell nesse sentido, o que é um destaque pela facilidade.Assim, Clojure é identificada também como a melhor linguagem para implementar esse caso de uso. Os dados quantitativos coletados, por exemplo, mostram que foi a linguaguem mais rápida para desenvolver o caso de uso.

## 4.3 Caso de Uso 3

Por precisar realizar buscas em estruturas de dados, esse caso explora bastante a iteratividade. Linguagens funcionais não costumam oferecer laços como as imperativas (for e while), sendo utilizada a recursão para esses casos.

Clojure foi a primeira linguagem, dessa vez. Definição de função de fato foi um pouco mais difícil em Clojure. Foi sentido falta de itens mais poderosos, como por exemplo um casamento de padrão onde seria possivel selecionar uma função para parâmetros iguais. Isso atrapalhou, por exemplo, na função de busca, que apesar de lidar fácilmente com estrutura de dados (no caso, uma lista), sua implementação não ficou tão trivial. Outro destaque negativo é o fato de não ser permitido chamar uma função que está definida posteriormente. Clojure usa uma estrutura bottom up, onde uma função só pode realizar chamadas para funções que foram definidas antes dela. Alémd isso, Clojure oferece apenas a recursão como ferramenta de iteração. Porém, este restrição não foi impedidditivo para a implementação.

Em Javascript, lidar com a estrutura de dados da linguagem foi algo simples e direto. A função *find* de array foi muito útil e foi de longe a mais fácil de se implementar, e a mais rápida também. Houve uma maior dificuldade, porém, para fazer um código puramente funcional nessa linguagem. Primeiro, não há garantia de transparência referencial na linguagem. Logo, estruturas de dados são mutáveis e o usuário precisa sempre ter o cuidado de criar uma nova cópia caso queira realizar alguma alteração. Além disso, definir função em Javascript é feito de maneira muito simples, apenas com o uso de argumentos. Casamento de padrões ou sobrecarga paramétrica não é uma opção nessa linguagem, dificultando a legibilidade. Além disso, o programador precisa ficar o tempo inteiro fixado em não cometer vicios de linguagem imperativa, como uso de laços, variáveis, estado.

Por último, Haskell apresentou alguns pontos negativos. A facilidade que a linguagem apresenta para lidar com listas através da ideia de cabeça e corpo foi perdida devido o fato de ter sido utilizado uma lista de String, que por si, é uma lista de char. Esse comportamento acaba por confundir o programador, pois este imagina que ao nomear a cabeça e o corpo da lista, esta irá fazer para a lista passada, não para o primeiro elemento dela, se este for uma lista. Apesar disso, a função foi simples de criar.

Javascript, para essa demanda, se mostrou extremamente poderosa. Sua sintaxe simples permitiu que o código fosse criado rapidamente (foram necessários apenas 16 minutos para implementar o caso de uso em Javascript, enquanto Haskell, a mais próxima, levou 74). É importante relembrar aqui apenas a necessidade de atenção ao desenvolver esse caso de uso em Javascript pelas construçõe da linguagem que permite reescrever o código de maneira procedural.

## 4.4 Caso de Uso 4

O caso de uso mais simples do estudo, requeria bastante o uso de iteração pois era necessário percorrer toda a lista de produtos cadastrados. Sem exceção, a implementação foi simples considerando todas as linguagens. Não há distinção de complexidade pois todas oferecem estruturas parecidas e simples para a implementação. Em Haskell, bastou uma simples função que imprime os detalhes de um produto enquanto a lista não está vazia. Clojure seguiu exatamente a mesma lógica e sem diferenças significativas. Para Javascript, foi utilizado um iterador que permite aplicar uma função para cada elemento da lista. Foi simples e rápido de implementar. Analisando o código produzido, Haskell e Clojure teriam uma abordagem mais puramente funcional, pois tratam a iteração dos dados através de uma função recursiva, enquanto Javascript o uso de um iterador que atualiza uma variável. Destaco a eficiência, mais uma vez, de Javascript. Para este caso de uso, foi a melhor linguagem, a que tomou menos tempo e não houveram erros de compilação nem necessidade de consultas à especificação. Para efeito de comparação, segue abaixo a implementação do caso de uso 4 nas três linguagens.

A seguinte tabela apresenta a relação entre a linguagem de programação escolhida para cada caso de uso:

| UC \Ling. | Haskell | Clojure | Javascript |
|---|---|---|---|
| 1 | | X | |
| 2 | | X | |
| 3 | | | X |
| 4 | | | X |

## 4.5 Análise Geral

É fato que cada linguagem de programação tem um foco em resolver algum tipo de problema de maneira mais fácil. O design da linguagem e as decisões tomadas durante a concepção delas visam atingir um objetivo específico mas que ao mesmo tempo seja genérico o suficiente para ser aplicado em diferentes contextos.

Assim, ao final do estudo foi possível perceber em Javascript uma vocação para aplicações simples. É muito rápido desenvolver algo nessa linguagem. Como é possível analisar na tabela, Javascript foi a linguagem mais rápida para fazer o experimento. Por ter uma sintaxe simples e mais parecida com a programação procedural, ela também foi a que menos teve necessidade de consultas à documentação durante a fase de implementação. Para aprender programação funcional, porém, não parecer ser linguagem apropriada. Especialmente no contexto dos alunos da disciplina de PLP, que inicam o a disciplina com conhecimentos prévios do paradigma imperativo. É difícil programar puramente de maneira funcional em Javascript e a possibilidade de poder usar uma abordagem imperativa acaba atrapalhando o processo de aprendizado. O paradigma funcional preza pela criação de funções genéricas e a reutilização delas para realizar lógicas mais complexas. Porém, definir uma função em Javascript não permite abstrações mais poderosas, como o casamento de padrões, empobrecendo o código.

Já Haskell, a linguagem que é atualmente usada na disciplina, na nossa opinião, seria uma ótima escolha para aprender a programar no paradigma funcional se não fosse a maneira pouco ortodoxa de lidar com os dados de entrada e saída, tão utilizados na disciplina. Definir funções nela é uma tarefa simples e poderosa. O casamento de padrões ajuda a criar funções de fácil leitura e compreensão, com um código mais simples e prático. As estruturas de dados da linguagem são poderosas e compreender a lógica de funcionamento delas é natural. Ser uma linguagem estaticamente tipada também facilita a compreensão do sistema produzido e fornece uma segurança durante a evolução da aplicação. Como dito, o lado negativo da linguagem é a maneira como ela lida com entrada e saída. Perde-se muito tempo tentando estruturar a aplicação para esse tipo de dado, é complexo, desestimulante e muitas vezes o autor passava mais tempo tentando corrigir erros nesse sentido do que com aspectos da programação funcional em si. Pelas métricas coletadas, é possível ver a grande diferença de tempo de implementação geral quando comparada com Javascript e até mesmo com Clojure. Suas mensagens de erro não são tão claras, fazendo com o que o autor tivesse a necessidade de pesquisar várias vezes sobre a linguagem durante o experimento. Isso atrapalha bastante o processo de aprendizado e acaba por desestimular o aluno.

Clojure, por outro lado, apresenta uma sintaxe um pouco menos legível que Haskell. Seu escopo por parênteses pode parecer estranho à primeira vista. Os número de consultas a documentação da linguagem mostra bem esse impacto inicial. O autor precisou de várias consultas para compreender como funcionava certos aspectos da linguagem, em especial nos casos de uso 1 e 3, que lidavam com o inicio de aprendizado na linguagem e com iteratividade, respectivamente. Porém, depois do primeiro impacto, a linguagem mostrou-se muito simples de compreender e de escrever código. A definição de função para diferentes valores de entrada é mais complexo do que em Haskell, de fato. Porém, depois que é compreendido como funciona, se mostra efetiva e poderosa. Por não ser pura, a linguagem mostra-se flexível o suficiente para que alguém que esteja aprendendo o paradigma funcional implemente seu problema de maneira declarativa, porém sem perder tempo com aspectos de efeito colateral. É uma excelente linguagem e aprender programação funcional nela une a poderosa definição de função que linguagens tipicamente funcionais oferece, a simplicidade de lidar com entrada e saída de dados e as restrições de mutabilidade de valores da programação funcional. Assim, é destacado Clojure como uma boa opção para quem quer aprender o paradigma funcional.

## 5 QUESTIONÁRIO COM ALUNOS

A fim de complementar nossa investigação e validar com os alunos aspectos como legibilidade de código e corretude, foi aplicado um questionário nas duas turmas de PLP do período 2019.2 consolidando um total de 42 respostas. Vale destacar que tais alunos estavam no processo de aprendizado do paradigma funcional. A legibilidade é uma métrica importante pois permite obter informações de pessoas que não conhecem as linguagens acerca do quão legível é o código produzido nelas à primeira vista, o que é sempre útil em se tratando do processo de aprendizado.

No questionário, foram apresentadas dois conjuntos de questões. No primeiro (questões 1, 2 e 3), foi dada uma especificação e apresentada a implementação dela nas três linguagens trabalhadas (Haskell, Clojure e Javascrit). A implementação era referente a parte do primeiro caso de uso, onde deveria ser implementado uma função que, dependendo de um valor do parâmetro recebido, deveria-se despachar a chamada para a opção correspondente. Nesse contexto, foi questionada aos alunos a legibilidade do código (questão 1), acerca de quão fiel a implementação estava da especificação (questão 2) e o quão simples seria evoluir o código para adicionar uma nova opção no menu (questão 3).

A segundo conjunto (questões 4, 5 e 6), o processo foi inverso. Foram mostrados três códigos escritos nas três linguagens e pedido para que descrevessem o comportamento (especificação) da implementação (questão 4). Vale mencionar que três os códigos eram referentes a mesma especificação. Além disso, os alunos foram questionados sobre os mesmos tópicos da questão anterior (legibilidade e facilidade de evolução - Questões 5 e 6).

As perguntas foram respondidas numa escala de 1 a 5, onde 1 significa "Discordo Plenamente" e 5 "Concordo Plenamente". Ao final, foi pedido que o aluno fizesse uma avaliação pessoal acerca das linguagens utilizadas. O questionário aplicado está disponível para acesso no nosso site[3].

### 5.1 Resultados e Discussões

**Pergunta 1**: O código da função é legível? Foi pedido para os alunos avaliarem de 1 a 5 a afirmação, sendo 1 - "Discordo Completamente" e 5 - "Concordo Plenamente":

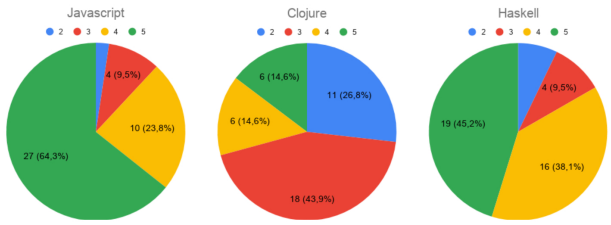


**Figure 3: Pergunta 1**

[3]https://github.com/antunesdantass/tcc

Confirmando as impressões do autor durante o estudo, o código em Clojure pareceu aos alunos menos legível, muito provavelmente devido a maneira como a função foi definida, com o casamento de padrões de Clojure. Já os códigos em Haskell e Javascript foram mais legíveis, especialmente este último devido a uma certa semelhança com código procedural.

**Pergunta 2**: A implementação está fiel à especificação do sistema? A escala é a mesma da questão anterior.

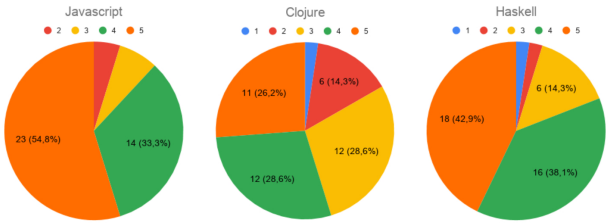


**Figure 4: Pergunta 2**

Aqui, é possível perceber como os alunos acharam simples a sintaxe de Javascript. A construção mais semelhante a de uma linguagem procedural fez com que eles sentirem mais familiarizados. O código em Haskell também foi bem interpretado pelos alunos, sendo avaliado como muito fiel à especifação. A implementação de Clojure também estava fiel à especificação, porém muitos alunos concordam plenamente com isso. É possível inferir que a maneira como a função foi definida, mais complexa e diferente do que os alunos estejam acostumados, tenha tido impacto nisso, uma vez que ele não conseguiam avaliar a corretude da implementação por não compreender plenamente o que foi feito.

**Pergunta 3**: Solicitada uma alteração básica, você coseguiria modificar o código sem dificuldades?

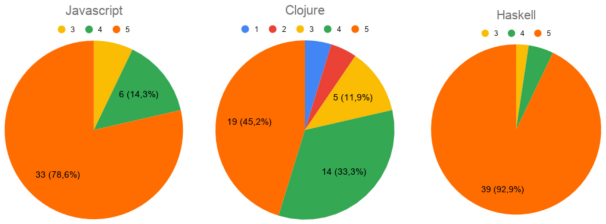


**Figure 5: Pergunta 3**

O comportamento se repete, com os alunos achando quase unanimemente simples de implementar a mudança em Javascript e em Haskell. Já em Clojure, os alunos sentiram uma maior dificuldade de imaginar como evoluir o código.

**Pergunta 4**: Foi pedido aos alunos que eles falassem, comparativamente, suas opiniões sobre a implementação do caso de uso um nas três linguagens. Os comentários deles também seguiu os resultados das perguntas anteriores, onde eles acharam complexa a implementação em Clojure e simples em Haskell e Javascript (onde X é Haskell, Y é Clojure e Z é Javascript): "A linguagem X possui interpretação mais fácil e creio que a alteração no código também. Em comparação a Y é mais complexa." "Linguagens X e Z com fácil compreensão e fácil de se implementar, diferente da Y, e entre as linguagens X e Z, aparenta-se uma maior praticidade na X."

**Pergunta 5**: Para esta pergunta, foi mostrada a implementação do cadastro de produtos em cada linguagem e sugerida uma modificação. Os alunos deveriam avaliar quão fácil seria realizar a mudança.

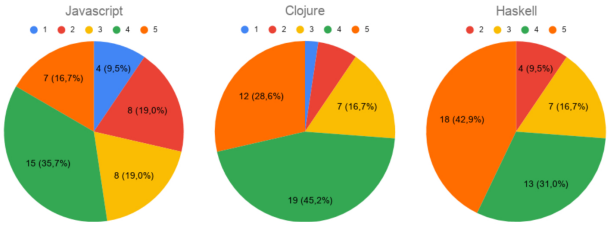


**Figure 6: Pergunta 5**

Para esta implementação, o resultado foi diferente do caso de uso abordado anteriormente. Aqui, Haskell e Clojure foram as linguagens que os alunos se sentiram mais confortáveis para realizar a evolução, enquanto Javascript a que eles sentiriam mais dificuldade. É possível inferir que os alunos tem dificuldade com agregação de função, recurso poderoso da programação funcional que foi utilizada em Javascript para a implementação mas não em Haskell e Clojure, que abordaram uma estrutura mais sequencial.

**Pergunta 6**: Semelhante a primeira questão, foi perguntado aos alunos o quão legível o código estava.

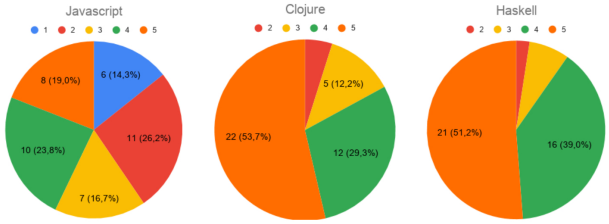


**Figure 7: Pergunta 6**

Aqui, o cenário da pergunta anterior se repete. Os alunos acharam os códigos em Clojure e Haskell bastante legíveis, enquanto em Javascript houve uma grande variedade de opiniões. Isso corrobora com a afirmação anterior acerca do uso de agregação de função, além de mostrar uma deficiência dos alunos com relação a capacidade de abstração.

**Pergunta 7**: Aqui foi pedido para os alunos comentarem sobre a implementação do caso 2 nas três linguagens. Por exemplo, um dos alunos achou a agragação de função utilizada em Javascript complexa, enquanto os códigos de Haskell e Clojure foram mais sequênciais (a linguagem A é Javascript e as linguagens B e C são Clojure e Haskell respectivamente): “A parece confuso e menos intuitivo. B e C são organizadas e de fácil percepção do que está sendo feito, apesar de mais extensos.”

## 6 TRABALHOS RELACIONADOS

A literatura apresenta experimentos acerca dos ganhos do uso de programação funcional com relação a outros paradigmas, como o Orientado à Objetos. Em [7], é comparado o desenvolvimento de algoritmos nesses dois paradigmas, avaliando a qualidade do código produzido. Foi concluído que o código produzido em linguagem funcional teve uma taxa de reuso 1,5 vezes maior que numa linguagem orientada à objetos.

Já [4] avalia o desempenho dos alunos que aprenderam programação funcional em disciplinas introdutórias em cursos de Ciência da Computação e Engenharia da Computação. Esse estudo mostrou que os alunos tiveram uma melhora considerável na resolução de problemas em linguagens funcionais do que em procedurais, tendo diminuído inclusive o índice de evasão dos cursos.

Já o estudo de [11] avalia se a escolha da primeira linguagem de programação a ser ensinada em um curso de Programação tem impacto nos resultados dos alunos. Após um estudo analítico, é verificado que não há um ganho de desempenho na substituição da linguagem, o que infere que o importante são os conceitos a serem ensinados e o aprendizado destes, independente da linguagem.

Em [16], é avaliado um grande repositório de soluções para problemas computacionais implementados em diversas linguagens que pertencem aos mais utilizados paradigmas. Os autores buscaram avaliar aspectos como velocidade de execução, coesão da implementação, etc. Eles chegam a conclusão, por exemplo, de que linguagens funcionais apresentam um código mais coeso e que linguagens estaticamente tipadas e compiladas tendem a sofrer menos erros (exceções) durante a execução.

Comparado aos trabalhos relacionados, este artigo acrescenta um relato de experiência, onde o autor buscou utilizar os papéis de professor e aluno para fornecer uma experiência mais próxima da vivenciada durante a disciplina, avaliando o processo de aprendizado seguindo os ítens do trabalho de [16].

## 7 CONSIDERAÇÕES FINAIS

Aprender uma nova linguagem de programação pode ser um desafio em diversos aspectos. Especialmente se essa linguagem for de um paradigma que o aprendiz não conhece. Aprender um novo paradigma representa uma mudança na maneira como alguém pensa durante o desenvolvimento de sistemas. A escolha da linguagem de programação que será usada nesse processo de aprendizado é de extrema importância, visto que ela fará a ponte entre as concepções da linguagem e o problema a ser resolvido.

Neste trabalho, realizamos um estudo comparativo entre linguagens de programação funcionais com respeito ao aprendizado/implementação dos aspectos básicos deste paradigma. O estudo realizado consistiu no desenvolvimento o mesmo sistema em três linguagens diferentes, observando aspectos chave das linguagens para esse processo. Neste estudo realizamos analises quantitativas e qualitativas. Além disso, foi realizado um estudo junto com alunos do período 2019.2 da disciplina Paradigmas de Linguagem de Programação da Universidade Federal de Campina Grande. Neste estudo, foram abordadas questões relacionadas à legibilidade do código produzido durante o experimento anterior.

Ao final, o autor realizou uma análise do que foi desenvolvido, do processo de desenvolvimento e das características do paradigma. Como conclusão, indicamos que Clojure seria uma melhor escolha para aprender o paradigma. Os problemas resolvidos pelos alunos da disciplina Paradigmas de Linguagem de Programação da UFCG são totalmente baseados em entrada e saída e as dificuldades impostas por Haskell, a linguagem utilizada atualmente na disciplina, pode atrapalhar bastante o processo de aprendizado, como atrapalhou o autor. Apesar da pesquisa ter indicado que, para o primeiro caso de uso, a implementação em Clojure não tenha sido tão legível, os alunos acharam o código feito para o segundo caso de uso legível e fácil de manter.

Como trabalho futuro, predendemos realizar um estudo com os alunos, ensinando-os Clojure ao invés de Haskell. Este estudo permitirá analisar qual o impacto da mudança e as opiniões dos alunos sobre a nova linguagem.

## REFERENCES


[1] 2019. Nubank promove conferência sobre linguagem Clojure. *Baguete* (Aug 2019). https://www.baguete.com.br/noticias/30/08/2019/nubank-promove-conferencia-sobre-linguagem-clojure

[2] Maria Cecília Calani Baranauskas. 1993. Procedimento, função, objeto ou lógica? Linguagens de programação vistas pelos seus paradigmas. *Computadores e Conhecimento: Repensando a Educação. Campinas, SP, Gráfica Central da Unicamp* (1993).

[3] MANUEL M. T. CHAKRAVARTY and GABRIELE KELLER. 2004. The risks and benefits of teaching purely functional programming in first year. *Journal of Functional Programming* 14, 1 (2004), 113–123. https://doi.org/10.1017/S0956796803004805

[4] Thais Helena Chaves de Castro, Alberto Nogueira de Castro Júnior, Crediné Silva de Menezes, Maria Cláudia Silva Boeres, and MCP Rauber. 2003. Utilizando programação funcional em disciplinas introdutórias de computação. *Anais do WEI* (2003).

[5] Chas Emerick, Brian Carper, and Christophe Grand. 2012. *Clojure Programming: Practical Lisp for the Java World.* " O'Reilly Media, Inc.".

[6] Michael Grady. 2010. Functional programming using JavaScript and the HTML5 canvas element. *Journal of computing sciences in colleges* 26, 2 (2010), 97–105.

[7] R Harrison, LG Samaraweera, Mark R Dobie, and Paul H Lewis. 1996. Comparing programming paradigms: an evaluation of functional and object-oriented programs. *Software Engineering Journal* 11, 4 (1996), 247–254.

[8] Juha Helminen and Lauri Malmi. 2010. Jype-a program visualization and programming exercise tool for Python. In *Proceedings of the 5th international symposium on Software visualization*. ACM, 153–162.

[9] Li-Ling Hu, Shian-Shyong Tseng, and Tsung-Ju Lee. 2013. Towards scaffolding problem-solving implementation process in undergraduate programming course. In *2013 IEEE 13th International Conference on Advanced Learning Technologies*. IEEE, 417–418.

[10] John Hughes. 1989. Why functional programming matters. *The computer journal* 32, 2 (1989), 98–107.

[11] Mirjana Ivanović, Zoran Budimac, Miloš Radovanović, and Miloš Savić. 2015. Does the Choice of the First Programming Language Influence Students' Grades?. In *Proceedings of the 16th International Conference on Computer Systems and Technologies (CompSysTech '15)*. ACM, New York, NY, USA, 305–312. https://doi.org/10.1145/2812428.2812448

[12] SJ Jones. 2001. *A brief informal history of the Computer Laboratory, University of Cambridge, Computer Laboratory.* Technical Report. Retrieved 2006/06/15, from http://www. cl. cam. ac. uk/UoCCL/misc/EDSAC99 ….

[13] Gustavo Jungthon and Cristian Machado Goulart. 2009. Paradigmas de Programação. *Monografia (Monografia)—Faculdade de Informática de Taquara, Rio Grande do Sul* 57 (2009).

[14] Gustavo Jungthon and Cristian Machado Goulart. 2016. Paradigmas de Programação. *Acesso em* 15 (2016).

[15] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. 2013. Learning computer science concepts with scratch. *Computer Science Education* 23, 3 (2013), 239–264.

[16] Sebastian Nanz and Carlo A Furia. 2015. A comparative study of programming languages in rosetta code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 778–788.

[17] C-H Luke Ong. 1992. The lazy lambda calculus: an investigation into the foundations of functional programming. (1992).

[18] Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. Learning and teaching programming: A review and discussion. *Computer science education* 13, 2 (2003), 137–172.

[19] Venkat Subramaniam. 2014. *Functional programming in Java: harnessing the power of Java 8 Lambda expressions*. Pragmatic Bookshelf.
[20] Simon Thompson. 1991. *Type theory and functional programming*. Addison Wesley.
[21] Simon Thompson. 1997. Where do I begin? A problem solving approach in teaching functional programming. In *International Symposium on Programming Language Implementation and Logic Programming*. Springer, 323–334.
[22] Peter Wegner. 1990. Concepts and paradigms of object-oriented programming. *ACM Sigplan Oops Messenger* 1, 1 (1990), 7–87.
[23] Patrick Henry Winston and Berthold K Horn. 1986. Lisp. (1986).