Desenvolvimento de uma aplicação web com linguagens funcionais puras

THALLES GUILHERME BOGAR PORTILHO

Universidade Federal de Uberlândia
Faculdade de Engenharia Elétrica
Curso de Graduação em Engenharia de Computação

Uberlândia
2021

THALLES GUILHERME BOGAR PORTILHO

Desenvolvimento de uma aplicação web com linguagens funcionais puras

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação da Faculdade de Engenharia Elétrica da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de bacharel em Engenharia de Computação.

Área de concentração: Engenharia de Computação

Orientador: Marcelo Rodrigues de Sousa

Uberlândia
2021

Dedico este trabalho a todas as pessoas que enxergavam este momento quando nem mesmo eu acreditaria mais que aconteceria. A todos os professores que durante a minha graduação, sempre exerceram seus papéis de forma invejável. Nunca, nenhum de vocês negou um pedido meu de ajuda e, por isso, sou eternamente grato.

Gostaria de dar um agradecimento especial ao professor e orientador deste trabalho, Marcelo Rodrigues de Souza, que mais de uma vez demonstrou compaixão e compreendimento em relação às dificuldades que tive em minha experiências acadêmicas e profissionais. Que em momentos onde seria normal considerar tais eventos como pífios, nunca os menosprezou, sempre de forma respeitosa. Um grande orientador, professor, coordenador e, acima de tudo, ser humano.

Por fim, dedico este trabalho às pessoas mais importantes da minha vida, minha família. Cada um de vocês contribuíram para minha chegada até aqui. Aos meus pais que batalharam de forma impecável para me dar as melhores oportunidades na vida, sempre visando minha felicidade e bem-estar.

Cada dia é uma nova vida, uma nova experiência.

Cada experiência é um degrau para o progresso da alma. Não fique preso ao passado. Você está, agora, diante de uma nova experiência. Dedique-se a ela de corpo e alma, e verá surgir o próximo degrau de evolução.

(Masaharu Tanigushi)

Resumo

Este trabalho apresenta o desenvolvimento de um jogo, denominado “Jogo do Peão”, em formato Web, totalmente desenvolvido em linguagens funcionais puras. Para sua concretização, foram utilizadas diversas tecnologias, com o intuito de colocar a eficácia destas em prova. Para a criação do *“Front-End”* foi utilizada a linguagem funcional Elm e para o *“Back-End”*, a linguagem Haskell, também pertencente ao grupo das linguagens funcionais. O “Back-End” engloba o servidor e uma inteligência artificial desenvolvida para realizar movimentos para um dos jogadores do jogo. Para o servidor, foi necessário a definição de todas as rotas e requisições que o mesmo pode aceitar, utilizando uma framework do Haskell, chamada Servant. A inteligência artificial foi implementada para realizar as jogadas do jogador branco, utilizando o algoritmo *minimax*.

Palavras-chave: Desenvolvimento WEB; Front-End; Back-End; Linguagens Funcionais; Inteligência Artificial; Servidor; Elm; Haskell; Aplicação WEB.

Abstract

This work presents the development of a game, called “The Pawn Game”, in Web format, fully developed in pure functional languages. For its implementation, several technologies were used, in order to put their effectiveness to the test. For the creation of the “Front-End” the functional language Elm was used and for the “Back-End”, the Haskell language, also belonging to the group of functional languages. The “Back-End” encompasses the server and an artificial intelligence designed to perform movements for one of the game’s players. For the server, it was necessary to define all the routes and requests it can accepts, using a Haskell framework called Servant. The artificial intelligence was implemented to perform the white player’s moves, using the minimax algorithm.

Keywords: WEB Development; Front-End; Back-End; Functional Languages; Artificial Intelligence; Server; Elm; Haskell; WEB application.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

API	Application Programming Interface
CSS	Cascading Style Sheets
DOM	Document Object Model
HTML	HyperText Markup Language
HTTP	Hyper Text Transport Protocol
JS	Javascript
JSON	JavaScript Object Notation
REPL	Read Eval Print Loop
URL	Uniform Resource Locator (Localizador Padrão de Recurso)
WEB	World Wide Web

Sumário

Capítulo 1

Introdução

Linguagens como JavaScript, jQuery, NodeJS e React são as mais conhecidas atualmente quando o assunto é o desenvolvimento de aplicações WEB, em específico, o conjunto gráfico das mesmas, chamado de *Front-End*. Apesar de serem as mais utilizadas, existem algumas necessidades dos desenvolvedores e usuários que ainda não são atendidas por nenhuma delas, e os mesmos acabam tendo que se acostumar e aprender a conviver sem que essas demandas sejam atendidas.

A linguagem Elm foi inicialmente projetada e criada por Evan Czaplicki para servir como ferramenta para a produção da interface gráfica, a parte que o usuário visualiza, de aplicações WEB. Seu objetivo era desenvolver uma linguagem capaz de, em boas mãos, criar aplicações que não apresentassem erros em tempo real, fornecer ao desenvolvedor mensagens amigáveis e legíveis dos erros de compilação e garantir que novos recursos pudessem ser adicionados à aplicação de forma rápida e segura. Elm pertence ao paradigma das linguagens funcionais e é essa característica em especial que permite com que a linguagem cumpra os objetivos traçados inicialmente pelo seu criador.

As linguagens funcionais são conhecidas por possuírem funções sem estado (*Stateless)* e variáveis imutáveis, estes dois conceitos fazem parte da linguagem Elm e são, em grande parte, responsáveis pela confiança e segurança que a linguagem fornece. Além disso, as linguagens funcionais, em grande parte, possuem um intérprete, ou interpretador de comandos, muitas vezes chamado de REPL. A linguagem Elm também herdou esta característica e possui o seu próprio REPL, que auxilia os desenvolvedores durante o trabalho fornecendo um ambiente de teste para as expressões da linguagem e funções do programa que estiver sendo desenvolvido.

Um ponto importante das linguagens utilizadas no desenvolvimento do Front-End de uma aplicação WEB é a compatibilidade dos navegadores a essas linguagens. Isso muitas vezes pode-se apresentar como um problema para o crescimento e disseminação das mesmas, pois, requer que o navegador seja compatível com essas linguagens. Elm não possui esse problema, pois, possui um compilador que transforma todo o código escrito em um código JavaScript equivalente, totalmente optimizado. Portanto, as aplicações construídas em Elm podem rodar em praticamente qualquer um dos navegadores atuais,

já que o JavaScript é uma linguagem extremamente consolidada.

Uma característica extremamente notória e que deu destaque para a linguagem Elm é o seu compilador, aclamado por toda a comunidade Elm e invejado pelas comunidades de outras linguagens. Com mensagens de erro excepcionalmente claras e objetivas, há relatos de que esta ferramenta passa a sensação de programar junto com um colega.

Apesar de tantos benefícios, Elm também possui alguns pontos negativos que levam a comunidade a questionar se realmente é uma boa opção de linguagem para se iniciar um projeto. Existem pessoas que acusam a documentação de ser um tanto quanto incompleta e desatualizada em alguns casos, que apontam a falta de um recurso na linguagem para "debug" como um grande problema e que, portanto, acreditam que a linguagem ainda não está pronta para ser utilizada.

1.1 Objetivo

O objetivo deste trabalho é demonstrar a eficácia de desenvolver uma aplicação WEB totalmente em linguagens funcionais puras e como isso pode ser uma ótima opção para quem busca maneiras de evitar problemas comuns em aplicações deste tipo como: dificuldade na manutenção e alteração do código, erros em tempo de execução e agilidade no desenvolvimento.

Para cumprimento deste objetivo, primeiramente serão apresentados os conceitos básicos da linguagem que servirão como embasamento teórico para demonstração das etapas no desenvolvimento de uma aplicação, com Front-End 100% feito em Elm. Nesta parte, serão apresentadas as ferramentas computacionais que possibilitam a linguagem Elm cumprir com suas premissas de criação.

Após a exposição da linguagem Elm, será apresentado a linguagem escolhida para compor o *back-end* da aplicação aqui desenvolvida, possivelmente conhecida por todos, e como ela se adéqua aos seus requisitos, a linguagem Haskell.

Finalmente, será demonstrado, de forma prática, como Elm lida com a criação de uma aplicação real utilizando os conceitos apresentados anteriormente. Esta aplicação consiste em um jogo de tabuleiro, chamado "Jogo do Peão", e nesta etapa serão apresentadas todas as escolhas e decisões que foram tomadas durante o desenvolvimento deste jogo.

1.2 Justificativa/Relevância

A linguagem Elm possui algumas particularidades que podem agregar muito para o futuro das aplicações WEB no mundo da computação, para a comunidade de desenvolvedores deste segmento e para todos os usuários. Dentre elas, pode-se citar:

- A linguagem possui um compilador considerado extremamente amigável. Os erros detectados são exibidos de forma clara, facilitando muito o entendimento e a

correção por parte do desenvolvedor. Isso faz com que o processo de criação da aplicação seja muito mais eficiente, já que não é necessário gastar muito tempo para entender o que deu errado. Na maioria das vezes, as mensagens informam até mesmo a solução para o problema.

- A comunidade é bastante ativa, o que é muito conveniente quando se precisa de ajuda em algum problema durante o desenvolvimento de uma aplicação.

- A linguagem Elm possui uma performance extremamente eficiente. As características e ferramentas utilizadas que possibilitam isso serão mostradas mais adiante neste trabalho.

- Possui uma sintaxe que pode parecer estranha no início, mas assim que o desenvolvedor se adapta, passa a ser bastante intuitiva e eficiente.

- A linguagem Elm é uma linguagem funcional pura, assim como a linguagem Haskell. Ela se apresenta como uma ótima opção para quem busca desenvolver uma aplicação totalmente com linguagens funcionais puras, utilizando Elm no *Front-End* e Haskell no *Back-End*.

Todos os pontos acima citados são positivos e podem contribuir imensamente com o futuro das aplicações WEB caso a linguagem se popularize e outras linguagens que surgirem ou já consolidadas, até mesmo em outros campos da computação, decidam seguir a mesma diretriz. O desenvolvimento WEB é um setor da computação extremamente importante nos dias atuais e é essencial possuir conhecimento deste assunto. Sendo assim, este trabalho irá contribuir para o aprendizado e familiarização deste setor que vem crescendo mais a cada dia e, além disso, para o estudo de uma linguagem promissora que pode acabar se popularizando e conquistando seu espaço no mercado, como já vem acontecendo em alguns países.

CAPÍTULO 2

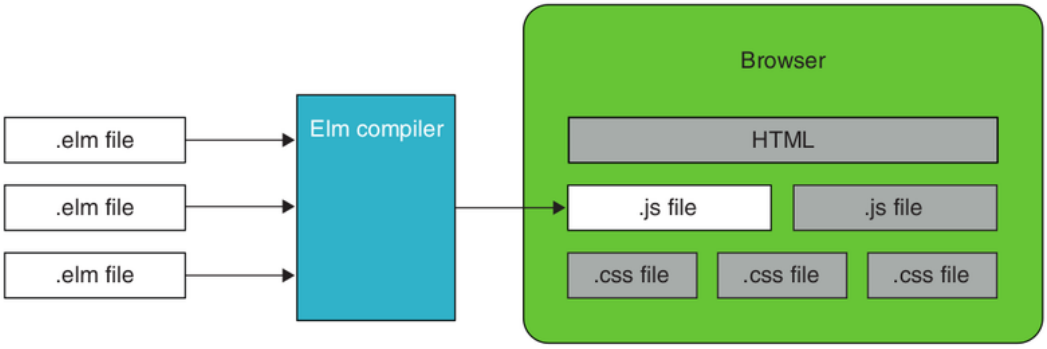
A Linguagem Elm

Uma aplicação WEB, ou sistema WEB, consiste em um *"software"* que está hospedado na *"internet"* e que pode ser acessado através de um navegador, este programa é composto por duas partes principais, que são o *Front-End* e o *Back-End*. O primeiro é o responsável por toda a parte gráfica da aplicação, ou seja, o que é visualizado pelo usuário no navegador. Para construir esta interface, o desenvolvedor do sistema conta com três linguagens: HTML (*HyperText Markup Language*), CSS (*Cascading Style Sheets*) e JS (*JavaScript*). *O HTML* é a ferramenta utilizada para criação dos elementos gráficos do programa, como, por exemplo botões, tabelas, textos e links. O CSS permite que o desenvolvedor consiga estilizar estes elementos alterando a cor, tipo da fonte de um texto e o posicionamento na página. O JavaScript é a linguagem que possibilita a interação do usuário com a aplicação definindo regras dinâmicas para ações do mesmo, como cliques em um botão ou a inserção de um texto em um campo de entrada de dados. Estas três linguagens, apesar de serem utilizadas em conjunto, possuem características de linguagens bem diferentes. A linguagem Elm é uma ferramenta que consegue englobar estes três componentes do *Front-End* em uma só linguagem, com mesma sintáxe, semântica e vocabulário e, portanto, apresenta-se como uma alternativa no desenvolvimento de aplicações WEB.

Atualmente, praticamente todos os navegadores possuem suporte a JavaScript, portanto, o que a linguagem Elm faz é compilar o código escrito em um arquivo elm, em um arquivo JS, que será então utilizado pela aplicação para ser executado pelo navegador. De forma simplificada, o compilador transforma todo o código escrito em Elm e o transforma em um código JavaScript optimizado, seguindo a risca uma determinada arquitetura de código que será descrita mais a frente neste capítulo. Além de fazer com que uma aplicação seja extremamente acessível, isso faz com que seja possível escrever partes da aplicação em Elm e partes em JavaScript, dependendo da necessidade do desenvolvedor, o que pode ajudar bastante durante a curva de aprendizagem do mesmo. Existem outras linguagens que também compilam seus códigos em JavaScript para serem executadas nos navegadores, como TypeScript e babel, porém, apesar de possuírem esta semelhança, a linguagem Elm possui arquitetura, sintáxe e semântica bem diferentes dessas linguagens também utilizadas

no desenvolvimento do *"Front-End"* de sistemas WEB.

Figura 1 – Esquematização da compilação de um programa escrito na linguagem Elm.



Elm in Action by Richard Feldman

2.1 Conceitos Básicos

Assim como toda linguagem, Elm possui alguns conceitos que são muito importantes para o entendimento de como a linguagem trabalha. Nesta parte do trabalho, serão apresentados quais são estes conceitos e como eles influenciam no desenvolvimento do código.

2.1.1 Expressões

Como dito anteriormente, Elm é uma linguagem funcional, e como todas as linguagens funcionais, as expressões são o componente mais básico e primitivo da linguagem. Por definição, expressão é um conjunto de números, símbolos e operadores que, ao serem avaliados, resultam em um valor (ELM PROGRAMMING, b). Em linguagens funcionais e, portanto, em Elm, todos os cálculos necessários para formar a lógica do programa são feitos através de expressões. Estas são posicionadas de forma estratégica pelo programador, todas em conjunto, para formar assim uma aplicação. Logo, é correto afirmar que um código escrito em Elm, ou qualquer outra linguagem funcional, é uma grande expressão composta de várias mini expressões, não importando qual a sua complexidade.

Tabela 1 – Exemplos de Expressões em Elm.

Expressão	Resulta em
“Hello, ” ++ “World!”	“Hello, World!”
2 ^ 4	16
42	42
2 + 3	5
2 * 3	6
11 / 2	5.5
11 // 2	5 (exclui a parte decimal)

2.1.2 Constantes

A maioria das linguagens possui o conceito de variáveis, que representam uma maneira de armazenar um valor para que o mesmo possa ser utilizado posteriormente no código. Na linguagem Elm, as variáveis são definidas apenas no escopo local de uma função, ou seja, seus valores só podem ser acessados de dentro da função onde elas estão declaradas. Além disso, as variáveis são imutáveis, isso significa que uma vez atribuído um valor a ela, não é possível alterar ou atribuir um valor diferente para aquela variável, dentro do mesmo escopo. Uma vez que a função onde a variável foi declarada termina sua execução corrente e inicia uma outra execução, ou seja, muda de escopo, aquela variável pode receber um novo valor.

Essa característica de imutabilidade faz com que o nome “variável” torne-se um tanto quanto incoerente com o conceito e, portanto, Elm escolheu chamá-las de “constantes” (ELM PROGRAMMING, a). Outra implicação desta imutabilidade, é que em Elm não existem variáveis globais. Qualquer valor, que esteja no escopo global no código, com um nome associado a ele é uma função que obrigatoriamente deve retornar uma expressão. O fato de não precisar se preocupar que uma simples alteração de código possa alterar o valor de uma variável global, ou até mesmo de uma variável de escopo local, faz com que seja extremamente mais fácil e ágil realizar modificações no programa, além de prevenir muitos erros que podem ocorrer com esse tipo de prática que em outras linguagens, como JavaScript, é permitida.

2.1.3 Funções

Outro componente imprescindível da linguagem Elm são as funções. Não somente as funções propriamente ditas, mas também o conceito que elas possuem dentro da linguagem. Em elm, as funções nada mais são do que um conjunto de expressões organizadas para

formar uma lógica computacional, conjunto este atribuído a um nome obrigatoriamente iniciado com letra minúscula, que é o nome da função, para que essa lógica possa ser reutilizada quantas vezes forem necessárias no código, apenas chamando a função pelo nome atribuído a ela (FELDMAN, 2020). Para definir uma função, basta escrever o nome, seguido pelos seus parâmetros (caso houver) e, após o sinal de igual, escrever a expressão que a define.

As funções nesta linguagem possuem algumas peculiaridades que valem a pena ressaltar. A primeira delas é que elas são imutáveis, isso significa que uma função irá retornar sempre o mesmo valor, dados os mesmos parâmetros. Esta é uma característica comum entre as linguagens funcionais, que faz com que uma função não produza, ou acarrete, efeitos colaterais no programa, são as chamadas funções puras. Outra peculiaridade das funções em Elm é a tipagem dos dados que elas recebem e retornam. Este recurso não é obrigatório, ou seja, o desenvolvedor não é obrigado a definir os tipos de dados da função, mas está disponível caso ele opte por fazê-lo. Vale ressaltar que é essa é uma boa prática de programação da linguagem extremamente recomendada, pois facilita para o compilador encontrar erros de tipagem no momento da compilação e alertar sobre o problema de forma extremamente clara e amigável, como será mostrado mais a frente neste trabalho. Ao garantir que uma função não causará efeitos colaterais no código com estes conceitos de funções puras e imutabilidade das mesmas, e além disso, incentivar uma boa tipagem dos dados, a linguagem Elm caminha para cumprir um dos seus objetivos de criação de produzir aplicações que não produzam erros em tempo de execução.

Figura 2 – Definição e teste da função "positivo", onde "positivo" é o nome, "num" é o parâmetro e "num > 0" é a expressão que define a função.

Em Elm, assim como em outras linguagens funcionais, não é necessário declarar um valor para ser retornado pela função com, ela irá retornar sempre o resultado da expressão designada a ela.

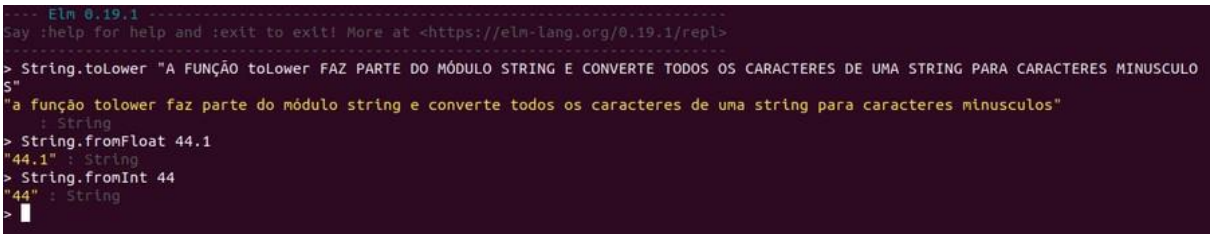
2.1.4 Módulos

A medida que um programa vai se tornando maior e mais complexo, é uma prática comum começar a separar parte dos códigos em diferentes arquivos, que podem ser

chamados de módulos. Um módulo, ou biblioteca, nada mais é do que um conjunto de funções que podem ser importadas no arquivo principal do código através de uma simples declaração no início do arquivo, concedendo ao arquivo acesso às funções daquele módulo importado, para serem utilizadas em tarefas específicas, comuns de aparecerem durante a programação.

Este conceito de módulo é algo extremamente comum entre as várias linguagens de programação que existem, não é algo inovador introduzido pela linguagem elm. Porém, vale a pena comentar que existem diversos módulos prontos, muitos deles inclusive já vem instalados por padrão junto com a linguagem, com funções para manipulações dos principais tipos de dados da linguagem. Existem módulos para se trabalhar com vetores gráficos, para realizar requisições HTTP e módulos para facilitar a criação de layouts e interfaces. Essa é uma pequena vantagem do elm, praticamente tudo que for necessário durante o desenvolvimento de uma aplicação, provavelmente existe um módulo com funções que podem auxiliar o desenvolvedor durante o trabalho. O módulo "String" é um módulo dos módulos que já vem instalados com a linguagem e fornece diversas funções que auxiliam na manipulação de Strings.

Figura 3 – Utilização de algumas funções do módulo "String" para realizar operações básicas com strings.



Outra curiosidade em Elm é que existe uma função que converte um dado para String, para cada tipo de dado. Em JavaScript, por exemplo, existe o método "toString" que faz a conversão de qualquer tipo de dado para String, isso abre margens para que o desenvolvedor, acidentalmente, passe como parâmetro para a função "toString" outra função qualquer, o que causaria um erro em tempo de execução para o usuário, já que a linguagem JavaScript não possui proteção contra isso. Já na linguagem elm, se o dado o dado a ser convertido for um Float, utiliza-se "fromFloat", se for Int utiliza-se "fromInt" e assim por diante, o que garante que o argumento correto seja passado para a função. Se for passado um tipo de dado Float para a função "fromInt", o compilador irá detectar este erro antes que chegue para o usuário de uma forma que comprometa a experiência do mesmo com a aplicação. Esse exemplo mostra a postura que a linguagem adota, não apenas no módulo "String", mas em toda sua concepção, para garantir que não haverão erros no momento de execução do código.

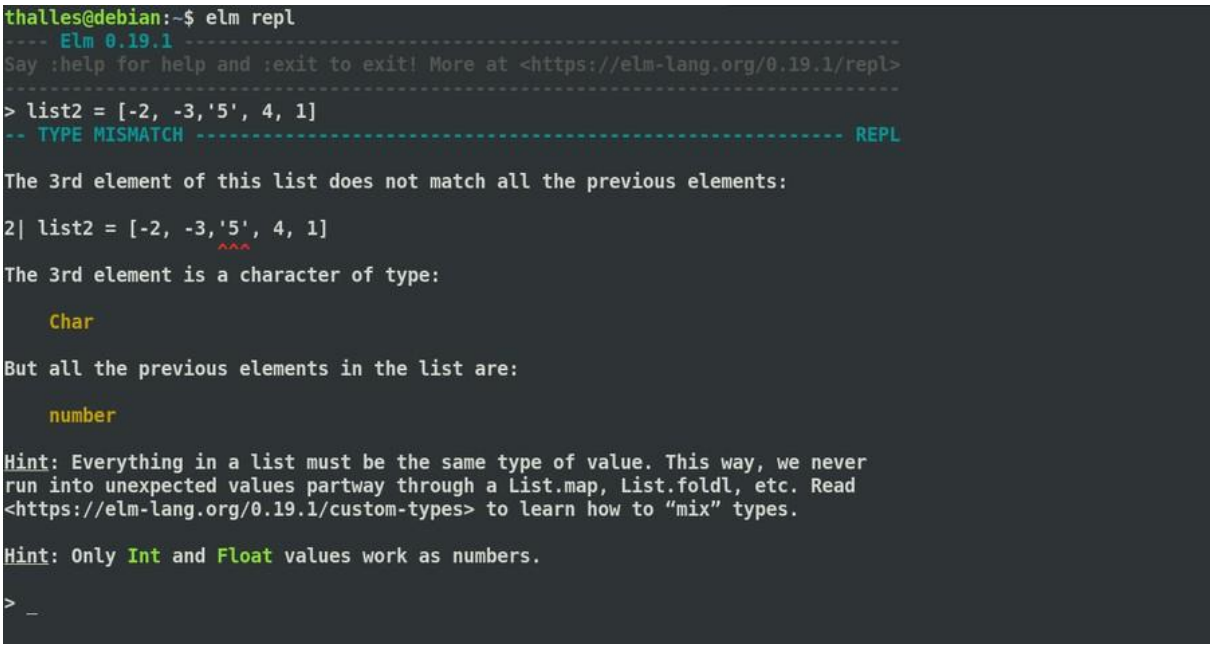
2.1.5 Listas

As listas, assim como em todas as linguagens funcionais, é uma das estruturas de dados mais utilizadas na linguagem elm. Sua sintaxe é bem parecida com um Array em JavaScript, inicia-se com um colchete e é possível acessar seu primeiro elemento, a quantidade de elementos que ela possui e realizar iterações em cada elemento seu. Diferente de Arrays em JavaScript, as listas em Elm são imutáveis, assim como as constantes e as funções, e são linkadas, isso significa que existe um ponteiro que aponta para onde a lista começa na memória e, portanto, não é possível acessar seus elementos através de seu index, é necessário realizar algumas manobras caso seja necessário acessar o valor de um elemento da cauda dessa lista.

Diferentemente dos arrays em JavaScript, as listas na linguagem elm não podem conter 2 tipos de dados diferentes, ela pode ser uma lista de String, uma lista de Int ou de Float, mas não pode ser uma lista de Int e Float ao mesmo tempo, o compilador reconhece isso como erro e não compila o código. Imagine que durante o desenvolvimento de um programa em JavaScript, é necessário filtrar os elementos de um array pegando apenas os números que são maior do que zero. Suponha que o array a ser filtrado seja "[-2, -3, '5', 4 , 1]", não há como prever o que acontecerá, pois, existe um elemento dentro do array que é impossível saber se é maior que zero ou não, pois não se trata de um número, mas sim, um carácter. Na linguagem Elm não existe esse problema, pois não é possível criar uma lista onde os elementos possuem tipos de dado diferentes, esse é mais um dos motivos pelo qual a linguagem consegue produzir aplicações muito mais seguras e confiáveis.

Figura 4 – Exemplo da declaração de uma lista em Elm e utilização da função "filter" do módulo "List" para filtrar os elementos da lista que são maiores que zero.

Figura 5 – Tentativa de criação de uma lista com 4 elementos do tipo Int e 1 do tipo Char.



2.1.6 Registros

Os registros também estão entre as estruturas de dados mais utilizadas na linguagem Elm e consistem, basicamente, em uma coleção de campos com um valor associado a cada uma deles. Comparando com JavaScript, sua sintáxe e utilização se assemelha bastante com a dos objetos, porém como todo o resto da linguagem, possuem algumas particularidades. Para criar-se um registro, basta abrir chaves e declarar cada um dos seus campos, separados por vírgula, e depois fechar chaves. Assim como em JavaScript, para acessar os campos do registro, deve-se escrever o nome do mesmo e o nome do campo a ser acessado, separados por um ponto. A atribuição de valores para os campos do registro deve se feita utilizando o sinal de igual (=) e, para modificar o valor dos campos de um registro, basta escrever o nome do registro com o nome do(s) campo(s) a ser alterado(s) separando por um barra reta (|). Caso houver mais de uma campo a ser modificado, deve-se separá-los por vírgula.

Figura 6 – Exemplo da criação de um registro chamado “registro” com os campos “nome” e “idade”, como acessar o valor de seus campos e também como alterar esses valores.

Na imagem acima, é possível ver que após fazer a modificação dos valores dos campos do registro, foi retornado o novo registro com esses valores modificados, porém, ao pedir para imprimir na tela o registro, é possível ver que não houve alteração nenhuma no mesmo. Isso deve-se, mais uma vez, a imutabilidade da linguagem, assim como as listas, constantes e funções, os registros em Elm também possuem essa característica. Isso significa que, ao fazer a modificação de um registro, o que a linguagem realmente faz é uma cópia do estado anterior e retorna essa cópia. Isso pode parecer ineficiente em termos de memória, mas é um preço que linguagem escolheu pagar pela imutabilidade de seus dados, que é peça fundamental em prevenir os erros em tempos de execução. Para amenizar o custo de memória e tempo, a cópia criada reutiliza os valores do registro antigo que não estão sendo modificados, este conceito é conhecido na computação como pelo nome de “Estrutura Persistente de Dados” (GEEKSFORGEEKS, 2017).

2.1.7 Tuplas

As tuplas são uma estrutura de dados bastante similar com os registros, também são utilizadas para armazenar valores, que podem ser de tipos diferentes, porém, sem a necessidade de nomear tais valores. Basicamente, é uma forma mais simples e concisa de representar registros com poucos campos. Devido a essa grande similaridade, a tuplas podem conter no máximo três elementos, uma vez que caso seja necessário mais que isso, o recomendado é utilizar registros.

Assim como as listas, as tuplas também possuem um módulo chamado “Tuple” que já vem instalado por padrão e que possui algumas funções úteis para lidar com esse tipo

de dado. Por exemplo, para acessar o primeiro ou segundo elemento de um tupla, basta utilizar as funções “Tuple.first” e “Tuple.second” respectivamente.

2.1.8 Tipos de dados

Alguns tipos de dados são comuns em todas as linguagens de programação e, inclusive, já foram citados anteriormente neste trabalho, como String, Int e Float. Porém, em Elm, é possível criar tipos de dados personalizados e ainda restringir os valores que aquele tipo de dado pode aceitar. Essa funcionalidade é extremamente útil em situações onde um parâmetro de uma função, ou o retorno dela só pode ser, por exemplo, três valores diferentes. O desenvolvedor pode, na tipagem da função, especificar que aquele parâmetro, ou o retorno daquela função, é do tipo customizado criado por ele e, desta forma, qualquer valor que não seja igual aos que foram definidos na criação daquele tipo de dados, serão vistos como errados pelo compilador que irá apontar o erro. Esta funcionalidade faz parte dos recursos que a linguagem Elm possui para evitar erros em tempo de execução.

Os tipos de dados customizado, assim como todos os outros na linguagem Elm, devem iniciar com letra maiúscula e os possíveis valores que eles podem ter também.

Figura 7 – Criação do tipo de dados “Esportes” que pode ser igual a “Futebol” ou “Basket”.

```
thalles@debian:~$ elm repl
---- Elm 0.19.1 ------------------------------------------------------------
Say :help for help and :exit to exit! More at <https://elm-lang.org/0.19.1/repl>
----------------------------------------------------------------------------
> type Esportes = Futebol | Basket
> melhorEsporte : Esportes -> Esportes
| melhorEsporte x = x
|
<function> : Esportes -> Esportes
> melhorEsporte Futebol
Futebol : Esportes
> melhorEsporte Basket
Basket : Esportes
```

2.1.9 O Tipo de Dado “Maybe”

A palavra “Maybe” bem do inglês e sua tradução para português é “talvez”. Esse tipo de dado customizado deve ser utilizado em conjunto com algum outro tipo de dado já existente e serve para indicar quando um valor pode ser uma coisa, ou não, um tipo de dado “Maybe Int” significa que ele pode ser um número inteiro ou não. Isso é muito utilizado em casos de conversões de tipos de dados, por exemplo, ao tentar converter uma

String para Int utilizando a função “toInt” do módulo String. Se o programa, por algum descuido do desenvolvedor, tentar converter a String “teste” para Int, isso não será possível, pois, esta String é composta apenas de carácteres e, portanto, é impossível de converter para Int. É por isso que na definição da função “toInt”, seu retorno é definido como sendo do tipo “Maybe Int”, caso a string passada como parâmetro for passível desta conversão, será simplesmente retornado o valor convertido, mas se não for, será retornado o valor “Nothing”, que significa nada. Isso faz com que uma chamada com um parâmetro errado de uma função não cause efeitos colaterais em todo o código e, assim, evitando erros em tempos de execução.

2.1.10 Aplicação Parcial de Funções

Como dito anteriormente neste trabalho, as funções na linguagem Elm, assim como em todas as outras linguagens, recebem um determinado número de parâmetros e retornam um valor, ou expressão no caso da linguagem aqui estudada. Porém, existe mais uma característica que a diferença, que á possibilidade de invocar uma função sem passar todos os parâmetros necessários para a mesma. Neste caso, o que a linguagem Elm faz é retornar, para esta chamada, uma outra função que irá receber os demais parâmetros necessários e terminar sua execução. Para isso, dá-se o nome da computação de “Aplicação Parcial de Funções” (FELDMAN, 2020).

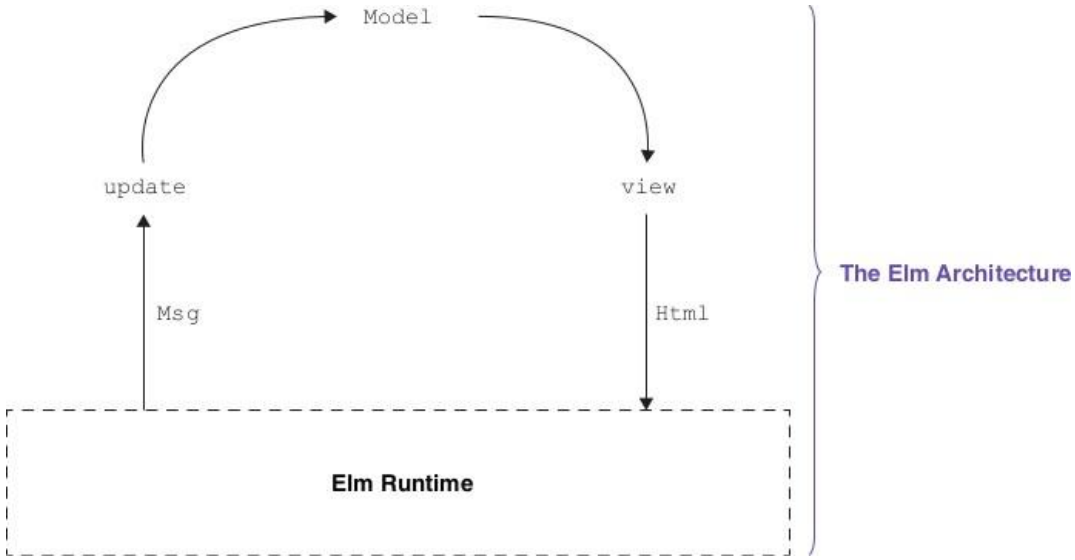
Este conceito já é amplamente conhecido na computação, a linguagem JavaScript, por exemplo, possibilita que isso seja feito utilizando as funções anônimas. A diferença está no fato de que as funções em Elm, por padrão, já possuem em sua definição este conceito, ou seja, caso o desenvolvedor da aplicação queria realizar uma aplicar uma função parcialmente, basta ele passar para esta função parte de seus argumentos e pronto, será retornado uma nova função que espera receber o restante de seus argumentos. Parece não haver um motivo muito satisfatório pelo qual a linguagem escolheu essa abordagem, o maior benefício disso está em conseguir escrever códigos mais concisos, que reaproveitam ao máximo o que já existe, sem repetir chamados ou parâmetros desnecessários, o que por sua vez, talvez torne o código mais sustentável em termos de futuras modificações e/ou melhorias.

2.2 Arquitetura da Linguagem

A arquitetura da linguagem Elm representa a forma com que a linguagem gerencia o fluxo de dados do programa para de fato produzir sua saída final, que de forma simplificada, é uma imagem na tela do navegador do usuário. Em Elm, ela é composta por três elementos

principais que são o “Model”, a função “Ppdate” e a função “View”, que serão descritas nos próximos módulos deste trabalho (FELDMAN, 2020).

Figura 8 – Esquematização de como a arquitetura da linguagem Elm gerencia o fluxo de dados do programa.



Elm in Action by Richard Feldman

2.2.1 Model

O Model, como o próprio nome já diz, é o modelo da aplicação e representa o estado atual da mesma. Ele deve ser definido em forma de registro, onde cada um de seus campos devem ser criados já pensando nas modificações que serão feitas na página durante a execução do programa, de certa forma referenciando os objetos e/ou elementos da página que serão alterados. O Model serve como parâmetro para as funções view e update que serão descritas a seguir.

2.2.2 View

A função view é quem abriga toda a parte gráfica da aplicação escrita na linguagem Elm, ela recebe o Model atual e retorna um código HTML que será utilizado para alterar o conteúdo da página. A linguagem Elm possui um módulo chamado “HTML” que contém

todas as funções para criação de elementos HTML utilizando a própria sintáxe da linguagem. Para os que já possuem conhecimento em HTML, será como aprender uma nova forma de representar seus elementos, e para os que não possuem, será como aprender esta linguagem de marcação com uma sintáxe igual a da linguagem que ela já está estudando. Essa é considerada uma parte bastante interessante da linguagem Elm, ela conseguiu juntar três linguagens com sintáxe bastante diferentes tudo em uma só, o que facilita muito o aprendizado.

Figura 9 – Exemplo da definição da função view

```
view : Model -> Html Msg
view model =
    div []
        [ h1 [ Html.Attributes.style "text-align" "center" ] [Html.text model.texto ]
        ]
```

A imagem acima mostra a função view de uma aplicação que apenas imprimi na tela o valor definido pelo campo “texto” do Model daquela aplicação, este valor pode ser alterado conforme o fluxo de dados do programa. Como tudo na linguagem Elm, os elementos da página são criados através de funções, no exemplo da imagem foram utilizadas as funções “div” e “h1”, ambas recebem duas listas como argumento e retornam uma expressão do tipo “Html Msg”. Concatenando as expressões retornadas por cada uma delas, tem-se o resultado final que é uma grande expressão formada de mini-expressões, como dito na sessão 2.1.1 deste trabalho: “Um código escrito em Elm, ou qualquer outra linguagem funcional, é uma grande expressão composta de várias mini expressões, não importando qual a sua complexidade.”

2.2.3 Update

A função update é a encarregada por realizar as modificações no Model na aplicação, de forma que o novo Model será enviado para a função view que reflete na página as alterações feitas. A função update deve receber como parâmetros o Model atual, que será modificado, e uma mensagem (Msg) que será utilizada para indicar qual a alteração que deve ser realizada naquele momento. O retorno desta função, será sempre o novo Model, ou estado da aplicação, com as modificações aplicadas.

A forma com que a linguagem Elm sinaliza essas modificações são através de eventos, como o clique de um botão pelo usuário ou a modificação do texto de um campo de entrada de dados na página. Esse evento automaticamente chama a função update passando um dados do tipo Msg para o mesmo, que irá realizar as modificações definidas por aquela mensagem enviada. Por exemplo, utilizando a função view criada no módulo anterior deste

trabalho, para alterar o texto exibido na tela, basta criar um botão com um evento de "onClick", que detecta o momento em que o usuário clicar nele, passando a mensagem "AlterarTexto" com o novo texto para a função update, que então realiza as modificações necessárias no Model da aplicação.

Figura 10 – Exemplo de uma aplicação completa escrita com a linguagem Elm.

```
type Msg
    = AlterarTexto String


type alias Model =
    { texto : String }


update : Msg -> Model -> Model
update msg model =
    case msg of
        AlterarTexto novoTexto ->
            { model | texto = novoTexto }


view : Model -> Html Msg
view model =
    div []
        [ h1 [ Html.Attributes.style "text-align" "center" ] [Html.text model.texto ]
        , button [] [ onClick AlterarTexto "novoTexto" ]
        ]
```

Com os três elementos da arquitetura da linguagem Elm apresentados até aqui: Model, view e update já é possível criar uma aplicação completa. Na imagem acima, foi criado um tipo de dado customizado chamado "Msg" que será utilizado para que a função update possa decidir qual modificação o programa deseja realizar no momento. Neste caso, o único valor que pode ser atribuído a um dado do tipo Msg é "AlterarTexto" que deve receber uma String. Foi criado um Model para a aplicação com apenas um campo chamado "texto" que também deve receber um valor do tipo String. A função view é a mesma utilizada como exemplo na seção anterior deste trabalho, porém, foi adicionado um botão com um evento de "onClick". O que esse evento faz é detectar o clique do botão pelo usuário e chamar a função update enviando ela a mensagem definida por este evento, no caso deste exemplo, a mensagem "AlterarTexto" juntamente com a String que representa o novo valor do texto. A função update então identifica a mensagem e realiza as alterações necessárias no Model da aplicação, que como explicado anteriormente, irá gerar um novo registro que será enviado para a função view que irá exibir esse novo modelo na tela.

2.2.4 Init

Como já foi apresentado até aqui, uma aplicação Elm começa sua execução pela criação do Model que é enviado para a função view e que renderiza um código HTML na página. Neste ponto, ainda não ocorreu nenhum evento que tenha enviado uma mensagem para a função update para que a mesma pudesse modificar o Model e enviar o novo Model para a função view. Sendo assim, o Model deve ser inicializado com os valores iniciais de seu campos e é para isso que serve a função init. É uma função bem simples que será executada logo após a definição do Model da aplicação e deve retornar o Model com seus valores iniciais para que a função view possa renderizar a página.

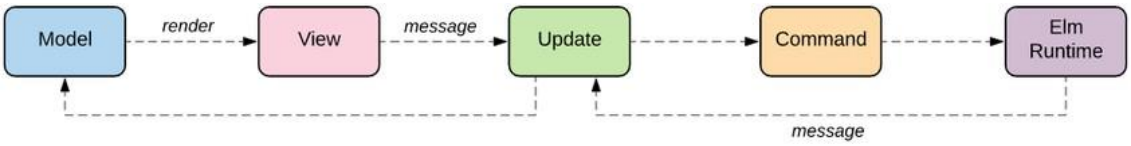
2.2.5 Comandos

Os efeitos colaterais no paradigma das linguagens funcionais são considerados como extremamente maléficos, pois, uma função que produz efeitos coletaterais é muito imprevisível e pode retornar valores diferentes para o mesmo valor de entrada passado a ela, dependendo do estado atual da aplicação. Já as funções puras, que não possuem efeitos colaterais, irão retornar sempre o mesmo valor para os mesmos valores de dados de entrada. Como já foi apresentado neste trabalho, o conceito de imutabilidade está muito presente na linguagem Elm, sendo utilizado em constantes, funções e em registros. O fato dos dados serem imutáveis traz o grande benefício de não ter funções que causam efeitos colaterais no código escrito. É dito que uma função que modifica uma variável de escopo global ou que realiza operações de entrada e/ou saída de dados, ou qualquer outro tipo de interação com o mundo exterior a ela, ou seja, qualquer parte de código e dados que não tenham sido criados dentro de sua própria definição, possui efeitos colaterais. Para o primeiro caso, a linguagem Elm, assim como as demais linguagens funcionais, resolve o problema não permitindo a criação de variáveis globais. Mas operações de entrada e saída de dados são realizadas durante todo o tempo de execução de uma aplicação, toda interação do usuário com a aplicação é uma entrada de dados, portanto, não seria possível simplesmente proibir esta prática. Portanto, faz-se a necessidade de realizar operações o mundo exterior de forma controlada e separada do resto do código e, para isso, a linguagem Elm criou os comandos.

A forma como a linguagem Elm introduz os comandos em sua arquitetura é definindo uma nova estrutura de retorno da função update, que agora irá retornar sempre um Model e um comando com sua respectiva mensagem. Essa mensagem é o mesmo tipo de dado utilizado em eventos na função view que chamam a função update para realizar alterações no Model da aplicação. Os comandos são chamados dentro da função update e sua execução causa uma nova chamada da função update com um nova valor de mensagem, que irá executar o que foi definido para ela de forma separada de onde aquele comando foi chamado,

desta forma, a linguagem consegue isolar as execuções que causam efeitos colaterais do restante do código.

Figura 11 – Fluxograma de uma aplicação com comandos escrita com a linguagem Elm.



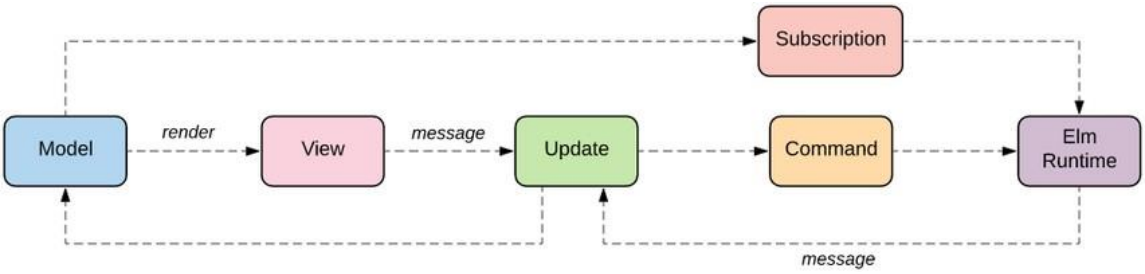
https://elmprogramming.com/commands.html#generating-random-numbers

Nesta nova arquitetura criada com a inserção dos comandos na aplicação, o Model é utilizado como modelo pela função view que irá renderizar os elementos HTML na página. Quando algum evento definido na função view ocorrer, será disparada uma mensagem para a função update que, irá modificar o Model e retornar o novo Model para função view que irá renderizar a nova página, ou irá executar um comando para realizar operações com o mundo exterior da aplicação. Este comando por sua vez, ao finalizar sua execução, irá enviar uma nova mensagem para a função update que segue o seu fluxo padrão. É importante ressaltar que os comandos são uma ferramenta necessária apenas se a aplicação desenvolvida necessite de comunicação com o mundo exterior, caso não haja essa necessidade, a linguagem Elm não obriga a utilização desta ferramenta para compilação do programa, diferente do Model, view e update, que são obrigatórios.

2.2.6 Assinaturas

Assim como os comandos, as assinaturas também foram criadas para realizar operações que lidam com o mundo exterior à aplicação. Basicamente, ela são utilizadas para detectar eventos na aplicação que não estão ligados a nenhum elemento da página, como o pressionamento de uma tecla do teclado, por exemplo, e executar algum trecho de código definido para aquele evento específico. Diferente dos comandos, as assinaturas recebem o Model como parâmetro, possibilitando que as ações definidas por elas sejam realizadas apenas se certas condições acerca do estado atual do Model forem cumpridas.

Figura 12 – Fluxograma com a introdução de assinaturas na arquitetura da aplicação.



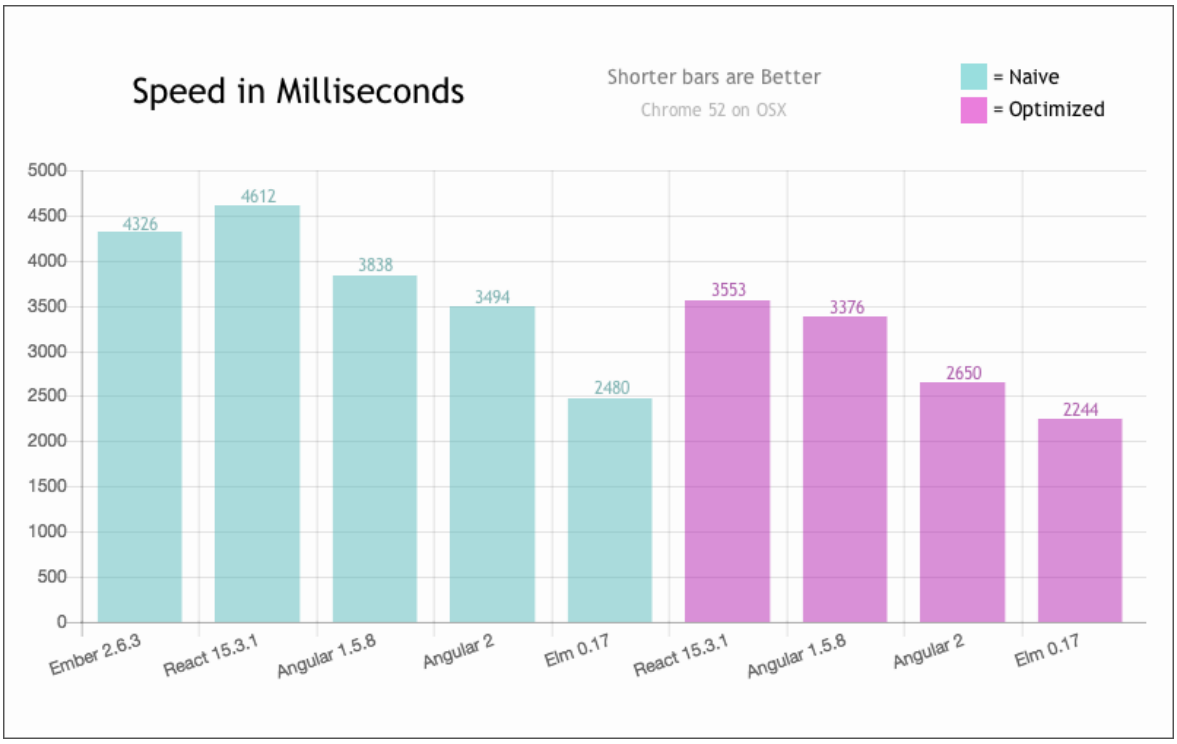
https://elmprogramming.com/subscriptions.html

Quando é criada uma assinatura para o programa, o mesmo passa a checar constantemente se ocorreu algum evento "cadastrado" na aplicação. Caso ocorra, o programa executa o trecho de código definido para aquele evento e envia uma mensagem para a função update que executa as modificações necessárias no Model da aplicação, casa houver. Assim como os comandos, as assinaturas são uma ferramenta para o desenvolvedor, sua utilização é obrigatória apenas se houver a necessidade.

2.2.7 DOM Virtual

Uma característica extremamente importante na arquitetura desta linguagem é o conceito de DOM virtual. A sigla DOM vem do inglês "Document Object Model" e é a representação em baixo nível de uma página exibida no navegador, em forma de nós e objetos. É através do DOM que as linguagens conseguem acessar determinados objetos e fazerem as alterações desejadas no mesmo. Em um modelo comum, as alterações na página são feitas modificando diretamente o DOM, que não é uma estrutura projetada para receber muitas modificações, mas sim, para exibir os elementos no navegador. Como dito anteriormente, a linguagem Elm trabalha com o conceito de DOM virtual, que nada mais é que uma versão do DOM real construída de forma extremamente optimizada para realizar alterações. A forma com que a linguagem faz isto é toda que uma requisição de modificação na página é realizada, o compilador cria um novo DOM virtual, faz as modificações neste e depois compara com o DOM virtual da versão, ou estado, anterior da página. O compilador então seleciona apenas as mudanças que foram realizadas e cria instruções de modificação do DOM real que serão realizadas de uma vez. Essa abordagem faz com que a linguagem Elm faça modificações na aplicação de forma muito eficiente, tornando a renderização da página bem mais rápida comparando com outras linguagens (FELDMAN, 2020).

Figura 13 – Comparativo entre linguagens do tempo em milissegundos na renderização de páginas.



https://elm-lang.org/news/blazing-fast-html-round-two

2.3 Requisições HTTP

O termo HTTP (Hyper Text Transfer Protocol) é utilizado para se referir a um protocolo de comunicação entre aplicações WEB. Esse protocolo define todas as regras de como devem ser feitas as requisições ao servidor para que a transferência de dado entre este e o cliente seja bem sucedida. O protocolo HTTP é utilizando em basicamente todas as aplicações, WEB ou mobile, da atualidade. Portanto, é lógico afirmar que para uma linguagem que foi criada para ser utilizada como ferramenta no desenvolvimento de aplicações WEB, deve possuir recursos que possibilitem o desenvolvedor utilizar este protocolo para realizar requisições dentro de seu código. É importante ressaltar que toda requisição HTTP lida com o mundo exterior à aplicação por isso, possui efeitos colaterais e, conforme apresentado na seção 2.2.4, deve ser feita através de comandos.

Para cumprir com esta tarefa, a linguagem Elm possui o módulo “elm/http”. O mesmo possui várias funções que podem ser utilizadas para enviar e receber dados pela aplicação e, portanto, algumas dessas funções serão apresentadas nas próximas seções, mas antes é necessário apresentar um novo tipo de dado customizado da linguagem Elm, o “Result”.

2.3.1 Tipo de dado “Result”

O tipo de dado “Result” é bastante similar ao “Maybe” apresentado na seção 2.1.9, a grande diferença é que ao invés de retornar “Nothing” para quando uma função falha, ele retorna o erro pelo qual a função falhou. Sua definição está apresentada logo abaixo.

Figura 14 – Definição do tipo de dado “Result”

```
type Result error value
  = Ok value
  | Err error
```

https://guide.elm-lang.org/error_handling/result.html

Basicamente, uma constante do tipo “Result” recebe dois parâmetros e pode ter dois valores: “Ok” ou “Err”. Caso o valor seja “Ok”, ela retorna o valor “value”, mas se for “Err” ela retorna o erro “error”. Este tipo de dado customizado é extremamente útil para requisições HTTP, uma vez que elas estão sempre sujeitas a falhas.

2.3.2 Http.get

A função Http.get, como o próprio nome já diz, executa uma requisição do tipo get para URL especificada. Para que ela possa realizar esta ação, deve ser passado como parâmetro um registro com os campos “url” e “expect. O primeiro simplesmente devem conter uma String com o endereço para o qual esta requisição deve ser feita e o segundo, merece uma pouco mais de atenção. Nas requisições feitas pela linguagem Elm, é obrigatório que o desenvolvedor sinalize para a linguagem qual o tipo de dado que ele espera receber daquela requisição. Portanto, o módulo elm/http fornece quatro funções para essa situação: ”expectString“, ”expectJson“, ”expectBytes“ e ”expectWhatever“. Os nomes de cada uma são intuitivos, a primeira é utilizada para quando se espera receber da requisição uma String, a segunda quando for um objeto JSON, a terceira quando forem Bytes e última pode ser utilizada caso não se saiba exatamente qual o tipo de dado que será recebido pela requisição. Todas elas devem receber um parâmetro do tipo Msg que será utilizado para identificar, dentro da função upload, qual o tipo ação que deverá ser tomada após obter a

resposta da requisição. Esta Msg, por sua vez, deve receber um valor do tipo ”Result“, que será utilizado para armazenar a resposta da requisição.

Figura 15 – Aplicação em Elm realizando uma requisição HTTP do tipo get.

```
type Msg
    = AlterarTexto
    | Resposta (Result Http.Error String)


type alias Model =
    { texto : String }


update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
    case msg of
        AlterarTexto ->
            (model, requisicaoGet)

        Resposta resposta ->
            ({ model | texto = resposta }, Cmd.none)



requisicaoGet : Cmd Msg
requisicaoGet =
    Http.get
    { url    = "http://localhost:8000"
    , expect = Http.expectString Resposta
    }




view : Model -> Html Msg
view model =
    div []
        [ h1 [ Html.Attributes.style "text-align" "center" ] [Html.text model.texto ]
        , button [] [ onClick AlterarTexto ]
        ]
```

A imagem mostra o mesmo programa apresentado na seção 2.2.3, porém, realizando uma requisição HTTP do tipo get para solicitar qual será o novo texto que deverá ser exibido na tela quando o usuário clicar no botão. Basicamente, quando isto é feito, a função “view” envia a mensagem “AlterarTexto” para a função “update”, que executa o comando “requisicaoGet”. Este comando executa a requisição para a URL definida na função “Http.get” esperando que a resposta seja uma String. Ao receber a resposta, ele envia a mensagem “Resposta” para a função “update” que atualiza o valor do campo “texto” do Model e envia o novo Model para a função “view”, que faz a renderização na página. Caso a requisição dê errado, o erro será retornado através da função “Http.Error” e este erro poderá ser acessado por meio do valor “Err” do tipo de dado “Result”.

2.3.3 Http.post e Http.request

A função “Http.post” é bastante similar a “Http.get” porém, com duas pequenas diferenças: o método de requisição que ela realiza é do tipo “post”. No método post do protocolo HTTP, deve ser enviado junto com a requisição o corpo da mensagem e, portanto, o registro que é passado como parâmetro para essa função deve possuir mais um campo, o campo “body”. Caso não se queira enviar nenhum dado, o módulo “elm/http” disponibiliza a função “Http.emptyBody”, que simplesmente envia um corpo vazio.

A função “Http.request”, diferentemente das outras duas apresentadas até agora que já vem com vários parâmetros do protocolo HTTP preenchidos, possibilita que o desenvolvedor configure com mais liberdade sua requisição, podendo definir seu método, o cabeçalho e o tempo de espera máximo pela resposta da requisição, conhecido como *“time out”.*

2.4 Integração com JavaScript

Comparada com JavaScript, a linguagem Elm é extremamente recente e é normal que ela ainda não ofereça todos os recursos que JavaScript oferece. Por isso, foram criados três mecanismos para que seja possível integrar um código Elm com JavaScript, caso seja necessário usar uma API de algum navegador que ainda não possui um módulo equivalente em Elm, por exemplo. Estes três mecanismos serão apresentados de forma bastante resumida, o intuito é apenas mostrar que, caso o desenvolvedor esteja tendo dificuldade para completar uma determinada tarefa com Elm, ele ainda pode fazer o uso destas ferramentas para integrar seu código Elm com JavaScript e construir uma aplicação mista.

2.4.1 Flags

A flag é uma ferramenta que possibilita que valores do código JavaScript sejam passados para o código Elm. Para utilizá-las, basta adicioná-las como argumento na função de inicialização do Elm dentro do arquivo HTML principal. Feito isso, o valor passado como flag será recebido como parâmetro da função “init” do arquivo elm, apresentada na seção 2.2.4 deste trabalho.

Apesar de ser bastante útil durante o desenvolvimento, a utilização deste recurso poderia causar problemas caso fosse definido na função “init” que uma flag deve ser do tipo Int e, por algum motivo, for enviado um valor do tipo String. É um cenário totalmente possível, visto que JavaScript não é uma linguagem que possui os recursos e a arquitetura que a linguagem Elm possui para garantir que tais erros em tempo de execução não

ocorram. Para lidar com esse problema, é realizada uma checagem de tipos de dados entre o que está sendo enviado como flag e o que o código Elm espera receber, antes mesmo de iniciar a execução do programa.

2.4.2 Ports

As ports são utilizadas para criar portas de comunicação entre um código JavaScript e um código Elm. Com essa ferramenta, é possível criar um fluxo de dados entre os dois códigos, fornecendo uma grande flexibilidade no desenvolvimento da aplicação.

2.4.3 Elementos Customizados

A possibilidade de criação de elementos customizados está cada vez mais presente nos navegadores atuais. Por isso, a linguagem Elm desenvolveu uma forma de obter acesso a elementos customizados criados pelo código JavaScript da aplicação. Basta criá-los no JavaScript e, no código Elm, acessá-los através do nome do elemento criado. Esta ferramenta torna possível que o desenvolvedor utilize a estrutura e valores desses elementos para realizar operações na linguagem Elm.

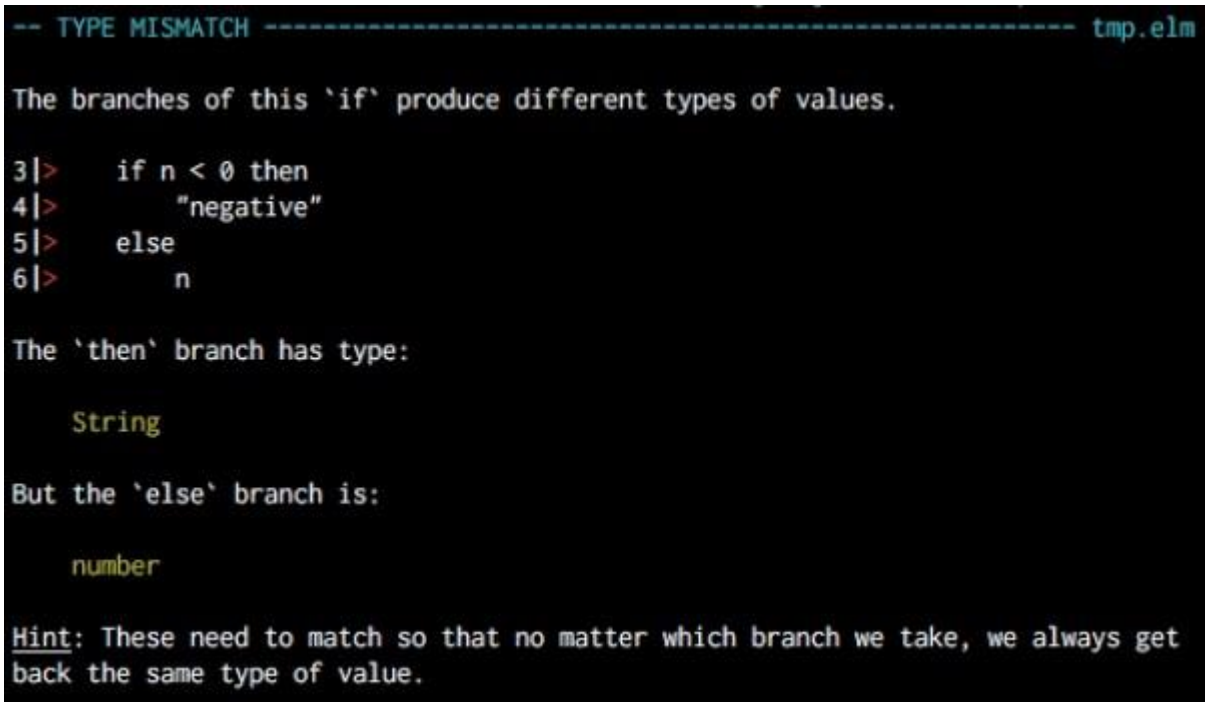
2.5 Compilador e Interpretador de Comandos

Todo programa que traduz, ou transforma, um código escrito em uma linguagem de alto nível para outra linguagem, podendo esta ser de alto nível ou não, é chamado de compilador. Na linguagem Elm, a função principal do compilador é converter o código que foi escrito em Elm para um código em JavaScript que será então utilizado pelo navegador do cliente para executar a aplicação desenvolvida. Esta ferramenta é a grande responsável por detectar erros que o desenvolvedor possa ter cometido no seu código, antes mesmo de criar o arquivo JavaScript que irá ser utilizado pelo navegador e, assim, impedir que ocorram erros em tempo de execução.

O compilador da linguagem Elm, além de executar sua função primária, se destaca de outras linguagens devido à forma com que ele exibe para o desenvolvedor os erros detectados no momento da compilação. Este foi um recurso que o criador da linguagem Elm, Evan Czaplicki, quis construir de forma que entender o que está de errado no código não fosse um trabalho árduo onde, muitas das vezes, é necessário que o desenvolvedor pesquise pelo código na “internet” para entender o seu significado. Para isso, ele construiu o compilador com mensagens extremamente claras e amigáveis que sinalizam o erro para o desenvolvedor e ainda sugere possíveis correções.

O interpretador de comandos, comumente chamado de REPL, fornece um ambiente de testes para que o desenvolvedor possa testar expressões da linguagem Elm antes de utilizá-las no código para checar o seu comportamento, caso ele tenha dúvidas. Pode ser usado para checar a tipagem de uma função, ou seja, o tipo de dado dos parâmetros que ela recebe e da expressão que ela retorna. Em resumo, é uma ferramenta que auxilia o desenvolvedor durante a construção de uma aplicação. A linguagem Elm possui o seu REPL que funciona em conjunto com o compilador que aponta os erros, em tempo real, do que o desenvolvedor está testando, ou checando, pelo interpretador. Ambas ferramentas tornam o desenvolvimento mais fluído e seguro, uma vez o desenvolvedor pode a qualquer momento que surgirem dúvidas, testar e checar partes de seu código pelo REPL, contando sempre com a ajuda do seu parceiro de desenvolvimento, o compilador.

Figura 16 – Mensagem de erro do compilador



https://elm-lang.org/news/compilers-as-assistants

A imagem acima mostra um exemplo em que o tipo de dado retornado por uma expressão "if" pode ser uma String ou um número, dependendo do resultado do teste lógico desse "if". Na linguagem Elm, isso é proibido e, portanto, o compilador identifica as linhas do código onde se encontra o problema, qual o problema, e ainda fornece uma explicação em forma de dica do motivo pelo qual isso é um erro.

2.6 Módulo elm/elm-ui

Como citado no início deste trabalho, para construir a interface gráfica de uma aplicação, o desenvolvedor conta três linguagens: JavaScript, HTML e CSS. O JavaScript é utilizado para a parte lógica da aplicação, é uma linguagem que lida com o fundo dela. Os conceitos da linguagem Elm apresentados até aqui também estão mais voltados para este lado. Comandos, assinaturas, modificações na tela provocadas por uma ação do usuário, requisições HTTP, são conceitos que cobrem exatamente este lado do desenvolvimento WEB. As linguagens HTML e CSS estão mais voltadas para a parte gráfica do Front-End e, assim como JavaScript, existem a muito mais tempo que a linguagem Elm. Elas possuem bibliotecas e frameworks que possibilitam o desenvolvimento com muito mais agilidade e, uma delas, é o Bootstrap. O Bootstrap é uma framework de CSS que disponibiliza várias estilizações de elementos HTML prontas e é extremamente utilizado nas aplicações dos dias atuais. Infelizmente, não é possível utilizar o Bootstrap através do Elm, mas isso não é um problema, por que a linguagem Elm possui o módulo elm-ui.

O módulo elm-ui fornece para o desenvolvedor maneiras de construir layouts de página de forma bem parecida com o Bootstrap, utilizando nomenclaturas semelhantes que tornam a curva de aprendizado mais amena. Ele fornece uma forma de criar elementos e estilizá-los utilizando a sintaxe da linguagem Elm e todos seus recursos como reutilização e reestruturação do código, caso o desenvolvedor necessite criar vários elementos idênticos em sua página, ele pode simplesmente criar uma função que retorna esse elemento e utilizar ela quantas vezes forem necessárias. O compilador obviamente também checa a utilização das funções deste módulo e, portanto, auxilia o desenvolvedor na também na criação de layouts. Além disso, é possível ainda utilizar trechos de CSS junto com os elementos para realizar estilizações mais simples.

Por se tratar de uma linguagem relativamente nova e um módulo ainda mais recente, existem determinas tarefas que o Bootstrap possibilita realizar que o módulo elm-ui ainda não consegue, porém o módulo está em constante desenvolvimento e se tornando mais completo a cada dia que passa. Também existe uma curva de aprendizado para utilizar este módulo, mas esta curva está mais ligada ao entendimento e conhecimento dos elementos que podem compor o layout de uma página do que ao módulo. Portanto, o esforço para aprender a utilizá-lo, é o mesmo comparado com Bootstrap. Assim como toda a linguagem Elm, tudo é muito bem documentado e a comunidade é extremamente proativa para ajudar novos desenvolvedores.

CAPÍTULO 3

Backend em Haskell

O Back-End de uma aplicação WEB é a parte responsável por lidar com o fluxo de dados da mesma. Toda a parte de armazenamento e consulta de dados é feita por ele. Para construção do back-end da aplicação descrita neste trabalho, foi escolhida a linguagem de programação Haskell.

3.1 Por que Haskell?

3.1.1 História da Linguagem

Na década de 1950, com o advento da computação, começaram a surgir as primeiras linguagens de programação de alto nível, uma delas foi o LISP que adotou um estilo de programação funcional . Com o passar do tempo, foram surgiram várias outras linguagens, principalmente durante a década de 1980, como Miranda, Hope e ML com utilização focada em pesquisa. Como naquela época, a computação ainda não era utilizada em larga escala, a quantidade de pessoas na comunidade que podiam contribuir no desenvolvimento de tais linguagens também não era muito grande, além de que, muita delas não eram *"open-source"*. Logo, o fato de haverem muitas linguagens de programação acabou não sendo tão produtivo, pois, dividia o pouco número de pessoas que existiam em várias comunidades diferentes, o que dificultou bastante o desenvolvimento de tais linguagens (OLEYNIKOV; DREIMANIS, 2019).

Enxergando este problema, um grupo de acadêmicos decidiram formar um comitê para criação e desenvolvimento de uma nova linguagem que seria utilizada como ferramenta de estudo, neste novo mundo que era a computação, e para ensinar às outras pessoas o paradigma de programação funcional. Depois de vários anos de trabalho e discussão de como esta nova linguagem cumpriria com seu objetivo de unir a comunidade, em 1990 foi lançada a primeira versão da linguagem Haskell.

A partir deste momento, a linguagem cresceu muito sua popularidade e se desenvolveu extremamente rápida, pois, contava com uma comunidade bastante grande época. Com o passar do tempo, algumas empresas começaram a demonstrar interesse em utilizar a linguagem na indústria. Como Haskell havia sido desenvolvido com o objetivo de ser utilizado para pesquisa e ensino, infelizmente ainda não era possível para estas empresas utilizarem a linguagem em seus negócios.

Graças a sua grande comunidade de contribuidores, a linguagem sofreu modificações que tornaram viável o seu uso na indústria. Atualmente, existem diversas empresas, muitas delas multi-nacionais, que utilizam Haskell em produção e, em 2020, foi criada a Fundação Haskell, que é financiada por muitas dessas companhias. Este contexto, criou um cenário de crescimento muito bom para a linguagem e, pode ser, que ela comece a ser cada vez mais utilizada no desenvolvimento de aplicações WEB. Portanto, é importante para a linguagem Elm, experimentar e constatar como é o desenvolvimento completo de uma aplicação utilizando Haskell, que possui um futuro muito promissor, na criação do back-end.

3.1.2 Semelhanças com Elm

Assim como a linguagem Elm, Haskell faz parte do paradigma de programação funcional. Este ponto em comum entre as duas linguagens, já trazem consigo algumas outras semelhanças. Ambas são consideradas linguagens de funções puras, ou seja, não causam efeitos colaterais na aplicação. As duas linguagens também trabalham fortemente em cima do conceito de imutabilidade.

Outra semelhança é que tanto Elm quanto Haskell, separam em seu código, as partes que interagem com o mundo exterior. A linguagem Elm, como apresentado anteriormente neste trabalho, utilizam os comandos para isso, e a linguagem Haskell utiliza o conceito de Mônadas. Haskell também possibilita que tipos de dados customizados sejam criados e, assim como Elm, também possui o tipo de dado *"Maybe"*, apresentado na seção 2.1.9 do capítulo 2.

Haskell também é uma linguagem extremamente tipada, onde todos os parâmetros e funções devem ter seu tipo de dado declarado juntamente em sua definição. Por último, a sintaxe das duas linguagens é extremamente semelhante, o que contribui muito na agilidade do desenvolvimento de uma aplicação.

A possibilidade de trabalhar com apenas um paradigma de linguagem durante a construção de uma aplicação WEB, e que ainda possui tantas semelhanças com Elm, foi algo que influenciou muito na escolha da linguagem Haskell para desenvolvimento do back-end do jogo desenvolvido neste trabalho. Haskell é a linguagem funcional que mais se assemelha à linguagem Elm.

3.1.3 Inteligência Artificial e Aprendizado de Máquinas

Na aplicação desenvolvida para este trabalho, foi criada uma inteligência artificial para realizar movimentos de um dos jogadores do jogo. Como as informações referente às posições das peças no tabuleiro, entre outras, seriam gerenciadas pelo back-end, fazia sentido criar esta inteligência artificial onde estes dados estavam sendo armazenados. Por isso, era importante que a linguagem escolhida para fazer este controle de dados, também fosse uma boa linguagem para a criação de inteligências artificiais e aprendizado de máquinas.

A linguagem Haskell, assim como outras linguagens funcionais, é considerada uma boa linguagem para se trabalhar com inteligência artificial. É uma linguagem que possibilita maneiras muito eficientes de escrever algoritmos de busca com construção de árvores e grafos, que são estruturas de dados fundamentais no desenvolvimento de inteligências artificiais. Falando em aprendizado de máquinas, códigos escritos em Haskell possuem uma alta capacidade de paralelismo, o que é um ponto muito positivo nesta questão. Por fim, a linguagem Haskell não produzirá as redes neurais mais rápidas dentre todas as linguagens, mas mesmo assim, ainda será bem melhor que Python neste quesito, atualmente a linguagem mais popular neste campo. Colocando em conjunto com os outros dois pontos apresentados anteriormente, foi concluído que Haskell seria uma boa escolha para compor o back-end do jogo.

Capítulo 4

Desenvolvimento da Aplicação

Para fazer a análise de como a linguagem Elm, com Haskell como back-end, se comporta no desenvolvimento de uma aplicação WEB, foi desenvolvido um jogo de tabuleiro chamado “Jogo do Peão”, que será apresentado neste capítulo.

4.1 Jogo do Peão

4.1.1 Descrição e Regras do Jogo

O Jogo do Peão consiste em um jogo de tabuleiro 8×8, o mesmo utilizado no xadrez, entre dois jogadores: o jogador preto e o jogador branco. No total, existem 5 peças, ou peões, que ocupam o tabuleiro, sendo quatro do jogador branco e apenas uma do jogador preto. Neste jogo, são utilizadas apenas as casas pretas do tabuleiro, portanto, no início do jogo, de um lado ficam os quatro peões branco e do outro lado, o único peão preto posicionado na casa do meio.

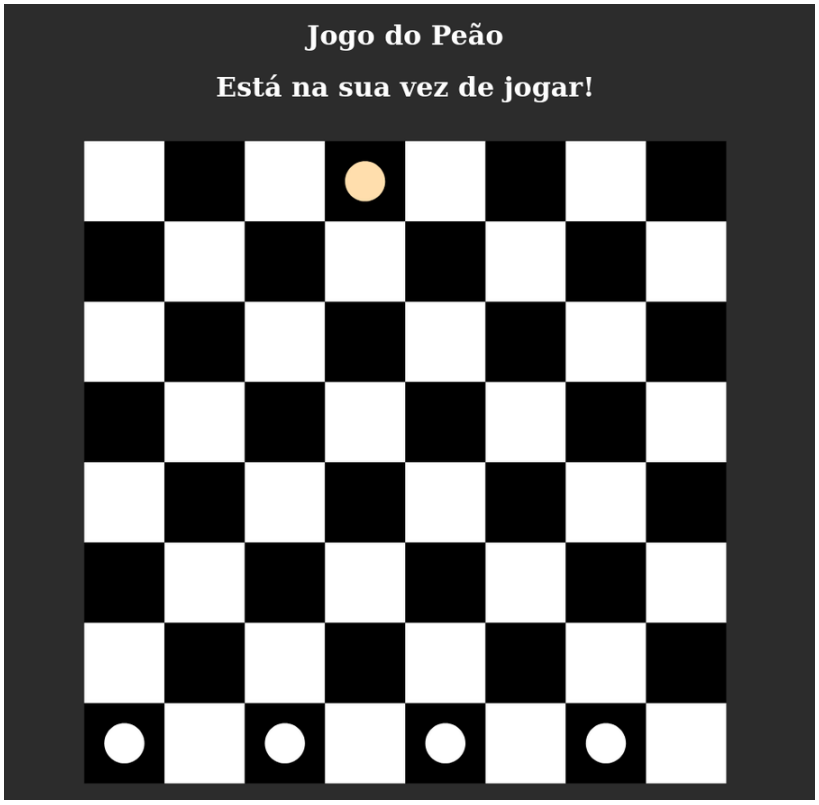
As condições de vitória do jogo são as seguintes:

- Jogador Preto: Vence se conseguir chegar na última casa no eixo y do tabuleiro.
- Jogador Branco: Vence se conseguir prender o peão preto, ou seja, se o peão preto, no momento de sua jogada, possuir zero possibilidades de movimento, o jogador branco vence.

Ambos jogadores podem mover seus peões apenas na diagonal, sendo que o jogador preto pode se movimentar para frente e/ou para trás, e o jogador branco apenas para frente. Vale ressaltar que não é possível mover um peão para uma casa que já esteja ocupada por outra peça, sendo ela da mesma cor que a sua ou não. Outro ponto importante é que a partida se inicia pelo jogador preto, ele é sempre o primeiro a jogar. A cor do peão do

jogador preto foi substituída pelo marrom, para facilitar a sua representação no tabuleiro. Caso contrário, ele ficaria camuflado com a cor da casa.

Figura 17 – Posições dos peões no início do jogo



Durante o jogo, terá sempre um texto logo acima do tabuleiro indicando se é a sua vez de jogar ou não. Se for, o texto "Está na sua vez de jogar!" será exibido até que você realize seu movimento. Se for a vez do oponente, será exibido o texto "O outro jogador está jogando. . . " até que o oponente realize seu movimento. Não existe tempo limite para que os jogadores realizarem a jogada.

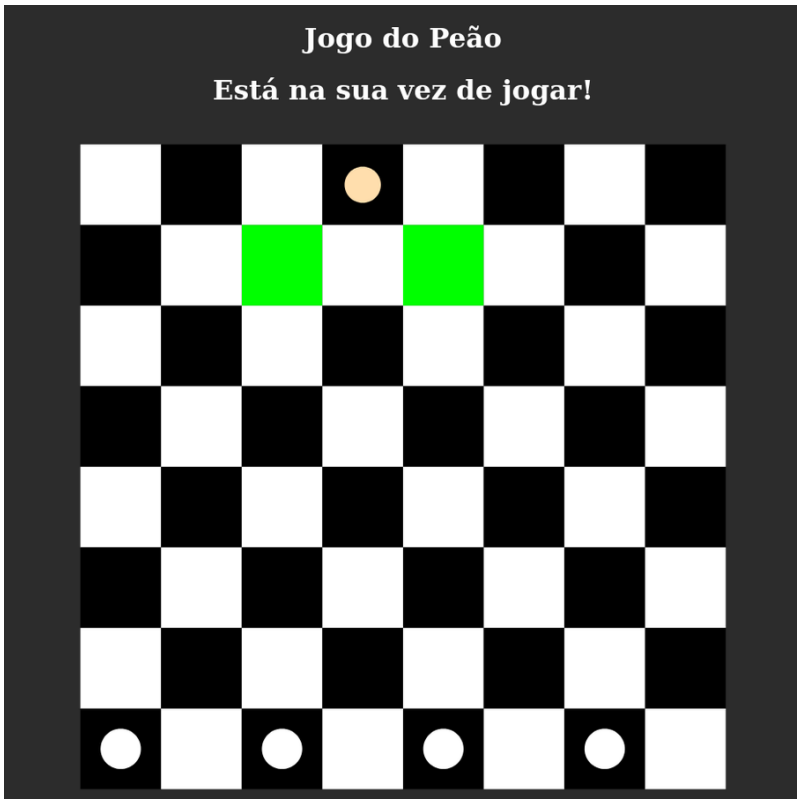
4.1.2 Como Jogar

A página inicial da aplicação é a página de escolha dos jogadores. Atualmente, foi desenvolvida uma inteligência artificial capaz de realizar movimentos apenas para o jogador branco. Portanto, caso queira jogar contra o computador, você deve escolher o jogador preto. Caso queira jogar contra um amigo ou amiga, basta escolher o jogador que quiser e esperar até que a outra pessoa escolha o que sobrou.

Para realizar os movimentos com os peões, o jogador deve clicar no peão desejado. Neste momento, o peão selecionado terá o seu tamanho um pouco reduzido, indicando

que ele foi selecionado, e será exibido na cor verde, as casas que o peão selecionado pode se movimentar, de acordo com as regras do jogo. Caso queira cancelar o movimento com aquele peão para analisar outras jogadas, basta clicar em outro peão do seu time, ou simplesmente clicar novamente no peão que havia sido selecionado para realizar o movimento. Assim que escolher o movimento desejado, é só clicar na casa destacada na cor verde e a jogada será realizada.

Figura 18 – Possíveis movimentos para o peão preto.



Quando um dos jogadores vencer, o tabuleiro irá desaparecer e em seu lugar aparecerá um texto informando qual jogador ganhou a partida, junto com um botão para iniciar um novo jogo.

4.2 Representação do Tabuleiro

A partir desta seção, será descrito como o jogo foi desenvolvido e o fluxo do código nos principais pontos da aplicação, começando pela representação do tabuleiro na tela.

4.2.1 Vetores Gráficos - Módulo elm/svg

A primeira tarefa no desenvolvimento do jogo, foi decidir a forma com que o tabuleiro seria representado na tela. Duas maneiras diferentes foram consideradas. A primeira foi através de uma imagem de um tabuleiro completo, onde depois seria definido em qual "pixel" da tela cada casa começaria e terminaria, para poder direcionar os movimentos dos peões corretamente. A segunda, foi através de vetores gráficos, onde cada casa do tabuleiro seria criada separadamente e colocada em conjunto com as demais para formar o tabuleiro completo. Após analisar as duas possibilidades, foi decidido seguir com a segunda opção. Entendendo como a linguagem Elm e sua arquitetura funciona, seria mais inteligente ter cada casa do tabuleiro criado separadamente para que fosse possível indicar mais facilmente onde os peões deveriam ser posicionados e também para onde eles seriam movimentados.

Para realizar esta tarefa, foi utilizado o módulo elm/svg, que fornece funções prontas para criação de vetores e elementos gráficos, como quadrados e círculos. Como o jogo é basicamente composto desses dois elementos, foi a opção mais inteligente.

4.3 Criação do Model

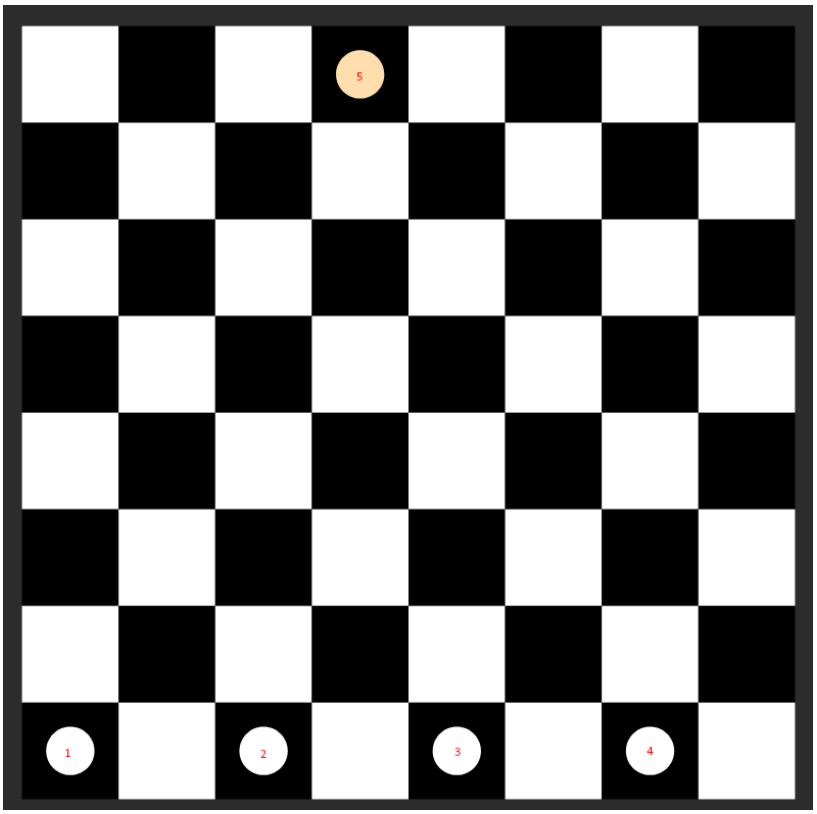
Depois de ter criado a representação do tabuleiro na tela, foi a hora de projetar como seria o Model da aplicação.

4.3.1 Elementos do Tabuleiro

Tendo em vista que cada peão possui uma posição diferente no tabuleiro e que cada um realiza ações de forma separada, foi decidido que o Model teria um campo armazenar a posição de cada um dos peões no tabuleiro. Foi necessário também definir como seriam representadas essas posições. Como o tabuleiro consiste basicamente em uma matriz de eixo x e y, a forma mais óbvia seria representar a posição dos peões em forma de tuplas, onde o primeiro elemento fosse um número do tipo Int representando a posição daquela peça no eixo x do tabuleiro, o segundo elemento desta tupla fosse também um número do tipo Int, mas que representasse a posição do peão do eixo y. Para definir esta tupla, foi utilizado o conceito de tipo de dados customizados para criar um tipo de dado, chamado de "PawnPosition". Com isso, toda vez que fosse necessário dizer ao código que um determinado dado ou determinada informação era uma tupla de Int, seria possível apenas dizer que o mesmo era do tipo "PawnPosition". Esse tipo de prática ajuda a manter o código mais coeso e fácil de realizar alterações no futuro.

Foi também necessário encontrar uma maneira de identificar quando um peão estivesse selecionado. Para isso, foi criado um outro tipo de dado customizado chamado "PawnState". Neste tipo de dado, foi criado os campos "status", "position" e "id", para armazenarem se existia um peão selecionado, ou em estado de movimento, a posição atual daquele peão no tabuleiro e o identificador dele, respectivamente. Os peões são identificados no programa pelo seu "id", que vai de 1 a 5, de acordo com a imagem abaixo.

Figura 19 – Representação dos peões em "ids".



Para finalizar, foram definidos mais quatro campos no Model da aplicação. O primeiro deles foi o campo "possibleMoves" que é utilizado para armazenar as casas que o peão selecionado pode se mover e, então, destacá-las com verde. O segundo foi o "playerTurn", que armazena qual jogador está jogando no momento para travar a movimentação do jogador que está esperando e exibir o texto acima do tabuleiro. O terceiro é o campo "gameOver", que indica qual o estado do jogo, se o branco ganhou esse campo terá o valor "1", se o preto ganhou ele terá o valor "2", e se o jogo ainda está em andamento o valor "0". Ele é utilizado para exibir a tela que informa quem venceu o jogo. E por último, o campo "player", que armazena qual é o jogador daquele. Abaixo, está a representação final do Model da aplicação.

```
type alias Model =
    { whitePawn1 : PawnPosition
    , whitePawn2 : PawnPosition
    , whitePawn3 : PawnPosition
    , whitePawn4 : PawnPosition
    , blackPawn  : PawnPosition
    , possibleMoves : List PawnPosition
    , pawnState : PawnState
    , gameOver : Int
    , playerTurn : String
    , player : String
    }
```

4.4 Modelo de Armazenamento dos Dados

A ideia do jogo, era fazer com que cada jogador pudesse jogar de casa com a suas respectivas máquinas. Para que isso fosse possível, era necessário que algumas informações ficassem disponíveis, através de um servidor, para que os clientes (dispositivos utilizados pelos jogadores) pudessem exibir na tela o estado atual do tabuleiro e também saberem se está na sua vez ou não de realizar um movimento.

As informações necessárias são: a posição de cada peão, qual jogador está na vez de jogar e o estado do jogo, informando se ainda está em andamento ou se já existe um vencedor. Como a quantidade de dados que precisam de armazenamento nesta aplicação é muito baixa, não era necessário a implementação de um banco de dados. Para armazenar essas informações, foram utilizados apenas um arquivo com uma String onde cada carácter dessa String representa uma das informações necessárias. Sendo assim, toda vez que um jogador realiza um movimento, ele envia uma requisição para o servidor, que será descrito na próxima seção, passando uma nova String com as informações atualizadas.

4.5 Haskell Servant - Criação do Servidor

Para tornar a aplicação, de fato, uma aplicação WEB, era necessário o uso de um servidor para receber requisições dos clientes. Para Haskell, não existe nenhuma ferramenta pronta, com um servidor todo configurado, como existe em outras linguagens. O Apache, por exemplo, não suporta a linguagem Haskell. Logo, foi necessário criar e configurar do zero um servidor exclusivo para esta aplicação.

Para esta tarefa, foi utilizado um framework do Haskell chamado Servant. Com ele, foi possível configurar as portas que seriam utilizadas para realizar requisições, os endereços e tipos de dados que seriam recebidos e retornados pelo servidor da aplicação.

O servidor foi configurado para receber requisições através da porta 8080 e foram criadas quatro tipos de requisições diferentes, que serão descritas agora.

4.5.1 Requisição “put”

Esta requisição foi criada para receber do cliente os novos valores, montados em uma String, e gravá-los no arquivo responsável por armazenar esses dados. Ao fazer uma requisição “put” para o servidor, o cliente obrigatoriamente deve enviar uma String na “query” com o nome de “newPos”, que será recebida pelo servidor e, então, inserida no arquivo.

Para exemplificar, suponhamos que o endereço IP do servidor seja 192.168.100.2, a requisição seria feita da seguinte forma: “http://192.168.100.2:8080/put?newPos=1131 517148B0’’. Desmembrando esta String, temos nos 10 primeiros caracteres, as posições dos peões no tabuleiro: “11” para o de id 1, “31” para o peão de id 2, “51” para o peão de id 3, “71” para o peão de id 4 e “48” para o peão de id 5. Os próximos dois caracteres, representam, respectivamente, qual jogador está jogando, sendo “B” indicando que é o jogador preto e “W” indicando que é o jogador branco, e o estado do atual do jogo, “0” significa que o mesmo ainda está em andamento, “1” significa que o branco ganhou e “2” indica que o preto ganhou.

4.5.2 Requisição “get”

A requisição “get” serve para enviar aos clientes as informações atuais no arquivo de dados. É utilizada principalmente para que o jogador que está em esperada, aguardando que o oponente realize a jogada, identifique quando o oponente realizar a jogada dele e qual foi o movimento de peão realizado. O cliente então recebe as novas posições dos peões e atualiza em seu Model.

Para utilizar esta requisição, basta executar uma requisição HTTP para o endereço “http://192.168.100.2:8080/get’’ que o servidor enviará a resposta contendo a String com os dados atuais do jogo.

4.5.3 Requisição “getChosenPlayer”

Esta requisição foi criada para ser utilizada na página de escolha dos jogadores da aplicação. Ela serve para informar ao cliente se um jogador já foi escolhido ou não e, caso tenha sido, qual foi este jogador. O cliente recebe essa informação e atualiza o Model de acordo com ela.

Para utilizá-la, basta executar uma requisição HTTP para o endereço “http://192 .168.100.2:8080/getChosenPlayer’’. O servidor irá receber esta requisição e enviar uma resposta contendo uma String identificando o estado atual de escolha dos jogadores.

4.5.4 Requisição “choosePlayer”

Por último, a requisição “choosePlayer” serve para que o cliente informe ao servidor que ele fez a escolha de qual jogador ele deseja utilizar naquela partida. Esta requisição deve ser realizada enviando, através da “query”, o parâmetro “pick”, com a String informando qual jogador ele escolheu. Esta informação será então recebida pelo servidor, que irá armazená-la dentro de um outro arquivo responsável por conter esses dados.

Ela deve ser realizada com o endereço “http://162.168.100.2:8080/choosePlayer? pick=white’’. O servidor recebe a informação pela requisição e escreve no arquivo que armazena a escolha dos jogadores, que o jogador branco foi escolhido. Este arquivo guarda essa informação em forma de String com dois caracteres. O primeiro carácter representa o jogador branco e o segundo o jogador preto. Logo, se no arquivo estiver “00”, nenhum dos jogadores foi escolhido, “10” significa que o jogador branco foi escolhido, “01” significa que o jogador preto foi escolhido e “11”, os dois foram já foram escolhidos. Os únicos valores que essa requisição pode receber pelo parâmetro “pick” são: “white”, “black” e “reset”. Este último é utilizado para resetar a escolha dos jogadores e iniciar um novo jogo.

4.6 Eventos da Aplicação

Como apresentado anteriormente, para possibilitar a interação do usuário com a aplicação, é necessário que sejam definidos os eventos que serão executados para isso. Nesta seção, serão apresentados todos os eventos que foram criados para criar a jogabilidade do “Jogo do Peão”.

4.6.1 Selecionar um Peão

Quando o jogador clica em um peão, reduz de tamanho e as casas no tabuleiro para onde aquela peça pode realizar um movimento são destacadas com verde. Para que isso fosse possível, foi necessário criar um evento, que é ativado quando o usuário clica na peça, para alterar o estado do Model atual da aplicação.

Conforme apresentado na seção 2.2.3, os eventos disparam uma mensagem para a função update, que executa as expressões definidas para aquela mensagem específica. Neste caso, a mensagem disparada para o evento de clique no peão foi chamada de

"PossibleMoves". Junto com a mensagem, são enviados quatro parâmetros, são eles: o jogador a quem aquela peça pertence, a posição atual do peão no tabuleiro, o "id" do peão o estado atual do tabuleiro. Estas informações são utilizadas para calcular os movimentos possíveis para aquele peão e, então, destacar esses movimentos em verde.

4.6.2 Cancelar um Movimento

O segundo evento implementado na aplicação foi o de cancelar o movimento de um peão. Ele também é ativado quando o usuário clica na peça, porém, esta peça precisa estar em estado de movimento. É dito que o peão está em estado de movimento quando ele já foi selecionado pelo jogador e está aguardando que o mesmo defina a casa para onde a peça deverá ser movida.

A mensagem definida para este evento foi nomeada de "CancelMove". Neste caso, não foi necessário enviar nenhum outro tipo de informação, pois, a única modificação necessária no Model era alterar o campo "pawnState", que identifica para a aplicação quando existe um peão em estado de movimento ou não. Não é necessário dizer para o Model qual o peão que está tendo seu movimento cancelado, apenas informar que nenhuma peça estará neste estado.

4.6.3 Realizar um Movimento

O próximo evento foi o de realizar o movimento com o peão. Este evento deveria ser ativado apenas quando o jogador, na sua vez de jogar, clicasse em uma casa do tabuleiro que estivesse dentro da lista de casas possíveis do peão selecionado mover-se. Sendo assim, este movimento está disponível apenas quando existe um peão em estado de movimento, e apenas nas casas destacadas com verde pelo evento de selecionar um peão.

A mensagem criada foi chamada de "MovePawn". Junto com a mensagem, são enviados dois parâmetros, são eles: a posição da casa escolhida para realizar o movimento e a lista de possíveis movimentos do peão selecionado. O programa então faz a checagem o movimento escolhido encontra-se dentro da lista e, caso esta condição seja verdadeira, realiza o movimento.

4.6.4 Envio do Movimento ao Servidor

Para que o outro jogador pudesse saber qual o movimento realizado pelo seu oponente, era necessário que toda vez que um movimento fosse realizada, a aplicação enviasse uma requisição HTTP "put" para o servidor para ele armazenar no arquivo os

novos dados. Como explicado anteriormente, as requisições HTTP envolvem o mundo exterior à aplicação e, portanto, devem ser executadas de forma diferente, através dos comandos.

Sendo assim, sempre que a função update finaliza a execução da mensagem "Move-Pawn", paralelamente ao envio do novo Model à função view, ela faz a chamada de uma função que realiza a requisição HTTP, seguindo o protocolo definido pelo servidor para aquela requisição. Com isso, o servidor recebe aquele chamado e realiza sua função de armazenar os novos dados para que o outro jogador, possa recebê-los através do evento que será descrito a seguir.

4.6.5 Requisição dos Dados do Jogo

Por último, foi definido o evento que possibilita que o jogador que está esperando seu adversário finalizar sua jogada, receba a informação de que isso ocorreu com a nova posição dos peões no tabuleiro. Para esta tarefa, foi utilizado o conceito de assinaturas apresentado na seção 2.2.6.

Foi definida uma assinatura para que, quando o jogador estivesse em modo de espera, a aplicação realize requisições HTTP "get" em intervalos de 200ms. Assim que o adversário realiza sua jogada, o jogador que está aguardando recebe essa informação, atualiza o campo "playerTurn" do Model e então, entra em modo de jogada.

4.7 Inteligência Artificial - Criação do Jogador Branco

Para dar mais contexto ao trabalho e na análise de utilização da linguagem Elm para o desenvolvimento de uma aplicação WEB, era necessário criar um back-end um pouco mais robusto para conseguir uma visão mais realista no estudo. Para isso, foi decidido criar uma inteligência artificial para realizar as jogadas dos peões brancos, utilizando a linguagem escolhida para compor o back-end, Haskell.

4.7.1 Algoritmo Minimax

O Minimax é um algoritmo muito utilizado em jogos para definir a melhor jogada que pode ser realizada por um jogador. A forma com que ele faz isso é analisando todas as jogadas possíveis do jogador, para uma determinada profundidade. Para cada jogada possível, o algoritmo antecipa as *n* possibilidades de jogada caso a primeira jogada seja realizada, onde *n* é a profundidade definida. Por exemplo, se a profundidade for 2, a inteligência artificial criada para o jogador branco irá analisar a primeira jogada possível

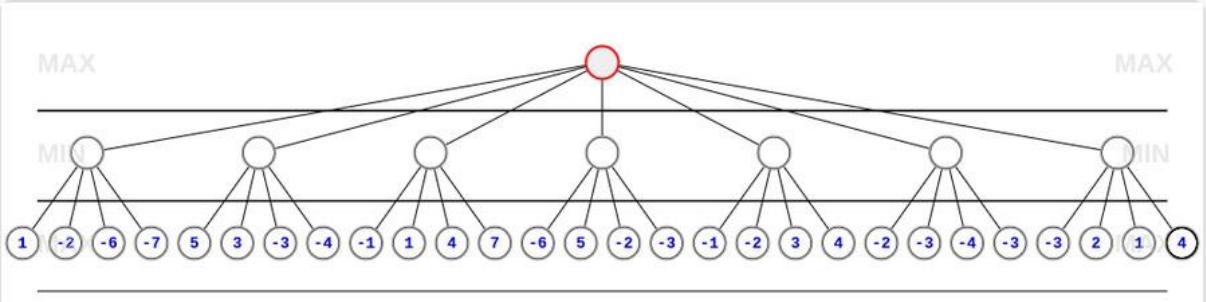
para o peão branco 1, depois ela analisa a primeira jogada possível para o peão preto e, por último, a primeira jogada possível para o peão 1 novamente, porém a partir da sua nova posição. Para cada jogada analisada, a profundidade é subtraída em 1 e, quando esta chega a zero, o minimax utiliza a função de avaliação para determinar a pontuação do tabuleiro naquele estado final, chamado de folha. Quanto maior a pontuação, melhor é a jogada para o jogador que está sendo avaliado.

O algoritmo então, constrói a árvore do jogo, onde cada nó representa um estado de tabuleiro. Os nós gerados a partir de um movimento do jogador branco são chamados de "min", e os nós gerados por um movimento do jogador preto, de "max". Para cada folha encontrada, o algoritmo faz a checagem se o valor daquela folha é maior ou menor que o valor de seu nó pai. Se o nó pai for um "min", ele checa se o valor da folha é menor, se for, o programa atribui aquele valor ao pai, e se não for, joga o valor fora. Se o nó pai for um "max", ocorre o contrário, se o valor da folha for maior que o valor do pai, o algoritmo atribui este novo valor ao pai, se não for, joga fora.

Com isso, no final da execução será identificado o caminho da árvore, ou qual jogada dentre todas as possíveis para o jogador branco, é o melhor a ser seguido. De forma simplificada, o minimax identifica a melhor jogada, utilizando uma função de pontuação definida pelo desenvolvedor, maximizando os ganhos do jogador e minimizando os ganhos do adversário.

Figura 20 – Estado do tabuleiro após o primeiro movimento do jogador preto.

Figura 21 – Árvore do jogo gerada a partir do estado de tabuleiro da Figura 20.



4.7.2 Função de Avaliação das Folhas

A função de avaliação utilizada para pontuar os estados de tabuleiros, ou nós, referentes a cada jogada avaliada nas iterações do algoritmo é composta por três partes principais.

A primeira delas, considera quantos movimentos possíveis o adversário, ou seja, o peão preto, possui naquele estado de tabuleiro. Quanto maior o número de movimentos possíveis para ele, pior é para o jogador branco e melhor para o próprio preto. Sendo assim, a função foi criada de forma que o resultado desta primeira parte seguisse esta lógica.

A segunda parte considera a distância dos peões brancos entre si no eixo y. Como o objetivo do jogador branco é não permitir que o peão preto chegue no final, é interessante que os peões brancos andem sempre juntos, tentando fechar as linhas por onde vão passando e nunca deixar buracos que o adversário possa se aproveitar. Sendo assim, esta parte da função foi criada de forma que quanto maior for a variação nas posições dos peões brancos no eixo y, menor será o valor por ela retornada. Para isto, foi feito a soma dos deltas entre as posições no eixo y do peão que está sendo movido, chamado de "yRef" e os demais peões.

No Jogo do Peão, é interessante para o jogador branco, uma vez que todas as suas quatro peças encontram-se na mesma linha (eixo y) to tabuleiro, começar a avançar para a próxima linha a partir do peão que tiver encostado na borda do tabuleiro. A terceira parte da função foi feita pensando nisso. Para o peão que está em uma linha par, será retornado uma pontuação maior para o peão que estiver mais à direita do tabuleiro e para o peão que estiver em uma linha ímpar, o contrário.

Existem casos específicos, que apenas pela lógica das três partes apresentadas acima, não teriam a sua pontuação representada da melhor forma possível, são os pontos fora da curva. Um desses casos, seria o peão preto, em um determinado estado de tabuleiro, possuir zero movimentos possíveis, o que levaria o peão branco a ganhar a partida. De acordo com as partes principais da função, poderia acontecer esta situação ao mesmo tempo, em que,

por algum motivo desconhecido, os peões brancos estivessem muito distantes entre si em relação ao eixo y, o que provavelmente levaria a inteligência artificial a escolher uma jogada com o intuito de diminuir essa diferença, que poderia não ser a jogada que faria o jogador branco vencer. Para resolver estes problemas, foi criada uma bonificação, que checa sempre a quantidades possíveis de movimentos do jogador preto no estado de tabuleiro que está sendo avaliada. Caso seja zero, é somado o valor de +infinito no resultado final da equação, para forçar a inteligência artificial a realizar aquele movimento, já que sem dúvida é o melhor de todos. Outro ponto fora da curva, seria uma situação em que a posição no eixo y do peão preto fosse menor ou igual à posição do peão branco com o menor valor no eixo y entre todos os peões brancos. Isso significaria que é impossível para o jogador branco barrar o caminho do peão preto e impedí-lo de chegar no final, já que o jogador branco não pode realizar movimentos para trás. Para este tipo de caso, foi criada uma penalização, que sempre faz essa checagem e, caso o estado de tabuleiro que está sendo avaliado, esteja nesta situação, é somado o valor -infinito no resultado final, forçando a inteligência artificial a não realizar aquela jogada, pois, certamente irá perder.

Por fim, cada parte dessa possui uma certa importância na avaliação. A parte 2 e 3 são mais importantes que a parte 1, por exemplo. Sendo assim, foi acoplado a cada uma dessas partes na equação um multiplicador, chamado de peso, para que fosse possível representar esta "importância" matematicamente. Atualmente, os valores dos pesos de cada parte da equação estão fixos, o que faz com que a inteligência artificial as vezes faça jogadas que claramente não seriam as melhores para situação. Para resolver esta situação, é possível criar um algoritmo genético que começará gerando valores aleatórios para os fatores da equação e testando seus resultados, para em seguida realizar as mutações dos genomas e gerar uma nova geração para ser testadas. Com isso, depois de um certo número de gerações, é bem provável que o algoritmo forneceria a melhor combinação de pesos possíveis.

4.8 Hospedagem da Aplicação - Amazon Web Services

Para ser possível que usuários de todo mundo acessem uma aplicação WEB, é necessário configurar a máquina em que esta aplicação está sendo executada. A Amazon Web Services (AWS) oferece diversos serviços de nuvem, desde infraestrutura, como computação, armazenamento e banco de dados, até tecnologias mais atuais como *"machine learning"*, *"internet"* das coisas e análises de dados. Tudo isso é feito através da alocação de recursos sob demanda, o usuário paga apenas pelo que utilizar.

A aplicação desenvolvida para este trabalho encontra-se hospedada nesse serviço. Por se tratar de uma aplicação bastante simples em termos computacionais, a infraestrutura necessária foi um servidor de apenas 2 CPUs virtuais e 4 GB de memória RAM. O sistema

operacional do servidor é o Linux, com a distribuição Debian 10. Para armazenamento de arquivos e programas, foi alocado e vinculado ao servidor, um "storage", ou dispositivo de armazenamento, magnético de apenas 30 GB.

Para configuração do servidor, foi necessário instalar a Plataforma Haskell, que conta com uma gama de ferramentas utilizadas pelo Haskell para criar projetos e gerenciar pacotes. Também foi necessário a instalação da framework Servant e, após instalada, criação de um projeto utilizando o *"stack"*, que é uma ferramenta da Plataforma Haskell que auxilia na hora de criar projetos com suas dependências de forma mais simples e automatizada. Tendo o projeto criado, foi feita a transferência dos arquivos príncipais do servidor, que são: o arquivo define todas as requisições aceitas pelo servidor, o arquivo texto que armazena o estado atual do tabuleiro e o arquivo que armazena a informação de que os jogadores já foram escolhidos.

Foi necessário também a instalação do apache que é utilizado para rodar a aplicação. Feito isso, foi feita a transferência dos arquivos JavaScript gerados pelo Elm e colocados dentro da pasta do apache. Também é necessário a utilização de um arquivo HTML apenas para carregar o script em JavaScript gerado pelo compilador da linguagem Elm e para definir a "div" em que toda a aplicação será carregada. Ao todo, são 6 arquivos, sendo três JavaScript e três HTML. Com isso, a máquina na AWS está pronta para rodar a aplicação.

Capítulo 5

Resultados e Próximos Passos

O desenvolvimento deste trabalho trouxe muito aprendizado na área de desenvolvimento WEB, linguagens funcionais de programação, servidores, inteligência artificial, aprendizado de máquinas, e até mesmo um pouco de computação em nuvem. Trouxe a oportunidade de conhecer e desenvolver utilizando uma linguagem que possui um grande potencial de crescimento no mercado, a linguagem Elm. Sua maior barreira neste caminho, é a natural resistência que desenvolvedores possuem para aprender novas tecnologias. É comum surgirem dúvidas quanto aos benefícios que essa linguagem pode trazer e se realmente existe a necessidade de migração de tecnologias extremamente consolidadas para outras alternativas.

Este projeto também possibilitou a aproximação com uma linguagem criada a bastante tempo, que é a linguagem Haskell. O mundo da computação é extremamente dinâmico, novas tecnologias surgem todo o tempo. Foi impressionante descobrir como esta linguagem, que já passou por tantas mudanças de cenário neste campo, ainda é útil e, além disso, utilizada por grandes multinacionais, como Facebook.

No campo da inteligência artificial, este trabalho trouxe a oportunidade de implementar um algoritmo em uma aplicação real. Este campo está conquistando cada vez mais espaço no cenário e é muito importante ter o mínimo de conhecimento e contato com o conceito. As possibilidades que a utilização deste tipo de tecnologia proporciona são inúmeras. O conceito de aprendizado de máquinas e algoritmos genéticos é extremamente poderoso e passa a sensação de que qualquer coisa é possível.

No que diz respeito à aplicação desenvolvida, o Jogo do Peão, foi possível criar uma experiência bastante agradável para o usuário/jogador. A aplicação funciona bem, é intuitiva e não possui falhas. A inteligência artificial criada para realizar os movimentos do jogador branco está longe de ser perfeita. É necessário realizar alguns ajustes, principalmente na função de avaliação das folhas. Com mais estudos sobre algoritmos genéticos e aprendizado de máquinas, talvez seja possível desenvolver um algoritmo para encontrar a melhor combinação dos fatores existentes na fórmula da função. Este, com certeza é o próximo passo para tornar o jogo mais interessante.

Por último, analisando o código escrito, foi possível compreender um pouco melhor

como é programar de forma imperativa. O código pode ser melhorado em alguns aspectos, algumas boas práticas acabaram sendo jogadas para escanteio devido à falta de um conhecimento profundo no paradigma funcional. Partes do código poderiam também terem sido implementadas de uma forma bem mais elegante e eficiente abusando mais de algumas ferramentas fornecidas pelas linguagens Elm e Haskell, ferramentas estas que não são comuns nas tecnologias atualmente consolidadas, o que causa uma certa dificuldade de enxergar os momentos certos para utilizá-las. Com certeza também é algo a ser melhorado nos próximos capítulos dessa aventura.

Capítulo 6

Conclusão e Considerações Finais

A linguagem Elm foi projetada de forma que dificulte, ou até mesmo impossibilite, que o desenvolvedor programe de forma irresponsável. O conceito de variáveis imutáveis, funções puras, de certa forma tirar a liberdade do programador, mas, ao mesmo tempo, garante que as boas práticas de programação serão obedecidas. Afinal de contas, elas existem por um motivo. Durante o desenvolvimento da aplicação, foi possível sentir essa falta de liberdade na pele, isso pode ser bastante duro para pessoas que já possuem uma certa experiência com linguagens não imperativas. Por outro lado, foi possível enxergar como essa "prisão" resulta em um código mais sustentável e fácil de realizar modificações. Em um ambiente empresarial, onde a qualidade está acima da quantidade, e o longo prazo é mais importante que o curto prazo, isso com certeza será muito benéfico.

O compilador criado para a linguagem Elm provou ser uma ferramente excepcional. Os relatos de que ele funciona como um colega de desenvolvimento, são reais. Programar em Elm nunca será uma tarefa solitária, o compilador sempre estará com você. Ele detecta todos os erros que podem, ou com certeza iriam, causar erros em tempo de execução e isso só é possível devido à toda arquitetura trabalhada pela linguagem. É incrível o ambiente que o criador da linguagem, Evan Czaplicki, criou para garantir que não existam pontas soltas no código.

A linguagem herda uma característica muito bem-vinda das linguagens funcionais, que é a produtividade e agilidade no desenvolvimento. Com as linguagens funcionais, o desenvolvedor se preocupa menos em instruir o computador de como ele deve realizar uma determinada tarefa, e mais em dizer para a máquina qual é a tarefa que ela precisa executar. Isso é possível graças à abstração que essas linguagens fornecem de como as instruções em baixo nível para alto nível e, com o tempo, torna o desenvolvedor mais ágil.

Infelizmente, as pessoas não estão habituadas a programarem dessa forma, e isso cria uma barreira para o crescimento da linguagem. Para utilizar esta linguagem, o desenvolvedor deve se esforçar em mudar sua maneira de pensar durante o desenvolvimento para uma maneira imperativa, o que muitas vezes causa desconforto, frustração e resulta na desistência do indivíduo ainda durante a curva de aprendizado. Esta é a única desvantagem identificada após o desenvolvimento deste trabalho. A documentação da linguagem Elm

é impecável, muito fácil de ser encontrada, extremamente clara e objetiva. A linguagem possui tutoriais para iniciantes excelentes. A comunidade é extremamente ativa e seu criador, Evan Czaplicki, encontra-se atualmente trabalhando 100% de seu tempo no desenvolvimento de novos recursos para a linguagem, totalmente financiado pela empresa *"No Red Ink"*, provavelmente a empresa que mais utiliza a linguagem Elm no mundo, e não se arrependem. Convido todos a buscar pelas palestras do Engenheiro de Software desta empresa, Richard Feldman.

Referências

ELM PROGRAMMING. **Beginning Elm - Contants**. Disponível em: https://elmprogramming.com/constant.html.

ELM PROGRAMMING. **Beginning Elm - Expression**. Disponível em: https://elmprogramming.com/expression.html.

FELDMAN, R. **Elm in Action**. [S.l.]: Manning, 2020.

GEEKSFORGEEKS. **Persistent Data Structures**. 2017. Disponível em: https://www.geeksforgeeks.org/persistent-data-structures/. Acesso em: 12/10/2021.

OLEYNIKOV, D.; DREIMANIS, G. **Haskell. History of a Community-Powered Language**. 2019. Disponível em: https://serokell.io/blog/haskell-history.