

# Expressões regular em linguagem funcional: Módulo de busca usando expressões regulares implementado em Haskell

Bruno Gomes

13 de outubro de 2020

## 1 INTRODUÇÃO

Qualquer estudante ou profissional que esteja envolvido no ambiente da tecnologia da informação certamente já teve contato com alguma espécie de linguagem de programação em algum ponto de sua jornada. De acordo com uma pesquisa feita pelo StackOverflow, as 5 linguagens de programação mais usadas são: JavaScript, Python, Java, Linguagens de script (Bash, Powershell, Shell) e C# (Stack s.d.).

Embora esse conjunto possa parecer como um conjunto heterogêneo de tecnologias, divergindo fortemente em convenções e nichos de uso, todas essas linguagens fazem parte da família de linguagens conhecidas como imperativas. Inquestionavelmente, as linguagens imperativas são muito importantes, pois as mesmas compõem a maioria do código sendo produzido diariamente, porém não são a única família de linguagens existentes. Nesse trabalho será discutido o paradigma de programação funcional, uma alternativa ao paradigma imperativo que domina o mercado.

O objetivo desse trabalho é introduzir o paradigma funcional ao leitor, começando por suas origens, introduzindo conceitos importantes e distinções com o paradigma imperativo. A fim de mostrar exemplos de código, será implementado um motor de busca para expressões regulares em Haskell. O código do motor de busca será usado de exemplo para ilustrar o paradigma funcional na prática. Para isso, será introduzido de maneira superficial a teoria das automatas e expressões regulares.

Em conclusão, esse trabalho tem como objetivo introduzir o paradigma funcional, através da linguagem Haskell, comparando os paradigmas funcional e imperativo, e discutindo a maneira funcional de resolver certos problemas computacionais.

## 2 Referencial Teorico

### 2.1 Expressões Regulares

As expressões regulares foram escolhidas como o problema computacional de interesse para introduzir programação funcional, porém como elas não são o foco deste trabalho, sua abordagem será simplificada. Nessa seção será introduzido o que é uma expressão regular, para que elas servem e como implementá-las.

### 2.1.1 Introdução

As expressões regulares, ou regex do inglês *regular expression*, são uma ferramenta muito poderosa para processamento de textos. De maneira simplificada, uma expressão regular é uma linguagem usada para descrever padrões de caracteres (Friedl 2009). A partir da expressão regular, e um texto alvo, usa-se um motor de busca que varre o texto a procura de segmentos para o qual a expressão regular é aceita. Um exemplo de uso seria uma expressão regular para buscar por todas as menções de hora em um texto, assumindo que o horário tenha um padrão uniforme (ex. HH:MM). Pode-se criar uma expressão regular para esse formato, e usando uma ferramenta de busca, encontrar todos os strings que atendam o formato definido.

Como dito anteriormente, a regex define uma linguagem para definir padrões de texto. Normalmente, essas linguagens fazem uso de caracteres especiais para indicar operações. As operações básicas são: concatenação, alternância, repetição e agrupamento.

A regex mais simples é um único caractere não especial, por exemplo "0", uma regex que procura pelo caractere "0". A operação de concatenação é implícita em uma regex, qualquer dois caracteres justapostos, não especiais, estão concatenados. Sendo assim, a regex "01" procura pelo string "01". A alternância normalmente é indicada pelo caractere "|", que indica que um caractere ou outro é válido. Um exemplo de alternância é a regex "01|0", que procura pelos strings "01" ou "0". Existem vários operadores de repetição, um dos mais usados é o operador de Kleene, normalmente indicado por "\*", esse operador indica que o caractere, ou grupo, que o precede pode ocorrer 0 ou mais vezes. Por exemplo, a regex "01\*" é equivalente a "0", "01", "011" e "0111...", ou seja, qualquer string que tenha um "0" seguido por qualquer número de "1"s. Por último, o agrupamento é definido usando parênteses "(01)", normalmente utilizado para indicar a repetição de um grupo de caracteres (Friedl 2009).

Existem várias outras funcionalidades e operadores, porém esses são os básicos. Para uma referência mais exaustiva, consulte (Friedl 2009).

### 2.1.2 Teoria das regex

As expressões regulares têm origem na teoria das linguagens formais, um assunto muito importante que formalizou a sintaxe das linguagens de programação (**theory-computation**). Fora a álgebra por trás desse tópico, existem ainda os inúmeros diferentes dialetos para regexes, visto que existem várias implementações diferentes com funcionalidades distintas. Sendo assim, regexes são um tópico extenso e complexo, e uma abordagem completa foge do escopo deste trabalho. Contudo, será explicado como uma regex é modelada em um ambiente computacional, o mínimo necessário para serem implementadas.

Uma regex é equivalente a uma máquina de estados, ou automata (**theory-computation**). Uma máquina de estado é um bom modelo matemático para um computador limitado, que opera sobre símbolos de entrada. A máquina de estado processa cada símbolo de entrada e muda de estado de acordo com sua construção. (**theory-computation**). A computação da máquina de estado encerra quando não existem mais símbolos de entrada, caso ela esteja em um de seus estados de aceitação, a computação foi válida.

Formalmente, uma máquina de estados consiste de: estados, símbolos de entrada,

um estado de início, estados de aceitação e uma função de transição. Cada estado é denominado por um nome único, normalmente um número. Os símbolos de entrada são um conjunto de caracteres que a máquina de estados reconhece, o conjunto de todos os símbolos reconhecidos forma o alfabeto da automata. O estado de início é o estado inicial da máquina de estados, sempre que iniciada ela se encontra nesse estado. A máquina de estado possui um conjunto de estados de aceitação, caso a a computação encerre em um desses estados, a computação foi válida. A função de transição é responsável por entender estados, essa função recebe um símbolo de entrada, o estado atual e retorna um novo estado. (**theory**)

Uma regex é equivalente a uma automata, segundo (**dragon-book**), podemos construir uma automata para uma regex de maneira indutiva. Na literatura, é enumerado as diferentes automatas equivalentes as regex primitivas, junto de como combinar essas automatas. Sendo assim, para construir um motor de busca deve-se converter os primitivos da regex em uma automata primitiva e em seguida, combinar as automatas.

Em conclusão, as expressões regulares são usadas para buscar padrões de texto. As expressões regulares são definidas usando uma linguagem própria, onde alguns caracteres tem significado especial. É possível converter uma regex em uma automata e usando o modelo da automata, é possível realizar uma busca em texto por uma expressão regular.

## 2.2 Programação Funcional

Programação funcional é um paradigma computacional que, de certa forma, contrasta com o paradigma imperativo. Esse paradigma é um tópico extenso e rico, com uma longa história por trás. Resumir esse assunto amplo é um desafio pois várias foram as contribuições e descobertas nesse campo de estudo. Um ponto de início é a definição dada por Bird (Bird 2016),

"Programação funcional é: um método para construção de programas que enfatiza funções e suas aplicações ao invés de comandos e suas execuções; programação funcional faz uso de notações matemática simples que permite que problemas sejam descritos de maneira clara e concisa. [...]".

Esse paradigma difere do imperativo pois a programação imperativa foca em passos para resolver um problema. O paradigma funcional tira o foco dos passos individuais para solucionar o problema e enfatiza uma estrutura para resolver o problema.

Embora seja difícil definir exatamente o paradigma funcional, a sua origem é bem clara. De maneira simplificada, o paradigma funcional veio a partir de um modelo computacional conhecido como Calculo Lambda. Segundo (**lambda**), o calculo lambda é um modelo de computabilidade criado por Alonzo Church em 1930. Nesse modelo, a operação básica é a aplicação de funções (**lambda**). O calculo lambda teve um impacto muito importante na programação funcional, e alguns autores defendem que o calculo lambda é fundamental para a aprendizagem de programação funcional, porém nesse trabalho esse tópico não será abordado.

A grande linguagem lisp marca as origens da programação funcional pois foi a primeira linguagem baseada no calculo lambda (**graham**). Essa seria apenas a primeira de um grande número de linguagens que se baseariam nesse modelo.

Deseja-se ressaltar que a famosa maquina de Turing, um conceito muito famoso da teoria da computabilidade, é equivalente ao calculo lambda. Embora a maquina de Turing e o calculo lambda seja ideologicamente diferentes, foi comprovado que os dois são equivalentes, hipotese essa conhecida como a hipotese de Churchill-Turing (**computability**). Isso significa que os problemas que podem ser resolvidos por uma maquina de Turing, e como consequencia um computador moderno, podem ser resolvidos usando uma linguagem funcional.

Em seguida serão abordados aspectos mais técnicos da programação funcional.

### 2.2.1 Imutabilidade

Um conceito comumente encontrado na programação funcional é a imutabilidade. Uma linguagem imutavel trata as variaveis de um programa de maneira similar à matemática, tal que o valor de uma variavel so pode ser definido no momento de sua inicialização. Isso é equivalente a definir todas as variaveis como final ou constante, dependendo da linguagem imperativa.

A imutabilidade é desejavel pois permite raciocinar sobre o programa de maneira mais facil, pois sabemos com certeza que um valor não ira mudar apos ter sido inicializado. Isso serve como uma especie de invariante que permite racionalizar sobre o programa, já que o programa não contém nenhum estado que varia com o tempo. Isso evita diversos problemas comuns que ocorre quando compartilhamos objetos, desde falta de atenção pela parte do programador até condições de corrida impostas pelo algoritimo.

Essa restrição é interessante pois altera muito a maneira como algoritimos são escritos. A ausencia da mutabilidade implica que não existem variaveis acumuladoras nem contadores. Sem contadores, a alternativa para repetir um bloco de código por n vezes passa a ser a recursão. A recursão é extremamente utilizada na programação funcional pois ela permite que uma função realize uma computação repetitiva, sem mutar valores.

### 2.2.2 Funções como um cidadão de primeira classe

Na programação, os tipos primitivos de uma linguagem são os blocos a partir do qual é possível construir estruturas complexas. Os tipos primitivos podem ser armazenados em uma variável, passados para uma função, e normalmente existem operadores para esses tipos. Na a programação funcional, uma função é um tipo de dado primitivo, isso significa que é possível declarar e armazenar uma função em uma variavel, passar uma função para uma função e receber uma função como retorno de uma função (**whyfpm**).

Na programação funcional, usar uma função como um tipo de dado é uma pratica essencial para criar abstrações. Uma função que recebe uma função como argumento é chamada de função de ordem superior. Hughes (**whyfpm**) argumenta que as funções de ordem superior são essenciais pois elas permitem uma melhor reusabilidade de código. Essa pratica é tão comum e poderosa que diversas linguagens populares, tal como JavaScript e Python, possuem esse tipo de função no seu core, tais como as funções: *map*, *filter* e *reduce*.

Um exemplo dessa abstração é a função *reduce*, ou como é chamada em Haskell, *fold*. Essa função é normalmente utilizada para iterar sobre uma lista de valores

e produzir um novo valor. Porém, essa abstração em específico é extremamente poderosa, em (**graham**) o autor argumenta a favor de sua expressividade.

As três funções mencionadas são exemplos de funções de ordem superior reutilizáveis e expressivas. É interessante notar que uma função de ordem superior, muitas vezes, pode ser estendida para aceitar diferentes tipos. Em Haskell, existem vários tipos de dados que aceitam a função *fold*, não só listas, e em (**whyfpm**) é argumentado que cada tipo de dado definido deve também implementar funções de ordem superior a fim de operar sobre esse tipo.

Em conclusão, funções como um cidadão de primeira classe permite tratar funções como valores e realizar transformações com elas de maneira transparente. Esse conceito permite que funções de ordem superior existam na linguagem e foi argumentado a favor do poder de abstração dessa prática.

### 2.2.3 Indo além

Existem vários outros importantes conceitos sobre programação funcional, porém por motivos de brevidade eles não serão comentados nesse artigo mas sim, mencionados e direcionados para outras literaturas.

Um tópico polarizador em programação funcional é o assunto de *laziness* e *eager*, referindo a quando um valor será computado. Existem vantagens e desvantagens para ambos; *laziness* é interessante por melhor performance em alguns casos, porém dificulta raciocinar sobre o programa. É importante mencionar que em (**whyfpm**), o autor argumenta que *laziness* é fundamental para abstrair programas funcionais.

Outro ponto importante é sobre tipos de dados algébricos. Tipos de dados algébricos permitem a implementação de tipos de dados recursivos. Em Haskell, uma lista é um tipo de dado recursivo, com dois construtores. Tipos de dados algébricos possibilitam *pattern matching*, uma maneira sucinta de verificar a estrutura de um dado. Uma boa introdução pode ser encontrada em (Miran 2012).

Por último, é importante mencionar alguns assuntos que surgiram no Haskell, dentre eles *type classes* e *monads*. *Type classes* foi a solução implementada em Haskell para um problema muito comum em linguagens de programação: *override* operadores (**haskell-ivory**). *Monads* é praticamente uma buzz-word em programação funcional, especialmente na comunidade Haskell. Sobre *monads*, é interessante mencionar que eles foram a solução para um grande problema que Haskell teve: como ser uma linguagem pura porém com efeitos colaterais (**haskell-ivory**).

### 2.2.4 Conclusão

Programação funcional é um tópico extenso com uma história rica e de maneira nenhuma seria possível introduzir tudo nesse artigo. Foi visto como as origens das linguagens funcionais diferem das linguagens imperativas, sendo baseadas no cálculo lambda. Em seguida foi introduzido dois conceitos importantes: imutabilidade e função como um cidadão de primeira classe. Foi argumentado a favor da modularidade e abstração que esses conceitos introduzem. Por fim, foi comentado sobre diferentes conceitos importantes para as linguagens funcionais, porém que fogem do escopo desse trabalho.

```

type Symbol = Char

data Operator = Concat | Alternation
data Quantifier = Kleene
data Token = SToken Symbol
            | QtToken Quantifier
            | OpToken Operator
            | GroupBegin
            | GroupEnd

data SubExpression = SubExp [Token]
                   | QuantifiedSubExp [Token] Quantifier

data ParseTree = Node ParseTree Operator ParseTree
               | QuantifierLeaf ParseTree Quantifier
               | Leaf Symbol

```

Figura 1: Tipos de dados definidos para o modulo de parse.

### 3 METODOLOGIA

Nessa seção será abordada a arquitetura do software desenvolvido, isso permite uma visão holística que define sua estrutura. O software desenvolvido é uma biblioteca de busca usando expressões regulares. Essa biblioteca foi quebrada em três módulos privados e um público. Os módulos internos definem os tipos de dados e implementam as transformações, enquanto o módulo público serve como uma interface que conecta os módulos. Os quatro módulos são: módulo de parse, módulo de automata, módulo de conversão e módulo público.

#### 3.1 Módulo de Parse

O módulo de parse é o primeiro estágio da pipeline que irá permitir a construção de uma automata de busca. Esse módulo é responsável por converter a entrada do usuário (uma String) em uma estrutura intermediária que é consumida pelo módulo conversor.

A saída do parser é uma estrutura em árvore, similar à uma árvore de parse gerada por um compilador. Nessa árvore, suas folhas são as primitivas do alfabeto de entrada (letras como “a”, “b” ou “c”), e os nós se interligam através dos operadores de concatenação e alteração. Existe uma exceção para os nós pois eles podem representar uma quantificação também. Logo, um nó é uma operação que liga subárvores ou é uma quantificação que permite o uso do operador “\*”.

A escolha de uma árvore para essa estrutura intermediária é conveniente por dois motivos. Primeiro, subárvores apresentam uma tradução, quase que, direta com automatas primitivas. Segundo, o uso da árvore elimina ambiguidades referentes à ordem das operações, sem precisar fazer uso de parênteses para indicar a qual grupo de caracteres um operador pertence.

Para construir a árvore, o módulo define alguns tipos de dados, na figura 1 são dadas suas definições. O tipo *Symbol* é sinônimo de um tipo de caractere. O tipo *Operator* define uma enumeração, podendo ser ou uma concatenação ou uma alternância. O tipo *Quantifier* define uma enumeração, representando o operador de Kleene, também conhecido como estrela. O tipo *Token* define um grupo de construções, podendo ser um token simbólico, token de quantificação, token de operação, delimitador de início de grupo ou delimitador de fim de grupo. O tipo *SubExpression*

Nome	Assinatura	Descrição
genToken	Symbol -> Token	Transforma
genTokens	[Symbol] -> [Token]	Transforma
normalizeStream	[Token] -> [Token]	Adiciona op
evenGroupPredicate	[Token] -> Bool	Valida que
uniqueQuantifierPredicate	[Token] -> Bool	Valida que
yankSubExp	[Token] -> (SubExpression, [Token])	Extrai uma
takeWhileGroupUneven	[Token] -> [Token]	Remove tol
takeWhileList	([Token], Bool) -> [Token] -> [Token] -> [Token]	Remove ite
validateTokens	[Token] -> Bool	Valida toke
sortAndTreefy	[Token] -> [Either Operator ParseTree]	Classifica T
buildSupExp	SubExpression -> ParseTree	Transforma
transformEithers	[Either Operator ParseTree] -> ([Operator], [ParseTree])	Agrupa ope
mergeOps	[Operator] -> [ParseTree] -> [ParseTree]	Combina á
buildTree	String -> ParseTree	Transforma

Tabela 1: Tabela de funções para o modulo de parse. Cada função é apresentada com sua assinatura e uma breve descrição.

representa uma sub-expressão composta por uma lista de tokens ou uma subexpressão quantificada. Finalmente, o tipo *ParseTree* representa uma árvore, podendo ter uma folha contendo um símbolo; um nó contendo uma árvore e uma quantificação; um nó contendo uma árvore, operador e outra árvore.

A implementação desse módulo em Haskell foi feita usando um conjunto de funções que opera sobre os tipos definidos anteriormente. A tabela 1 indica as funções desse módulo, junto com seus tipos de entrada, saída e uma breve descrição sobre cada uma.

No módulo é definida a função *buildTree* que recebe uma String e retorna uma árvore, uma função que faz todo o processo de conversão. Essa função funciona como uma pipeline de transformações sobre a entrada, até gerar a saída final. Primeiramente *buildTree* recebe um valor do tipo string o transforma para uma lista de tokens, que passam por um processo de normalização. O processo de normalização adiciona operadores de concatenação entre simbolos, visto que concatenação é uma operação implícita na regex. Após normalizada, a regex é validada usando uma lista de predicados, caso a regex esteja ok, a computação prossegue, caso não o programa retorna um erro. Os tokens validados são convertidos para um par de listas: os simbolos são convertidos para uma lista de folhas e os operadores são adicionados à lista de operadores. Por fim, as folhas são combinadas usando os operadores da lista de operadores, o resultado do processo de união é uma árvore onde os nós são as folhas unidas pelos operadores.

O fluxo acima descreve o processo de processamento para uma expressão regular sem sub-grupos. No caso de um subgrupo, o mesmo processo é seguido, porém de maneira recurssiva. Ao encontrar um grupo o algoritimo de agrupamento de tokens chama a pipeline de conversão novamente. No momento em que não existem mais subgrupos, o algoritimo retorna com uma árvore e o processamento pode continuar.

```

data SigmaElem s = Symbol s | Epsilon deriving (Show, Eq)
type State = Int

type Sigma s = Set.Set (SigmaElem s)

type Delta s = (State -> SigmaElem s -> Set.Set State)

data Automata s = NFA { alphabet :: Set.Set (SigmaElem s),
                        states :: Set.Set State,
                        q0 :: State,
                        qas :: Set.Set State,
                        delta :: Delta s
                      }

```

Figura 2: Tipos de dados definidos para o modulo de automatas.

Nome	Assinatura	Descrição
buildSigma	[a] -> Set (SigmaElem a)	Constroi um alfabeto
unionAndApplyOverDiff	(Set a -> Set a) -> Set a -> Set a -> (Set a, Set a)	Retorna a união de dois conjuntos
deltarOverSet	Delta s -> SigmaElem s -> Set State -> Set State	Aplica delta sobre um conjunto de estados
epsilonClosure	Delta s -> Set State -> Set State	Retorna todos os estados alcançáveis a partir de um estado inicial
stateEpsilonClosure	Delta s -> State -> Set State	Retorna estados alcançáveis a partir de um estado inicial
move	Delta s -> State -> SigmaElem -> Set State	Dado um símbolo, retorna o conjunto de estados alcançáveis
eval	Delta s -> State -> Set State -> [s] -> Bool	Aplica a maquina de estados a uma string

Tabela 2: Tabela de funções para o modulo de automatas. Cada função é apresentada com sua assinatura e uma breve descrição.

## 3.2 Modulo de Automatas

O modulo de automatas implementa funções que permite simular o funcionamento de uma automata. Esse modulo define um tipo de dado que representa uma maquina de estados e expõe funções que operam sobre uma maquina de estado.

Primeiramente, foram definidos os tipos de dados necessarios para modelar a maquina de estado. A figura 2 contém o trecho de código que define os tipos de dados. Foi definido o tipo *SigmaElem* que representa um elemento do alfabeto da automata, esse tipo contém o construtor *epsilon* que indica o símbolo de entrada nulo. O tipo *Sigma* representa o alfabeto da automata, ou seja conjunto de *SigmaElem*. O tipo *State* é um alias para um numero inteiro. O tipo *Delta* é um alias para função de transição da maquina de estado, igual ao definido na literatura. O tipo *NFA* define uma maquina de estados. *NFA* é uma tupla de cinco elementos: um alfabeto de tipo *Sigma*, um conjunto de estados, o estado inicial, conjunto de estados de aceitação e a função de transição. Percebe-se que a modelagem de uma automata segue exatamente o modelo definido na literatura.

As funções definidas nesse módulo são apresentadas na tabela 2. A implementação da automata foi feita de acordo com a literatura (**dragon-book**), onde é apresentado os algoritimos para simular uma automata. O algoritimo apresentado define algumas funções auxiliares, tal como *epsilonClosure* e *move*. As outras funções do modulo foram introduzidas devido a conversão do algoritimo imperativo da literatura para um algoritimo funcional.

Para fazer uso do maquinario que simula uma automata é necessario declarar uma *NFA*. A simulação de uma computação é feita usando a função *eval* que recebe uma maquina de estados, uma lista de símbolos e retorna um valor booleano. Após



Nome	Assinatura	Descrição
singletonNFA	SigmaElem s -> Automata s	Gera uma a
singletonDelta	SigmaElem s -> State -> State -> Delta s	Gera a func
wrapState	State -> SigmaElem s -> Set State -> Delta s -> Delta s	Função aux
addState	State -> Delta s -> Delta s	Adiciona u
addStateWithTransitions	State -> [(SigmaElem s, Set State)] -> Deta s -> Delta s	Adiciona u
alternateDelta	State -> State -> NFA s -> NFA s -> NFA s	Combina d
concatNFA	NFA s -> NFA s -> NFA s	Combina d
kleenifyNFA	State -> State -> NFA s -> NFA s	Adiciona es
foldParseTree	ParseTree -> NFA s	Converte a

Tabela 3: Tabela de funções para o modulo de conversão. Cada função é apresentada com sua assinatura e uma breve descrição.

processar todos os símbolos, caso a maquina de estado esteja em um estado de aceitação o retorno da função é verdadeiro, caso contrario falso.

Usando a arquitetura apresentada, a implementação das funções definidas permite a simulação de uma maquina de estados. Essa maquina de estados sera utilizada para executar a busca pela expressão regular. Para isso, é necessário converter a regex em árvore em uma maquina de estados.

### 3.3 Modulo de conversão

O modulo de conversão é responsável por converter a expressão regular armazenada em uma árvore para uma automata que possa ser executada.

A literatura contém maquinas de estados para as primitivas de uma expressão regular. Usando essas primitivas, é possível combina-las para construir expressões mais complexas. O módulo de conversão disponibiliza funções para criar e combinar essas primitivas. A partir dessas funções, basta trafegar pela árvore para transformá-la em um unico valor, ou seja, usar um *fold*.

A tabela 3 apresenta as funções definidas para o módulo. Note que esse modulo não introduz um tipo de dado próprio, apenas realiza transformações.

### 3.4 Modulo público

O módulo público cria uma interface para que a biblioteca seja facil de utilizar. Nesse módulo é definida uma única função, *match*, que recebe duas strings: uma expressão regular e um texto sobre o qual a busca é feita. *Match* apenas faz chamadas para as funções definidas nos outros módulos e não introduz nenhuma logica nova. Esse módulo é apenas uma interface para o usuário.

### 3.5 Práticas comuns e observações

Embora os módulos da biblioteca tenham objetivos distintos, algumas práticas foram utilizadas em todos eles. Para o desenvolvimento das funções foi feito o uso de uma metodologia usando testes untiaris, em que para cada função é escrito um teste para validar seu comportamento. Para isso, foi utilizado a biblioteca HUnit do Haskell, uma biblioteca que auxilia na execução de casos de testes. A prática de desenvolver

testes para as funções dos módulos foi de extrema importância pois permite validar cada unidade lógica do código de maneira individual.

Nessa seção foram introduzidos os diferentes módulos que compõe a biblioteca proposta: *parse*, *automata*, *conversão* e *publico*; também foi visto as funções propostas para cada módulo. Essa descrição da arquitetura do software permite o seu entendimento sem entrar nas nuances associadas a implementação das funções. Além disso, permite que diferentes implementações existam, pois uma vez definida a interface do programa, basta que as partes previstas existam para que o mesmo funcione, sem existir a dependência de como a interface é implementada. O código completo para a biblioteca está disponível no apêndice 1 e no site <http://github.com/lodek/regex-engine>.

## 4 Analise e Discussão

A análise do código escrito será feita por módulo. Será abordado tópicos relevantes sobre cada módulo a fim de exemplificar conceitos da programação funcional. Por fim, será feita uma análise sobre o processo de desenvolvimento como um todo.

### 4.1 Módulo de Parse

O módulo de *parse* é responsável por transformar uma *regex* em formato de string em uma árvore. Um ponto crucial desse módulo é ler os caracteres do *stream* de entrada e identificar quando uma expressão completa foi lida. Por exemplo, armazenar os caracteres que compõem uma subexpressão de uma expressão regular.

Em uma linguagem imperativa, esse problema é trivial, basta adicionar os símbolos da subexpressão à uma estrutura de pilha e removê-los quando eles formarem uma unidade completa. Devido à imutabilidade das linguagens funcionais, isso não é possível.

Uma alternativa que respeita a imutabilidade faz uso de recursão. Quando a função de conversão encontra um caractere que indica o início de um grupo, essa função chama a si própria para que seja construída a árvore da subexpressão. Ela lê os caracteres de entrada até que seja identificado o fim de um grupo, onde é retornado a árvore para a subexpressão. Essa solução dispensa o uso de uma pilha e flags para guardar o estado do código, mantendo a imutabilidade.

Com as árvores construídas elas devem ser unidas usando operadores, onde para cada operador, as duas primeiras árvores da pilha de árvores são unidas e o resultado é adicionado ao topo da pilha. Nesse caso, a solução requer tanto mutação quanto repetição. Esse problema pode ser resolvido usando um *fold*, basta notar que o objetivo é consumir o valor de uma lista e gerar um único valor a partir disso. A função de união itera sobre os operadores e usa como acumulador a lista de árvores. Para cada operador, os dois primeiros elementos são unidos e o novo acumulador é uma nova lista onde a árvore resultante é o primeiro elemento concatenado ao resto da lista original. No final da execução o acumulador é uma lista com um único elemento, a árvore final.

A implementação do módulo de *parse* foi a parte mais desafiadora do ponto de vista técnico. Os algoritmos bem documentados para *parse* de *stream* de caracteres são imperativos e não possuem uma tradução direta para funções devido à mutação. Existem alternativas mais apropriadas para escrever um parser em Haskell que faz

uso de *Monads*. Essa metodologia não foi escolhida pois monads são um tópico avançado e não alinham com a proposta educativa do trabalho.

## 4.2 Modulo de Automatas

O modulo de automata implementa algoritmos que permitem a simulação de uma automata.

Seguindo a literatura (**dragon-book**), uma das funções usadas para simular uma automata deve receber um conjunto de estados e retornar um conjunto de todos os estados acessíveis usando uma transição nula, a função "epsilon closure". A solução desse problema imperativamente faz uso de uma estrutura de repetição, uma lista de resultados a ser visitados e uma lista de estados visitados. Para cada estado a ser visitado, adicione-o a lista de estado visitados e teste quais estados são acessíveis a partir dele, para cada estado acessível, cheque se ele está na lista de estados visitados, caso não, adicione-os à lista de resultados a ser visitados; esse procedimento é repetido até a lista de estados para visitar ficar vazia.

Uma alternativa funcional desse problema é interessante pois demonstra uma solução inusitada, que faz uso de operações de conjunto. Para essa solução é necessário dois conjuntos, um de estados visitados e outro de estados a ser visitado. Assim, verifica-se todos os estados acessíveis a partir dos estados de entrada, esse gera um conjunto resultado temporario. O conjunto de estados a ser visitado passa a ser a diferença do conjunto de estados visitado e o conjunto temporario. O conjunto de estado visitados passa a ser a união do conjunto original com o conjunto temporario. A função que gera o conjunto de estados acessível é recursiva e se repete até que os estados a serem visitados forma o conjunto vazio. A alternativa funcional para a função *epsilon closure* é interessante pois mostra que, por mais que mutação de valores seja uma prática intuitiva para o desenvolvedor experiente, muitas vezes existem alternativas que podem ser mais breves e mais elegantes.

A partir da função acima, o resto da implementação da automata é trivial pois as funções da literatura podem ser facilmente traduzidas para o paradigma funcional.

## 4.3 Modulo de conversão

O módulo de conversão é responsável por transformar uma árvore de parse em uma automata.

O "cerébro" da automata é dada pela função de transferencia, que retorna as possíveis transições para um estado da automata. Como foi visto, a construção da automata é feita de maneira incremental, onde automatas mais complexas são construídas a partir de um conjunto finito de estruturas bases. Para combinar as funções de transição foram definidas funções que recebem duas funções de transição e retornam uma nova função de transição. Esse uso é um exemplo de uma função de ordem superior.

Como funções são cidadãos de primeira classe em Haskell, pode-se passar funções para funções de maneira transparente. Isso faz com que combinar funções seja feito de maneira trivial, usando uma função lambda como o retorno das funções de combinação.

Para construir a automata final, basta converter os nós e as folhas da árvore em automatas, fazendo o uso das funções fábricas para gerar as funções de transição

intermediárias. A conversão da árvore para uma automata é mais um caso de uso para um *fold*, visto que deseja-se iterar sobre a árvore de parse, um tipo de dado multi-valor, e gerar um único valor a partir dela.

Visto que grande parte do trabalho do modulo de conversão envolve a manipulação de funções, é notável que os desafios impostos pela implementação desse modulo fazem bom uso das ferramentas da programação funcional. O problema de combinar funções é feito de maneira clara e concisa devido a natureza desse paradigma.

#### 4.4 Observações gerais

O desenvolvimento do código proposto usando o paradigma funcional foi desafiador. A experiência de um programador imperativo, em muitos casos, gera atrito com o paradigma funcional. As restrições impostas pela falta de iteração e mutação forçam diferentes maneiras de resolver um problema. Em fala coloquial, o paradigma gera uma sensação de não saber qual palavra usar em uma frase.

Haskell, sendo uma linguagem puramente funcional, retira as ferramentas normalmente conhecidas de um programador imperativo. Muitas vezes ao escrever um programa imperativo, a tarefa é clara e embora o caminho não esteja totalmente claro, a liberdade dada pela mutabilidade de variáveis e pelo sequenciamento de instruções para realizar a computação permite que o problema seja resolvido de maneira interativa. Essa prática de resolver o problema durante a implementação não funciona bem em Haskell pois quanto maior a função mais complexo fica desenvolve-la. Devido a isso, o programador é incentivado a usar funções pequenas para resolver o problema. O resultado é um processo diferente para o desenvolvimento de código, um processo onde os passos macro para resolver um problema devem estar muito mais claros. O problema a ser resolvido deve estar totalmente compreendido antes do desenvolvimento começar, pois visto que funções grandes são difíceis de implementar, todas as funções auxiliares e a sequência de transformações necessárias devem estar completamente entendidas pelo programador. O mesmo não é verdade no paradigma funcional. Nada impede o programador de escrever uma única função para realizar inúmeras tarefas, embora seja uma prática desencorajada devido ao problema de compreensibilidade do código que isso gera. É perfeitamente natural adicionar várias etapas de processamento e transformações em uma rotina na programação imperativa; não existe nenhum atrito no processo. Essa diferença no processo de desenvolvimento não livra o código de bugs, eles ainda ocorrem, mas o paradigma funcional força o entendimento do problema de maneira macro antes da implementação no micro, o que aumenta as chances de que o código escrito esteja correto na primeira vez.

Um aspecto que não ficou claro durante o desenvolvimento do projeto é como lidar com um problema estruturalmente complexo. O problema de implementar um motor de busca regex tem poucos tipos de dados, somente a árvore de parse e a automata. Seria interessante ver como uma linguagem funcional se comportaria em um problema com vários tipos de dados e estruturas diferentes.

## 5 CONSIDERAÇÕES FINAIS

O objetivo desse trabalho foi expor alguns conceitos por trás do paradigma funcional e exemplificar esses conceitos através da construção de um módulo de processamento

de expressões regulares. O módulo foi implementado usando a linguagem Haskell, uma linguagem funcional.

Primeiramente, foi explicado o que são regexes do ponto de vista de um usuário e quais problemas elas resolvem. Em seguida, foi introduzido os conceitos por trás das expressões regulares, tais como sintaxe e operações. Juntamente, foi introduzido as máquinas de estado, ou automatas, sua definição e como simular uma automata em um ambiente computacional. Finalmente, foi visto a equivalência entre uma expressão regular e uma automata.

Sobre o paradigma funcional, comentou-se sobre imutabilidade, *laziness*, funções como cidadãos de primeira classe e funções de ordem superior. Durante a exposição inicial do paradigma funcional, foi contrastado como as origens desse paradigma difere do paradigma imperativo.

Para a implementação da biblioteca proposta, foi analisada a arquitetura projetada para a ferramenta. A arquitetura é composta por 4 módulos, que realiza a transformação de uma regex em formato de string para sua automata equivalente. Para cada módulo, foi explicado o seu fluxo lógico e as funções que o compoe.

Por fim, foram selecionados alguns dos problemas encontrados na implementação da biblioteca e analisamos como resolver esses problemas fazendo uso dos conceitos introduzidos. Essa abordagem permitiu expor os pontos as diferenças entre os paradigmas, mostrando como resolver problemas de uma maneira funcional.

Em conclusão, esse trabalho tem o objetivo de mostrar um mundo diferente da programação, um mundo que vem sido incorporado às linguagens imperativas, mesmo que muitos programadores desconheçam suas origens.

## Referências

- Bird, Richard (2016). *Thinking functionally with Haskell*. Cambridge University Press.
- Friedl, Jeffrey E. F. (2009). *Mastering regular expressions*. O'Reilly.
- Miran, Lipovača (2012). *Learn you a Haskell for great good!: a beginners guide*. No Starch Press.
- Stack, Overflow (s.d.). *Stack Overflow Developer Survey 2019*. URL: <https://insights.stackoverflow.com/survey/2019>.