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

Bruno Gomes

12 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 essa lista de linguagens possa parecer como um conjunto heterogêneo de tecnologias, divergindo fortemente em convençoes e nichos de uso, todas essas linguagens fazem parte da família de linguagens conhecidas como imperativas. Inquestionavelmente, as liguagens 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 sua origem, serão introduzidos 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.

Por fim, será discutido de maneira breve o impacto que as liguagens funcionais tem no mundo da programação. Embora o seu uso seja muito menor, várias foram as descobertas feitas por pesquisadores estudando esse paradigma. Curiosamente, essas discobertas muitas vezes se propagam para outras linguagens, inclusive as imperativas.

Em conclusão, esse trabalho tem como objetivo introduzir o paradigma funcional, através da linguagem Haskell, comparando os paradigmas funcinal 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 as expressões regulares, porém como elas não são o foco destre trabalho, sua abordagem será simplificada. Nessa secão será introduzido o que é uma expressão regular, para que elas servem e como implementa-las.

2.1.1 Introducão

As expressões regulares, ou regex do inglês *regular expression*, são uma ferramenta muito poderoza na computação, utilizadas para processar texto. De maneira simplificada, uma expressão regular é uma linguagem usada para descrever padrões de caracteres. 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 horario 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 operções básica são: concatenação, alternação, repetição e agrupamento.

A regex mais simple é simples é um único caractere não especial, por exemplo "0", uma regex que procura pelo caractere 0. A operação de concatenção é implicita em uma regex, qualquer dois caracteres não especiais estão concatenados. Sendo assim, a regex "01"procura pelo string "01". A alternação normalemente é indicada pelo caractere ", que indica que um caractere ou outro é valido. Um exemplo de alternação é a regex "01|0", que procura pelos strings "01"ou "00". 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 veses. Por exemplo, a regex "01*"é equivalente a "0", "01", "011"e "0111...", ou seja, qualquer string que tenha um "0"seguido por qualquer numero de "1"s. Por ultimo, o agrupamento é definido usando parenteses "(01)", normalmente utilizado para indicar a repetição de um grupo de caracteres.

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 tem origem na teoria das linguagens formais, um assunto muito importante que formalizou a síntaxe das linguagens de programação. Fora a algebra por traz desse tópico, existem ainda os inúmeras diferentes dialetos para regexes, visto que a existem varias implementações diferentes com funcionalidades distintas. Sendo assim, regexes são um tópico extenso e complexo, que foje do escopo deste trabalho. Para tanto, 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 mátematico para um computador limitador (**theory-computation**). A máquina de estado opera sobre símbolos de entrada, a cada símbolo enviado à ela, ela muda de estado. A computação da máquina de estado encerra quando não existem mais símbolos de entrada, caso ela esteja em um estado de aceitação, a computação foi bem sucessida.

Formalmente, uma maquina de estados consiste de: estados, símbolos de entrada, um estado de inicio, estados de aceitação e uma função de transição. Os estados são denomidados por um nome único, normalmente um número. Os símbolos de entrada são o conjunto de caracteres que a maquina de estados reconhece. O estado de inicio é o estado inicial da máquina de estados, sempre que iniciada ela se encontra nesse estado. A maquina de estado pode ter um conjunto de estados que são considerados válidos quando não existem mais símbolos de entrada. A funcão de transicão é responsável por inteligar 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 automatas. Sendo assim, para construir um motor de busca deve-se converter os primitivos da regex em uma automata primitiva e em seguinda, 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 paradgima computacional que, de certa forma, contrasta com o paradigma imperativo. Esse paradigma é um topico extenso e rico, com uma grande historía por traz. Resumir esse assunto amplo é um desafio pois várias foram as contribuições e descobertas nessa campo de estudo. Um ponto de inicio é a definição dada por Bird (Bird 2016),

> "Programção funcional é: um método para construção de programas que enfatiza funções e suas aplicações ao invéz de commandos 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 nos passos individuais para solucionar o problema e enfatiza uma estrutura para resolver o problema.

Embora seja dificil 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 basica 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 numero 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, essa hipotese é 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 similar à matemática, tal que o valor de uma variavel so pode ser definido no momento de sua inicializacão. Isso é equivalente a definir todas as variaveis como final ou constante, dependendo da linguagem imperativa.

A imutabilidade é desejavel pois permite racicionar sobre o programa de maneira mais facil, uma funcão nunca tera um efeito colateral (**history**). 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. 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 existe 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 funcão 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 que opera sobre esses tipos. Para 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 retorna 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 a funções de ordem superior são essencial pois elas permitem uma melhor reusabilidade de código. Essa pratica é tão comum e poderoza que diversas linguagens populares,

tal como JavaScript e Python, possuem esses tipos de função no seu core, tais como as funções: map, filter e reduce.

E de fato, um exemplo de uma abstração muito poderoza 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 especifico é extremamente poderoza, em (**graham**) o autor argumenta a favor de sua expressividade.

As três funções mencionadas são exemplos de funções de ordem superior reutilizaveis e expressivas. É interessante notar que uma função de ordem superior, muitas vezes, pode ser extendida para aceitar diferentes tipos. Em Haskell, existe varios tipos de dados que aceitam a função fold, não so listas, e em (**whyfpm**) é argumentado que cada tipo de dados definida deve tambem implementar funções de ordem superior para operar sobre essa estrutura.

Em conclusão, funções como um cidadão de primeira classe permite tratar funções como valores e realizar tranformações sobre 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 breviedade 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 algebricos. Tipos de dados algebricos permitem a implementação de tipos de dados recurssivos. Em Haskell, uma lista é um tipo de dado recurssivo, com dois construtores. Tipos de dados algebricos possibilitam *pattern matching*, uma maneira sucinta de verificar a estrutura de um dado. Uma boa introdução pode ser encontrada em (Miran 2012) e (**rust**).

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 historia rica e de maneira nenhuma seria introduzir tudo sobre nesse artigo. Foi visto como as origens das linguagens funcionais diferem das linguagens imperativas, sendo baseadas no calculo 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, foram comentado sobre

diferentes conceitos importantes para as linguagens funcionais, porém que fogem do escopo desse trabalho.

3 METODOLOGIA

Nessa seção será abordada a arquitetura do software desenvolvida, isso permite uma visão holistica que define uma estrutura. O software desenvolvido é uma biblioteca para busca usando expressão regular. Essa biblioteca foi quebrada em três modulos publicos e um privado. Os modulo internos definem os tipos de dados e implementam as transformações, enquanto o modulo público serfve como uma interface que conecta os modulos e uma interface de entrada / saida. Os quatro modulos são: modulo de parse, modulo de automata, modulo de conversão e modulo publico.

3.1 Modulo de Parse

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

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

Para construir a árvore, o modulo define alguns tipos de dados. Na fígura 1 é dada a definição dos tipos de dados definidos. O típo *Symbol* é sinomimo de um tipo de caractere. O tipo *Operator* define uma enumeração, podendo ser ou uma concatenação ou uma alternação. 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 simbolico, token de quantificação, token de operação, delimitador de inicio de grupo ou delimitador de fim de grupo. O tipo *SubEpression* 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 arvore 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. Note que a função buildTree recebe uma String e retorna uma árvore, sendo assim essa função realiza a transformação completa sobre a entrada.

```
type Symbol = Char

data Operator = Concat | Alternation deriving (Show, Eq)
data Quantifier = Kleene deriving (Show, Eq)
data Token = SToken Symbol | QtToken Quantifier | OpToken Operator | GroupBegin | GroupEnd deriving (

data SubExpression = SubExp [Token] | QuantifiedSubExp [Token] Quantifier deriving (Show, Eq)

data ParseTree = Node ParseTree Operator ParseTree | QuantifierLeaf ParseTree Quantifier | Leaf Symbo
```

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

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 tok
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.

```
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 alfab
unionAndApplyOverDiff	(Set a -> Set a) -> Set a -> Set a -> (Set a, Set a)	Retorna a união d
deltarOverSet	Delta s -> SigmaElem s -> Set State -> Set State	Aplica delta sobre
epsilonClosure	Delta s -> Set State -> Set State	Retorna todos est
stateEpsilonClosure	Delta s -> State -> Set State	Retorna estados a
move	Delta s -> State -> SigmaElem -> Set State	Dado um símbolo
eval	Delta s -> State -> Set State -> [s] -> Bool	Aplica a maquina

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

A arquitetura do módulo de parse é dada pelas funções na tabela 1 e os tipos de dados enumerados no texto. A partir desses elementos, é possível implementar as funções tal que o modulo receba uma expressão regular em uma string e retorne uma árvore que preserve o significado semantico da expressão regular de entrada.

3.2 Modulo de Automata

O modulo de Automata implementa o modelo computacional das maquinas de estados. 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. O tipo *Sigma* 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. Finalmente, analogo à literatura, uma maquina de estados 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

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.

literatura para um algoritimo funcional.

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 em uma automata que possa ser executada.

A literatura contém maquinas de estados para as primitivas de ume expressão regular. Usando essas primitivas, é possível combina-las para construir expressões mais complexas. O módulo de conversão faz disponibiliza funções para criar e combinar essas primitivas. A partir dessas funções, basta trafegar pela árvore para transforma-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 epxressão regular e um texto sobre o qual a busca é feita. *Match* apenas faz chamadas para as funções definidasa 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 para implementar todos eles. Para o desenvolvimento das funções foi feito o uso de uma metodologia usando testes untiarios, 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 as funções propostas para cada módulo. Essa descrição da arquitetura do software permite o seu entendimento sem entrar nas nuancias 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 dependencia de como a interface é implementada. O código completo para a biblioteca está disponivel no apendice 1 e no site http://github.com/lodek/regex-engine.

4 Analise e Discussão

A analise do codigo escrito sera feita por modulo. Sera abordado topicos relevantes sobre cada modulo a fim de exemplificar conceitos da programacao funcional. Por fim, sera feita uma analise sobre o processo de desenvolvimento como um todo.

4.1 Modulo de Parse

O modulo de parse é reponsável 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, como armazenar os caracteres que compõem uma subexpressão em uma expressão regular.

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

Uma solução que respeita a imutabilidade é usando recurssão. Quando a função de converssão encontra um caractere que indica o inicio de um grupo, essa função chama a si propria para que seja construída a árvore da subexpressão. Essa função 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, respeitando assim a imutabilidade do código.

Com as árvores construidas elas dever 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. Nessa caso, a solução requer tanto mutação quanto repetição. Esse problema pode ser resolvido usando um fold, basta notar deseja-se 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, as duas primeiras árvores são consumidas e o novo acumulador é uma nova lista onde a árvore resultante é o primeiro elemento, concatenado aos elemntos restantes da lista original. No final da execução o acumulador é uma lista com um unico elemento, a árvore final.

A implementacao do modulo de parse foi a parte mais desafiadora do ponto de vista tecnico. Os algoritimos bem documentados para parse de um stream de caracteres sao imperativos e nao possuem uma traducao direta para funcoes devido a mutacao. Existem alternativas mais apropriadas para escrever um parser em Haskell que faz uso de Monads. Essa metodologia nao foi escolhida pois monads são um tópico avançado e não alinhariam com a proposta educativa do trabalho.

4.2 Modulo de Automatas

O modulo de automata implementa algoritimos que permite a simulação de uma automata.

Seguindo a literatura, uma das funções usadas para simular uma automata deve receber um conjunto de estados e retornar um conjunto de todos os estados acessiveis 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 á 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 acessiveis 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 *epislon 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 autmoata é feita de maneira incremental, onde automatas mais complexas são construidas a partir de um conjunto finito de estruturas bases. Para combinar as funções de transição foram definidas funções que recebem duas funções de transiçao e retornam uma nova função de transição. Esse uso é um exemplo de uma função de ordem superior.

Como funções são cidadoes 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 intermediarias. 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 involve a manipulação de funçõese, é notavel 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 codigo proposto usando o paradigma funcional foi desafiador. A experiencia 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 nao 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 variaveis e pelo sequenciamento de instrucoes para realizar a computacao permite que o problema seja resolvido de maneira interativa. Essa pratica 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 eh incentivado a usar funcoes pequenas para resolver o problema. O resultado é um processo diferente para o desenvolvimento de codigo, 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 comecar, pois visto que funções grandes são dificeis de implementas, todas as funcoes auxiliares e a sequencia de transformacoes sobre o valor de entrada deve estar completamente entendido pelo programador. O mesmo não é verdade no paradigma funcional. Nada impede o programador de escrever uma única função para realizar inumeras tarefa, embora seja uma pratica desencoragada devido ao problema que isso gera na compreensibilidade do codigo no futuro. É perfeitamente natural adicionar varias etapas de processamento e transformações em um procedimento na programação procedural; não existe nenhum atrito no processo. Essa diferença no processo de desenvolvimento nao livra o código de bugs, eles ainda ocorrem, mas o paradigma funcional forca o entendimento do problema de maneira macro antes da implementacao no micro, o que aumenta as chances de que o código escrito esteja correto na primeira vez.

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

Por fim, sobra o assunto de testes de software. Desenvolvedores experientes entendem muito bem a necessidade de escrever testes para códigos escritos. Existem vários tipos de testes mas os mais básicos são os testes unitarios. Foi observado que a construção de testes unitarios em uma linguagem funcional pura é um processo mais simples do que em linguagem orientadas a objetos, pois visto que objetos normalmente armazenam algum estado e possui referencia para outros objetos, a construção de testes unitarios involve tecnicas de mock. Os mocks são usados para imitar classes, muitas vezes imitando as dependencias internas de uma classe, isso

permite que uma classe seja testada de maneira isolada. O uso de mock resolve o problema porém é um processo tedioso e verboso, sendo necessário especificar o comportamento de vários mocks para cenarios diferentes. Como em uma linguagem puramente funcional, nenhuma função depende de estado, validar o comportamento de uma função é trivial pois toda função possui uma entrada e uma saída clara, dispensando o uso de mocks. Assim, os testes unitarios realmente passam a ser testes caixa-preta pois a implementação interna do código é irrelevante, não sendo necessario verificar que algum estado interno da classe tenha sido alterado.

5 CONSIDERAÇÕES FINAIS

O objetivo desse trabalho foi expor alguns conceitos por traz do paradigma funcional e exemplificar esses conceitos através da construção de um módulo de processamento de expressoes regulares. O modulo 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 traz das expressões regulares, tais como síntaxe e operações. Juntamente, foi introduzido as maquinas de estado, ou automatas, sua definição e como simular uma automata em um ambiente computacional. Finalmente, foi visto a equivalencia entre uma expressão regular e uma automata.

Sobre o paradigma funcional, comentou-se sobre imutabilidade, *laziness*, funções como cidadões de primeira classe, funções de ordem superior e composição de funções. Durante a exposição inicial do paradigma funcional, foi contrastado como as origins 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 modulos, 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 funciona.

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

Referências

Bird, Richard (2016). *Thinking functionally with Haskell*. Cambridge University Press.

Friedl, Jeffrey E. F. (2009). *Mastering regular expressions*. OReilly.

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`.