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

Bruno Gomes

9 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 EXPRESSÕES REGULARES E PROGRAMAÇÃO FUNCIONAL

Como foi abordado na introdução, esse trabalho une dois temas: expressões regulares e programação funcional. Esses temas serão, primeiramente, discutidos separadamente, e em seguida será feito uma previa de como será feita a construção do motor de processamento de expressões regulares.

2.1 Introdução as expressões regulares

Expressões regulares, tambem conhecidas como *regex* (da junção do nome em inglês, *regular expression*) são utilizadas para realizar buscas complexas sobre strings. Para Cox, "expressões regulares são uma notação que descreve um conjunto de strings. Quando alguma string está no conjunto associado à expressão regular, pode-se dizer que essa expressão regular corresponde a esse string."(Cox 2007).

As regexes são utilizadas frequentemente, tanto para extrair informações que seguem um padrão ou para realizar buscas mais flexíveis ou parciais. Como exemplo, suponha o problema de extrair todas os strings que correspondem a um horario em um texto. A escrita de um horario segue uma estrutura padrão, HH:MM:SS onde HH delimita as horas, MM delimita os minutos e SS delimita os segundos. Sem ter que construir todas as possíveis combinações de horas que seguem esse formato, uma simples varredura de texto é incapaz de extrair essa informação. Esse problema pode ser resolvido tranquilamente usando regexes.

Uma expressão regular que realiza esta busca é,

$$[0-9]\{2\}:[0-9]\{2\}:[0-9]\{2\}. \tag{1}$$

Em palavras, os símbolos [0-9] representa qualquer carácter numerico entre 0 e 9. De maneira geral os símbolos [] representam um conunto de caracteres (Python s.d.). O token {2} indica uma repetição, sendo equivalente à regex [0-9][0-9], ou seja dois carácteres numericos. Os simbolos {} são usados para representar repetição (Python s.d.). O caractere ":"é interpretado de maneira literal. Fazendo a união, a regex acima equivale a qualquer string que tenha o formato DD:DD:DD onde D indica qualquer digito de 0-9. Podemos ver que esse formato é exatamente o formato definido anteriormente.

É importante ressaltar que existem inumeras variações e implementações de regexes, onde existem diferentes meta-caracteres para descrever operações. Em (Friedl 2009), o autor discute as diferenças em regex entre as linguagens: PHP, .NET, Java e Perl. Na documentação oficial da linguagem Python (Python s.d.) é dito que o dialeto usado é basedo nas expressões regulares da linguagem Perl com alguns adicionais. Usuários UNIX também estão familiarizados com os *wildcards* presentes nos shells, uma forma de regex. Em resumo, existem diversos dialetos porém o objetivo das regexes não se altera, buscar por padrões. A regex acima e todas as regexes subsequentes nesse texto serão escritos no dialeto da linguagem Perl.

2.2 Programação Funcional

Essa seção aborda programação funcional e suas caracteristicas de maneira resumida. Há muito a se falar sobre esse assunto pois ele é extensso e tem uma longa história.

Para abordar a programação funcional será tomado um foco que toma como base a computação.

De maneira geral, programas de computadores existem para resolver problemas computacionais. Segundo (Klein 2015) um problema computacional é "[...] uma especificação de entrada-saída que um procedimento tenha que satisfazer."e um procedimento é "[...] uma descrição precisa de uma computação; ele aceita entradas (chamadas de argumentos) e produz uma saída (chamado de valor de retorno).". Ou seja, independente do paradigma utilizado para resolver o problema (funcional ou imperativa), ambos são capazes de definir um procedimento para um problema computacional, a grande diferença está em como esse procedimento é definido.

Segundo (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. [...]".

A programação imperativa foca em passos para resolver um problema, cada passo desse pode ser traduzido de maneira rasoavelmente direta em instruções de uma CPU. Isso faz com que o procedimeto escrito imperativemente reflita muito mais a máquina do que ao homem. O paradigma funcional tira o foco nos passos individuais para solucionar o problema e enfatiza uma estrutura para resolver o problema.

Em seguida serão abordados aspectos mais técnicos da programação funcional.

2.2.1 Funções para resolver problemas

Como visto anteriormente, a programação funcional propõe que problemas computacionais sejam resolvidos de maneira mais declarativa. O foco muda de "quais passos é preciso para resolver esse problema"para "quais transformações aplicar nas minhas entradas para produzir a saída".

Para exemplificar essa idea considere o seguinte problema. Dado uma lista de nomes, com nome, nome do meio e sobrenomes, crie uma lista com todas as combinações de primeiro nome e ultímo nome, ignorando nomes do meio e aonde as combinações cuja soma do primeiro e ultimo nome não exceda 15 caracteres (incluindo o espaço). Para isso, ao invez de analisar os passos para processar esses dados, uma boa ideia é pensar em como manipular os dados para se obter o resultado desejado. Para esse problema, sugere-se a seguinte solução:

1. Separar cada nome da lista de nomes nos espaços e armazenar os nomes uma lista.
2. Filtrar listas que só possuem um elemento (somente um nome).
3. Filtrar listas e remover nomes do meio.
4. Criar uma lista de nomes e uma lista sobrenomes.
5. Realizar o produto cartesiano sobre essa lista e gerar uma lista de tuplas.
6. Tranformar tuplas em strings fazendo a concatenação do primeiro nome e do sobrenome.

Percebe-se que cada passo acima realiza uma única ação, sendo ela simples e clara e é interessante modelar cada um desses passos como uma função. Na linguagem Haskell o tipos dos argumentos e do retorno de uma função é dado pela notação nomeDaFuncao :: arg1 -> arg2 -> ... -> retorno, onde arg1 e arg2 definem os tipos dos argumentos (Miran 2012). Para identificar listas em Haskell é usado o símbolo [], ou seja [Char] indica uma lista de caracteres e tuplas são indicatas com () onde (String, String) indica uma tupla com dois elementos, ambos strings. Podemos agora reescrever o problema acima definindo todas as funções que serão utilizadas.

Primeiramente definiremos o problema enunciado como uma função usando a notação introduzida. O problema inicial é a função $combinarNomes :: [String] -> [String]$, ou seja uma função que recebe uma lista de Strings e retorna uma lista de Strings. Em seguida, iremos declarar as funções que representam cada passo acima.

1. separarNomes :: [String] -> [[String]]
2. tirarIncompleto :: [[String]] -> [[String]]
3. removerSobrenomes :: [[String]] -> [[String]]
4. gerarNomesESobrenomes :: [[String]] -> ([String], [String])
5. gerarCombinacoes :: ([String], [String]) -> [(String, String)]
6. concatenarNomes :: [(String, String)] -> [String]

Segundo (Miran 2012) a assinatura de uma função em Haskell combinado com um nome descritivo diz muito sobre a função e de fato, dados os nomes e sua assinatura, pode-se facilmente deduzir o que cada função está fazendo.

O objetivo dessa seção foi dar um exemplo alto nível de como é resolvido um problema de maneira funcional.

2.3 Automatas e expressões regulares

Como visto, as expressões regulares representam uma maneira conveniente de descrever conjutos de string. Embora conveniente, a maneira na qual as expressões regulares foram introduzidas não permite uma tradução direta delas para um ambiente computacional. Essa seção faz a ligação entre esses objetos teóricos e uma descrição matématica das mesmas.

2.3.1 Definição de uma automata

Segundo (Sipser 2013), as automatas modelam um computador com uma quantidade minúscula de memória. A ideia central de uma automata é representar uma estrutura computacional a partir de um conjunto de estados e entradas.

Os estados da automata constitui um conjunto denominado de Q, o conjunto de estados. Dentre esses estados existe um único estado inicial da automata chamado de $q_o \in Q$. Automatas recebem entradas a partir de simbolos, o conjuto de todos os símbolos reconhecidos por uma automata define um conjunto Σ chamado de alfabeto da automata. Os estados da automata podem ou não estar conectados, quando existe uma conexão entre dois estados essa conexão é representada por um simbolo $\alpha \in \Sigma$. As transições entre estados de uma automata é representado por

uma função δ onde $\delta : Q \times \Sigma \mapsto Q$, ou seja δ recebe dois argumentos, um estado e um símbolo e mapeia esse par a um estado. Finalmente, a automata possui um conjunto de estados de aceitação F, onde $F \in Q$, caso a automata termine sua execução em um estado $q \in F$, o string de entrada foi aceito pela automata. Formalmente, então, uma automata é uma tupla com 5 elementos $(Q, \Sigma, \delta, q_o, F)$ (Sipser 2013).

A partir da descrição formal de uma automata, podemos definir uma rotina de computação. De maneira breve, o objetivo dessa rotina é verificar que após processar o string de entrada a automata se encontra em um estado de aceitação.

Como foi visto, uma automata pode receber um conjunto de entradas que definem seu alfabeto Σ. Deseja-se definir um procedimento onde dado um string de entrada e uma automata no seu estado inicial, retorne o estado final após processar a entrada. Esse procedimento é definido como dado uma entrada $w = w_1 w_2 ... w_n \mid w_i \in \Sigma$ e uma automata M no seu estado inicial, será retornado um estado $q \in Q$. Caso o estado final seja um estado de aceitação ($q \in F$), é dito que M aceita w (Sipser 2013). Formalmente, segundo (Sipser 2013) M aceita $w = w_1 w_2 ... w_n$ se: existe uma sequencia de estados $r_0, r_1, ... r_n \in Q$ se $r_0 = q_0$; $\delta(r_i, w_{i+1}) = r_{i+1}$, para $i = 0, ..., n-1$; $r_n \in F$.

O ponto chave desse discussão é apresentado por (Sipser 2013), onde foi provado que é possível construir uma automata para qualquer regex. Sendo assim, é possível definir padrões de busca usando uma expressão regular, converter essa expressão regulara para uma automata e usar essa automata para realizar a busca pelo padrão em um string.

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

Nome	Assinatura	Descrição
genToken	Symbol -> Token	Transforma
genTokens	[Symbol] -> [Token]	Transforma
normalizeStream	[Token] -> [Token]	Adiciona o
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.

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 figura **??** 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 2 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.

A arquitetura do módulo de parse é dada pelas funções na tabela 2 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 fígura **??** 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 **??**. 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.

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 **??** 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 e comando cli

O módulo público cria uma interface para que a biblioteca seja facil de utilizar. Nesse módulo são definidas funções auxiliares e também um programa que pode ser rodado de maneira idependente.

As funções do modulo público são apresentadas na tabela **??**. Essas funções apenas fazem chamadas para as funções dos diferentes modulos da biblioteca e não introduz nenhuma logica nova.

O comando CLI é diferente pois ele deve tratar com a entrada e saída de valores, ou seja, realizar operações com o sistema. O programa criado recebe uma expressão regular como um argument do programa e le um texto para receber realizar a busca no stream de entrada padrão (stdin) e retorna o trecho do texto que satisfaz a regex.

3.5 Práticas comuns e conclusão

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 CONSIDERAÇÕES FINAIS

O objetivo desse trabalho foi expor alguns conceitos por traz do paradigma funcional e demonstrar como esses conceitos são úteis 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 explicado, a partir de exemplo, como o paradgima funcional difere do paradigma imperativo. Finalmente, foram introduzidas automatas, sua descrição formal, o algoritmo para executar uma automata e como regexes são equivalentes a automatas.

A partir desses conceitos foi explicado o método a ser utilizado nesse trabalho e um pouco mais sobre o objetivo. O trabalho tem o foco de introduzir o paradigma funcional a partir de um problema real (processamento de regex). Foi dado um exemplo de como pensar sobre um problema de maneira funcional, a partir de funções e como criar sequencias de funções para transformar a entrada no produto final. Após isso foi feita a análise do código fonte, escrito de maneira funcional, onde foram introduzidas ferramentas referentes a esse paradigma. Essa abordagem permitiu expor os pontos fracos e fortes do paradigma e também como fazer o que essas ferramentas fazem em uma linguagem imperativa.

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.

5 CRONOGRAMA

Seq.	Fases	Atividades	Projeto de pesquisa
1	1a Fase - projeto de pesquisa	Definicao do tema	Dez/19, Jan/20
1	1a Fase - projeto de pesquisa	Estudo Haskell	Dez/19 - Abr/20
2	1a Fase - projeto de pesquisa	Pesquisa Regex e algoritimos	Fev/20 - Abr/20
3	1a Fase - projeto de pesquisa	Escrever projeto de pesquisa	Jan/20 - Mar/20
4	2a Fase - TCC	Planejar Algoritimo	Abr/20
5	2a Fase - TCC	Implementar modulo	Abr/20 - Jun/20
6	2a Fase - TCC	Escrever Relatorio	Jun/20 - Jul/20
7	2a Fase - TCC	Revisao	Jun/20 - Jul/20

5.1 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.

5.1.1 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).

5.1.2 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.

5.2 Programação Funcional

Primeiro, vejamos a historia das linguagens imperativas e seu desenvolvimento. Usaremos isso para contrastar com a progressão das linguagens funcionais.

A maquina de Turing foi muito importante para a evolução dos computadores, um modelo idealizado capaz de resolver vários problemas a partir de instruções simples. Usando o modelo da maquina de Turing, os processadores foram projetados para mimicar essas operações, introduzindo instruções leitura, comparação e jump. Como consequencia, vieram as linguagens assembly, uma especie de dicionario que permite programar usando palavras ao invés de códigos de instruções. Embora assembly tenha facilitado muito a programação, ela ainda é muito próximo do hardware e extremamente trabalhoso de definir os passos para escrever um programa. Para resolver isso veio a programação estruturada, que introduziu os ifs, whiles e

fors. O processo de design de linguagens foi longo, foram diversas abstrações, construidas de maneira iterativa. Atualmente o paradigma considerado mais alto nível e muito usado é a orientação a objetos. Perceba que as linguagens imperativas foram construídas a partir de uma construção simples e que o resultado foi um processo longo para tornar esse modelo mais intuitivo aos humanos, especialmente com a programação orientada a objetos.

Enquanto as linguagens imperativas tem como descendente comum a maquina de Turing, a programação funcional tem origem em um outro modelo mátematico da computação, o calculo lambda. O calculo lambda é um outro modelo matemático, que ao invés de focar em instruções, foca na construção e aplicação de funções. A introdução do calculo lambda foge do escopo desse trabalho, porém o mesmo é a origem das linguagens funcional. Embora a maquina de Turing e o calculo lambda seja ideologicamente diferentes, foi comprovado através da hipostose de Alan Turin and Wistom Churchill que os dois são equivalentes. Logo, os problemas resolvidos usando linguagens imperativas podem também ser resolvidos pelas linguagens funcional, e vice-versa. O fato desses dois paradigmas terem uma origem extremamente diferente é ótimo para ilustrar o motivo pelo qual muitos programadores tem dificuldade com linguagens funcional. Isso se deve ao fato delas serem construídas a partir de conceitos extremamente diferentes, porém ambos tem o mesmo objetivo: resolver problemas computacional.

Segundo (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. [...]".

A programação imperativa foca em passos para resolver um problema, cada passo desse pode ser traduzido de maneira rasoavelmente direta em instruções de uma CPU. Isso faz com que o procedimeto escrito imperativemente reflita muito mais a máquina do que ao homem. O paradigma funcional tira o foco nos passos individuais para solucionar o problema e enfatiza uma estrutura para resolver o problema.

Em seguida serão abordados aspectos mais técnicos da programação funcional.

5.2.1 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. 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.

5.2.2 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.

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.

5.2.3 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**).

5.2.4 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.

6 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.

6.1 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

Nome	Assinatura	Descrição
genToken	Symbol -> Token	Transforma
genTokens	[Symbol] -> [Token]	Transforma
normalizeStream	[Token] -> [Token]	Adiciona o
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 2: Tabela de funções para o modulo de parse. Cada função é apresentada com sua assinatura e uma breve descrição.

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 figura **??** 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 2 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.

A arquitetura do módulo de parse é dada pelas funções na tabela 2 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.

6.2 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 **??** 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 **??**. 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.

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.

6.3 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 **??** apresenta as funções definidas para o módulo. Note que esse modulo não introduz um tipo de dado próprio, apenas realiza transformações.

6.4 Modulo público e comando cli

O módulo público cria uma interface para que a biblioteca seja facil de utilizar. Nesse módulo são definidas funções auxiliares e também um programa que pode ser rodado de maneira idependente.

As funções do modulo público são apresentadas na tabela **??**. Essas funções apenas fazem chamadas para as funções dos diferentes modulos da biblioteca e não introduz nenhuma logica nova.

O comando CLI é diferente pois ele deve tratar com a entrada e saída de valores, ou seja, realizar operações com o sistema. O programa criado recebe uma expressão regular como um argument do programa e le um texto para receber realizar a busca no stream de entrada padrão (stdin) e retorna o trecho do texto que satisfaz a regex.

6.5 Práticas comuns e conclusão

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.

Referências

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

Cox, Russ (jan. de 2007). *Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...)* URL: `https://swtch.com/~rsc/regexp/regexp1.html`.

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

Klein, Philip N. (2015). *Coding the matrix: linear algebra through applications to computer science*. Newtonian Press.

Miran, Lipovača (2012). *Learn you a Haskell for great good!: a beginners guide*. No Starch Press.

Python, Software Foundation (s.d.). *6.2. re - Regular expression operations¶*. URL: `https://docs.python.org/3.5/library/re.html`.

Sipser, Michael (2013). *Introduction to the theory of computation*. Course Technology Cengage Learning.

Stack, Overflow (s.d.). *Stack Overflow Developer Survey 2019*. URL: `https://insights.stackoverflow.com/survey/2019`.