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

Bruno Gomes

28 de março de 2020

1 INTRODUCAO

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-overflow**). 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 irá ser discutido o paradigma de programação funcional, uma alternativa ao paradigma imperativo que domina o mercado.

O foco desse trabalho será a criação de um módulo em Haskell para realizar buscas em textos usando expressões regulares. Durante essa jornada serão feitas comparações entre algoritmos escritos de maneira imperativa e funcional, usando as linguagens Python e Haskell, respectivamente. As expressões regulares partem da teoria da computação, mais especificamente da teoria das automatas. Será definida a teoria as automatas, como elas são capazes de processar expressões regulares e finalemente será abordado a implementação do módulo em Haskell para a busca em texto.

Embora o paradigma funcional seja muito menos disseminado, ele é de extrema importância e possuem um grande impacto fora de seu nicho. as descobertas e inovações Diferentes inovações e descobertas no paradgima funcional, de certo modo infecta as linguagens imperativas. Como exemplo disso temos que a partir da versão 8 do Java, foram introduzidas interfaces funcionais e *arrow functions*, conceitos esses que surgiram a programação funcional. Na linguagem Python, outro gigante da programação imperativa, existem as *list comprehenssions*, uma maneira idiomática de se construir listas em Python. Esse recurso muito amado da linguagem foi inspirado em um recurso muito similar que existe na linguagem Haskell.

Em conclusão, embora as linguagem funcionais sejam muito menos comum, elas definitivamente deixaram e continuam deixando marcas nos gigantes da pro-

gramação. Elas transcendem seu pequeno nicho de usuários e afetam a grande maiorias das pessoas que produzem código regularmente, mesmo que muitos não tenham ciência disso. Sendo assim, esse trabalho tem como objetivo introduzir o paradigma funcional, focando na linguagem Haskell, realizando comparações entre os dois paradigmas 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 descrito pela expressão regular, pode-se dizer que essa expressão regular corresponde a esse string."(**cox**).

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. O token 2 indica uma repetição, sendo equivalente à regex [0-9][0-9], ou seja dois carácteres numericos. O carácter : é interpretado como o símbolo dois pontos 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-characteres para descrever operações. Em (**mastering**), o autor discute as diferenças em regex entre as linguagens: PHP, .NET, Java e Perl. Na documentação oficial da linguagem Python (**python-re**) 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. Em resumo, existem vários dialetos porém os 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 o tópico sobre 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 (**matrix**) 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**), "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 propoem 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". Um problema muito interessante para abordarmos a ideia de "transformações"pode ser o algorítmo de *merge sort*.

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 cuja e só aceitar as combinações cuja soma do primeiro nome e ultimo nome não excede 15 carácteres (incluindo o espaço). Para isso, ao invez de analisar o processo para processar esses dados uma boa ideia é pensar em como manipular os dados para se obter o resultado esperado. 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 (**lipovaca**). Para identificar listas em Haskell é usado o símbolo [], ou seja [Char] indica uma lista de carácteres 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 uma função $combinarNomes :: [String] -> [String]$, ou seja uma função que recebe uma lista de Strings e retorna uma lista de Strings. Em seguida, iremos declarar a função que representa 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 (**lipovaca**) 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 exmplo 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, da maneira que as expressões regulares foram introduzidas não permite uma implementação direta delas em 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 (**comp**), as automatas modelam um computador com uma quantidade minuscula 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 $q_p \in Q$. Automatas recebem entradas a partir de simbolos, o conjuto de todos os símbolos reconhecidos por uma automata constroi um conjunto Σ chamado do 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. As transições entre estados de uma automata é representado por uma função δ onde $\delta : QX\Sigma \mapsto Q$, 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 $(Q, \Sigma, \delta, q_o, F)$ (**comp**).

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

Como foi visto, uma automata recebe um conjunto de entradas que define seu alfabeto Σ. Deseja-se definir um procedimento onde dado um string de entrada w cujo todos elementos $w_i \in \Sigma$ e uma automata M no seu estado inicial, se após ler todos os elementos w_i de w a automata está em um estado $q \in F$, ou seja um estado

de aceitação. Caso essa proposição seja verdadeira, é dito que M aceita w (**comp**). Formalmente, segundo (**comp**) M aceita $w = w_1w_2...w_3$ 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 (**comp**), onde foi provado que é possível construir uma automata para qualquer expressão regular. 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 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

4 CONSIDERACOES FINAIS

O objetivo desse trabalho foi expor alguns conceitos por traz do paradigma funcional e demonstrar como esses conceitos sao uteis atraves da construcao de um modulo de processamento de expressoes regulares. O modulo foi implementado usando a linguagem Haskell, uma linguagem funcional pura.

Primeiramente foi abordado as origens da programacao funcional, tal como as principais diferencas entre o paradigma funcional e imperativo. Ainda, foram introduzidos alguns conceitos exclusivos das linguagens funcionais, tais como Monads. Foi abordado de maneira abrangente o conceito por traz das expressoes regulares (regexes) e alguns casos onde sao uteis, juntamente com parte da sua historia. Por ultimo foi discutido diferentes metodos para se resolver o problema computacional referente a pesquisa de expressoes regulares.

Usando os conceitos apresentados foi demonstrado o processo de construcao do modulo de regex.