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

Bruno Gomes

21 de setembro 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 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 de um 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 *lambda expressions*, conceitos esses que surgiram a programação funcional (Oracle s.d.).

Em conclusão, embora as linguagem funcionais sejam muito menos comum, elas definitivamente deixaram e continuam deixando marcas nos gigantes da programação. Elas transcendem seu pequeno nicho de usuários e afetam a grande maioria das pessoas que produzem código regularmente, mesmo que muitos não tenham ciência

disso. Sendo assim, esse trabalho tem como objetivo introduzir o paradigma funcional, através da linguagem Haskell, comparando 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 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.

2.4 METODOLOGIA

Como explicado na introdução, o foco destre trabalho é demonstrar alguns elementos da programação funcional. Para isso, foi escolhido o problema de implementar um módulo de busca de strings usando expressões regulares. Será escolhido trechos de código especialmente interessante do módulo escrito que serão explciados a fundo.

Foi visto que uma expressão regular pode ser convertida em uma automata equivalente. Sendo assim, o problema possui duas tarefas: criar submódulo para converter uma regex em uma automata e implementar um submódulo que permita criar e operar uma automata. Serão definidas as arquiteturas de cada submódulo tal como o encadeamento de funções que serão chamadas para resolver cada problema, análogo ao que foi feito anteriormente. Será explicado, de maneira alto nível, o que cada função faz baseada em suas entradas e saídas. Isso irá motivar a introdução de tipos de dados únicos a programação funcional.

Além dessa inspeção de "caixa preta"das funções, os pontos principais do módulo será explicado em detalhe, o que permitirá aa análise de conceitos importantes no paradigma funcional. Será feita uma comparação entre trechos escrito de maneira funcional e imperativa. Essa comparação tem dois objetivos: introduzir conceitos referentes a linguagem funcional e identificar em quais situações um código funcional é mais simples, ou mais complexo, que o seu equivalente de maneira imperativa.

Para introduzir os conceitos do paradigma funcional, o código irá ser projetado tal que demonstre as diferentes ferramentas que compõe a caixa de ferramentas de um programador funcional. As ferramentas simples abordarão conceitos comos imutalidade e recursão e as ferramentas mais complexas irão introduzir abstrações muito perculiares da programação funcional tal como *Functors* e *Applicative Functors*. A metodologia escolhida tem como objetivo ser transparente quanto aos lados bons e ruins da programação funcional e também auxiliar a associação do paradigma imperativo ao funcional. Dessa forma, um leitor familiar com programação imperativa poderá entender como um problema resolvido de maneira imperativa pode ser traduzido para um algoritimo funcional.

Em conclusão, o trabalho irá resolver o problema de criar um módulo de procura em texto usando expressões regulares. O problema será quebrado em funções, exemplificando como resolver um problema a partir de funções ao invez de passos. O código fonte do módulo criado será usado para introduzir conceitos sobre o paradigma funcional e familiarizar o leitor com algumas ferramentas. Ao mesmo tempo, trechos de códigos funcionais serão comparados com seu equivalente escrito em uma linguagem imperativa, o que permitira associar conceitos imperativos a funcionais e expor os pontos fortes e fracos desse paradigma.

3 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.

4 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

4.1 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.

4.1.1 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).

4.1.2 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.

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.

Oracle (s.d.). *What's New in JDK 8*. URL: https://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html.

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.