**EXPRESSÕES REGULARES EM LINGUAGEM FUNCIONAL: MÓDULO DE BUSCA USANDO EXPRESSÕES REGULARES IMPLEMENTADO EM HASKELL**

GOMES, Bruno

LIMA, José Airton G. de

**RESUMO**

---

**Palavras chave:** Expressão Regular, Programação funcional, Haskell

**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 (2019), as 5 linguagens de programação mais usadas são: JavaScript, Python, Java, Linguagens de script (Bash, Powershell, Shell) e C#.

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

O foco deste 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 finalmente 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 paradigma 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).

Em conclusão, embora as linguagens 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, este trabalho une dois temas: expressões regulares e programação funcional. Esses temas serão, primeiramente, discutidos separadamente, e em seguida será feito uma prévia de como será feita a construção do motor de processamento de expressões regulares.

2.1 Introdução às expressões regulares

Expressões regulares, também conhecidas como regex (da junção do nome em inglês, regular expression) são utilizadas para realizar buscas complexas sobre strings. Para Cox (2007, tradução nossa), "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.".

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 todos os strings que correspondem a um horário em um texto. A escrita de um horário 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}. Em palavras, os símbolos [0-9] representa qualquer caractere numérico entre 0 e 9. De maneira geral os símbolos [] representam um conjunto de caracteres (PYTHON SOFTWARE FOUNDATION). O token {2} indica uma repetição, sendo equivalente à regex [0-9][0-9], ou seja dois caracteres numéricos. Os símbolos {} são usados para representar repetição (PYTHON SOFTWARE FOUNDATION). 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 dígito de 0-9. Podemos ver que esse formato é exatamente o formato definido anteriormente.

É importante ressaltar que existem inúmeras 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 é dito que o dialeto usado é baseado nas expressões regulares da linguagem Perl com alguns adicionais (PYTHON SOFTWARE FOUNDATION). 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, realizar busca 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 características de maneira resumida. Há muito a se falar sobre esse assunto pois ele é extenso 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,

Programação funcional é: um método para construção de programas que enfatiza funções e suas aplicações ao invés de comandos e suas execuções; programação funcional faz uso de notações matemática simples que permite que problemas sejam descritos de maneira clara e concisa. (2014, p. 1, tradução nossa)

A programação imperativa foca em passos para resolver um problema, cada passo desse pode ser traduzido de maneira razoavelmente direta em instruções de uma CPU. Isso faz com que o procedimento escrito imperativamente 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 ideia considere o seguinte problema. Dado uma lista de nomes, com nome, nome do meio e sobrenomes, crie uma lista com todas as combi- nações de primeiro nome e último nome, ignorando nomes do meio e aonde as com- binações cuja soma do primeiro e último nome não exceda 15 caracteres (incluindo o espaço). Para isso, ao invés 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:

a. Separar cada nome da lista de nomes nos espaços e armazenar os nomes uma

lista.

b. Filtrar listas que só possuem um elemento (somente um nome).

c. Filtrar listas e remover nomes do meio.

d. Criar uma lista de nomes e uma lista sobrenomes.

e. Realizar o produto cartesiano sobre essa lista e gerar uma lista de tuplas.

f. Transformar 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 indicadas 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 conjuntos 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 matemática 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 qo ∈ Q. Automatas recebem entradas a partir de símbolos, o conjunto 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 símbolo α ∈ Σ. As transições entre estados de uma automata é representado por uma função δ onde δ : Q×Σ ↦ 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 ∈ Q, caso a automata termine sua execução em um estado q ∈ F, o string de entrada foi aceito pela automata. Formalmente, então, uma automata é uma tupla com 5 elementos (Q,Σ, δ, qo,F) (Sipser, 2013).

A partir da descrição formal de uma automata, podemos definir uma rotina de computação. De maneira breve, o objetivo desta 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 = w1w2...wn |wi ∈Σ euma automata M no seu estado inicial, será retornado um estado q ∈ Q. Caso o estado final seja um estado de aceitação (q ∈ F), é dito que M aceita w (Sipser, 2013). Formalmente, segundo Sipser (2013) M aceita w = w1w2...wn se existe uma sequência de estados r0,r1, …, rn ∈ Q se r0 = q0; δ(ri,wi+1) = ri+1, para i =0, ..., n − 1; rn ∈ F.

O ponto chave dessa 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 regular para uma autômata e usar essa automata para realizar a busca pelo padrão em um string.

2.4 METODOLOGIA

Como explicado na introdução, o foco deste 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 explicados 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 a 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 às linguagens 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 como imutabilidade e recursão e as ferramentas mais complexas irão introduzir abstrações muito peculiares 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. Desta forma, um leitor familiar com programação imperativa poderá entender como um problema resolvido de maneira imperativa pode ser traduzido para um algoritmo 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 invés 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 permitirá associar conceitos imperativos a funcionais e expor os pontos fortes e fracos desse paradigma.

**3 METODOLOGIA**

A biblioteca foi separada em 3 módulos internos e um módulo público. Os módulos internos contêm todo as partes internas responsáveis por transformar a regex em forma de string que o usuário envia em uma máquina de estados e simular essa máquina de estados.

Os módulos internos são: módulo de parse léxico, módulo de automatas e módulo de conversão. Nessa seção serão abordados cada um desses módulos em detalhe, tal como práticas de desenvolvimento comum a todos os módulos. É importante ressaltar que näo será feita um análise linha a linha do código escrito, somente os pontos interessantes. O código fonte completo está disponível no apêndice 1 ou no website do autor: <http://github.com/lodek/regex-engine>.

3.1 PRÁTICAS DE DESENVOLVIMENTO E ESTRUTURA

Seção incompleta.

Escrever sobre HUnit, testes unitários, QuickTest.

3.1 PARSER LÉXICO

O parser léxico é o primeiro estágio da pipeline que irá permitir a construção de uma automata de 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 ligam a outros nós 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 eliminar as ambiguidades referentes à ordem das operações, sem precisar fazer uso de parênteses para indicar a qual grupo de caracteres um operador opera sobre.

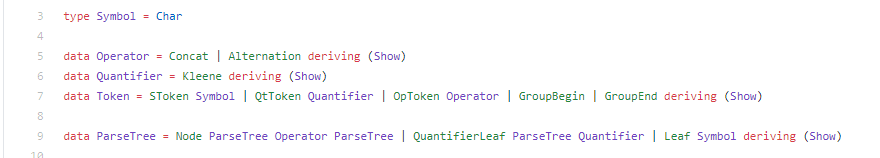
A implementação desse módulo em Haskell foi feita usando um conjunto de funções. A tabela X indica as funções desse módulo, junto com seus tipos de entrada, saída e uma breve descrição sobre cada uma.

No paradigma de orientação a objetos é muito importante criar um modelo em software de um sistema físico tal que ele seja facilmente composto. Essa característica é chamada de compossibilidade e os objetos, junto de suas interfaces públicas, ditam o quão fácil é compor um código. Embora na programação funcional não existam objetos no sentido de dados mais comportamentos, ainda existe a necessidade de criar um modelo de dado que armazena as informações necessárias. Na linguagem Haskell isso é feito através dos tipo de dados algébricos.

O tipo de dado algébrico são uma maneira de expressar um conjunto de valores relacionados em Haskell. Essa construção é bem peculiar e poderosa. na linha 5 da figura X, é definido um novo tipo de dado, “Operator”, esse tipo de dado pode ter 2 valores “Concat” ou “Alternation”. Note que o tipo Operator está definindo um conjunto de nomes que, enumeram quais operações são possíveis na regex. Nessa caso, o tipo Operator é bem similar a um enum, muito comum ao que existe em linguagens imperativas, porém isso é somente o começo.

Na linha 7 é definido o tipo Token. Token representa as diferentes categorias que os caracteres de uma regex pode representam. Um token pode representar: um símbolo, uma operação, uma quantificação e o começo ou fim de um agrupamento. Note que o tipo token define um token de quantificação, Token eToken possui um Quantifier. Mais especificamente, a construção Token do tipo token, armazena um valor do tipo Quantifier. Perceba que o mesmo acontece para os tokens de símbolo e operação. Percebe-se que os tipos de dados algébricos não são análogos nem a enums nem a classes, pois um enum não pode conter campos diferentes para cada valor do enum é um objeto criado a partir de uma classe necessariamente contém todos os atributos da classe.

Por último, na linha 9 é definido o tipo Parse Tree, que possui três construções. Note que a construção Node possui 3 valores, Parse Tree, Operator e Parse Tree, ou seja, isso é uma definição recursiva. Parse Tree pode conter ele mesmo como uma de suas construções. Essa peculiaridade faz com que a criação de árvores em Haskell seja um tarefa trivial, basta declara um tipo de dado com diferentes construtores e que um construtor recebe um valor do próprio tipo. O tipo Parse Tree modela a estrutura intermediária que será convertida para uma automata.



Sendo assim, esse módulo define um conjunto de funções que transforma uma String de entrada em uma Parse Tree. O módulo contém a função build Tree, que realiza toda a cadeia de transformações para gerar essa árvore, abstraindo as funções intermediárias para tal trabalho. A função build Tree é definida na figura X.

A função recebe um String e retorna uma Parse Tree. Nessa função é feito o uso de uma sintaxe especial do Haskell a composição de funções, indicada pelo operador “.”. A composição em Haskell é análogo a composição em matemática, ou seja, se f :: a -> b, g :: b -> c, é possível definir uma função h tal que h seja equivalente a aplicar f e depois g. Essa ideia é expressada em Haskell usando o operador de composição.



Por fim, vemos que a função recebe um String e partir dele é gerado uma lista de tokens, essa lista de tokens é processada e normalizada, em seguida a funcao sortAndTreefy itera sobre essa lista de tokens e separa tokens de operação e cria as folhas da árvore final. Depois, essa lista de árvores e Operadores é separada em duas listas, uma de operadores e uma de árvores, por último é aplicado os operadores sobre essas árvores, em efeito compondo as folhas em galhos, até gerar a árvore final. Por último, a função head retorna o primeiro elemento de uma lista. Essa lista contém a única árvore que é a combinação de todas as folhas e operadores.

Isso conclui o módulo de parse da biblioteca. Os tipos de dados algébricos foram de tremenda importância para garantir que o código escrito fosse compreensível e enxuto. Essa construção peculiar que existe em poucas linguagens é muito poderosa pois permite, em uma mesma declaração de tipo de dado, declarar constantes, construções que armazenam variáveis de tipo heterogêneo e a criação de estruturas recursivas, uma maneira muito concisa de uma árvore.

3.2 AUTOMATAS

3.3 CONVERSOR

3.4 INTERFACE PÚBLICA

**4 RESULTADOS E DISCUSSÕES**

**5 CONSIDERAÇÕES FINAIS**

Neste trabalho foi exposto alguns conceitos por trás do paradigma funcional e demonstrar como esses conceitos são úteis através da construção de um módulo de processamento de expressões regulares. O módulo foi implementado usando a linguagem Haskell, uma linguagem funcional.

Primeiramente, foi explicado o que são regexes do ponto de vista de um usuário e quais problemas elas resolvem. Em seguida foi explicado, a partir de exemplo, como o paradigma funcional difere do paradigma imperativo. Finalmente, foram introduzidas automatas, sua descrição formal, o algoritmo para executar uma autômata e como regexes são equivalentes a automatas.

A partir desses conceitos foi explicado o método a ser utilizado neste trabalho e um pouco mais sobre o objetivo. O trabalho introduziu 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 sequências 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 tentou mostrar um mundo diferente da programação, um mundo que vem sido incorporado às linguagens imperativas, mesmo que muitos programadores desconhecem suas origens.

**REFERÊNCIAS**

BIRD, Richard. ***Thinking functionally with Haskell****.* 1. Ed. Cambridge: Cambridge University, 2014.

COX, Russ. ***Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...).*** 2007Disponivel em: <[https://swtch.com/~rsc/regexp/regexp 1.html](https://swtch.com/~rsc/regexp/regexp1.html)>. Acesso em: 29 mar. 2020.

FRIEDL, Jeffrey E. F. ***Mastering regular expressions.*** 3. ed. Sebastopol: OReilly, 2009.

KLEIN, Philip N. ***Coding the matrix: linear algebra through applications to computer science****.* 1. ed. Newton: Newtonian Press, 2015.

MIRAN, Lipovača. ***Learn you a Haskell for great good! A beginners guide.*** 1. ed. San Francisco: No Starch Press, 2012.

ORACLE. ***What’s New in JDK 8.*** Disponivel em: <<https://www.oracle.com/technetwork/>java/javase/8-whats-new-2157071.html>. Acesso em: 29 mar. 2020.

PYTHON SOFTWARE FOUNDATION. ***6.2. re - Regular expression operations***. Disponivel em: <h[ttps://docs.python.org/3.5/library/re.html](https://docs.python.org/3.5/library/re.html)>. Acesso em: 29 mar. 2020.

SIPSER, Michael. ***Introduction to the theory of computation.*** 3. ed. Boston: Cengage Learning, 2013

STACK OVERFLOW. ***Stack Overflow Developer Survey 2019.*** 2019. Disponivel em: <https://insights.stackoverflow.com/survey/2019>. Acesso em: 29 mar. 2020.