Desmistificando Expressões Regulares

RESUMO

O mundo das expressões regulares ainda é bastante obscuro para muitos que as utilizam. Isso se deve a pouca documentação existente e da pouca didática aplicada ao assunto. Esse artigo tem como objetivo desmistificá-las, mostrando simplicidade tanto na criação como na utilização destas.

PALAVRAS-CHAVE: expressões regulares; simplicidade; criação; utilização.

José Antônio da Cunha
Gerência Educacional de Tecnologia da Informação – CEFET-RN
Av. Salgado Filho, 1559 Morro Branco CEP 59.000-000 Natal-RN
E-mail: jcunha@cefetrn.br

.

1. Introdução

Mesmo se tratando de um assunto relativamente antigo, as expressões regulares, as quais chamarei daqui pra frente de regex, ainda são de difícil entendimento para muitos, seja por falta de documentação em português, a qual é praticamente inexistente, ou por dificuldade de entender alguns que existem, os quais não são nada didáticos.

A grande maioria das pessoas que as utilizam não entende seu funcionamento nem como estas são criadas, apenas copiam da internet ou de livros expressões pré-prontas e as aplicam.

Este artigo tem como objetivos difundir o conhecimento sobre as regex, mostrar suas utilidades, assim como desmistificar sua montagem e aplicação.

O artigo está dividido da seguinte forma: no item 2 descrevo sobre as regex, contando um pouco de sua história, mostrando suas utilidades e suas micros partes, os metacaracteres; no item 3 apresento em forma de passo-a-passo exemplos de como montar uma regex; no item 4 mostro alguns aplicativos e linguagens de programação que dão suporte às expressões, com exemplos utilizando C# e Java; no item 5 traço conclusões sobre o trabalho; e no item 6 apresento as minhas referências bibliográficas.

2.0 Expressões Regulares

Segundo Jargas (2001), uma expressão regular é um método formal de se especificar um padrão de texto.

Essa especificação de um padrão tem como principais objetivos realizar busca, substituição ou validação. Este último é o mais utilizado pelos programadores, pois pode utilizar as expressões para validar padrões como: data, hora, número IP, telefone, CPF, RG, e-mail e qualquer padrão que seja necessário.

O funcionamento interno das expressões regulares foge do escopo principal deste artigo, pois depende da teoria dos autômatos finitos determinísticos e não-determinísticos além de teoria de linguagens formais.

2.1. História

Tudo começou nos anos 40, quando Warren McCulloch e Walter Pitts, neurologistas, descreveram o funcionamento dos neurônios. Mais tarde, Stephen Kleene passou o estudo para uma notação matemática [2].

Após o trabalho de Kleene, as expressões eram utilizadas apenas para fins matemáticos. Até que em 1968 as regex entraram na computação, sendo utilizadas por um algoritmo de busca no editor qed do Unix.

Mais tarde o qed virou ed, e com este veio a ferramenta grep, na qual “re” representa a expressão regular propriamente dita, o “g” especifica que o comando será feito em todo o arquivo (comando global), e o “p” para imprimir as linhas com subcadeias coincidentes.

As regex surgiram entre as linguagens de programação em 1986, quando foi criado o pacote regex para a linguagem C. A partir daí as demais linguagens começaram a dar suporte também, como veremos no decorrer do artigo.

3. Os Metacaracteres

José Antônio da Cunha
Gerência Educacional de Tecnologia da Informação – CEFET-RN
Av. Salgado Filho, 1559 Morro Branco CEP 59.000-000 Natal-RN
E-mail: jcunha@cefetrn.br

Os metacaracteres são caracteres com funções especiais. Existem 14 metacaracteres primitivos (deles pode-se gerar outros), além de classes para facilitar algumas funções, tal como a POSIX. Porém nesse artigo me limitarei a falar sobre os primitivos. Esses 14 metacaracteres podem ser divididos em 04 (quatro) categorias: representantes, quantificadores, âncoras, e outros. Nesta seção veremos qual a função de cada metacaractere, assim como alguns exemplos básicos sobre cada um.

3.1. Representantes

3.1.1. O ponto ".": Tem a função de poder representar qualquer caractere.

Exemplos:

Br.sil => Brasil, Br2sil, Br sil, ...
Sk. => Sk8, Ski, Sk ,...

3.1.2. A lista "[...]": Limita os caracteres que são permitidos.

Exemplos:

5[,.]467 => 5,467 ou 5.467.
Gra[nm]a => Grana ou Grama

3.1.3. A lista negada "[^...]": Limita os caracteres que são proibidos.

Exemplos:

[^A-Z] => d, 7, 0, c, [, @, \, ...
[^A-Za-z^] => Tudo que não seja letra (maiúscula ou minúscula) e o acento circunflexo.

3.2. Quantificadores

3.2.1. O opcional "?": Pode ter zero ou uma aparição.

Exemplos:

</?[BIPbip]> => <B>, <I>, <P>, , , , </B>, </I>, </P>, , , .
anda[r]? => anda ou andar.

3.2.2. O asterisco "*": Pode ter zero, ou mais aparições.

Exemplos:

9*5 => 5, 95, 995, 9995,...
Ca[rr]*o => Cão, Carro, Carrrro, ...

3.2.3. O mais "+": Pode ter uma ou mais aparições.

Exemplos:

5+8 => 58, 558, 5558,...
Alo[w]+ => Alow, Aloww, Alowww,...

3.2.4. As chaves "{n, m}": Pode ter de n à m aparições.

José Antônio da Cunha
Gerência Educacional de Tecnologia da Informação – CEFET-RN
Av. Salgado Filho, 1559 Morro Branco CEP 59.000-000 Natal-RN
E-mail: jcunha@cefetrn.br

Exemplos:

RN {1,3} => RN, RNRN, RNRNRN.
[10] {2,4} => 1010, 101010, 10101010.

Através das chaves o desenvolvedor pode gerar qualquer um dos três quantificadores citados anteriormente. A Tabela I mostra a universalidade das chaves.

Tabela I – Universalidade das chaves

Metacaractere	Representação com chaves
?	{0,1}
*	{0,}
+	{1,}

3.3. Âncoras

3.3.1.O circunflexo "^": Marca o início de uma linha.

Exemplos:

^[0-9] => Casa linhas que começam com um número.
^a => Casa linhas que começam com a letra 'a' minúscula.

3.3.2.O cifrão "$": Marca o fim de uma linha.

Exemplos:

[0-9]$ => Casa linhas que terminam com um número.
-$ => Casa linhas que terminam com '-'.

3.3.3.A borda "\b": Limita o início e/ou o fim da uma palavra.

Exemplos:

\bcasa => casa, casar, casamento, casarão,...
casa\b => casa, descasar, ...
\bcasa\b => casa

3.4. Outros

3.4.1.O escape "\": "Escapa" o metacaractere, deixando-o com o valor de caractere literal.

Exemplos:

\. => Casa o ponto literal.
* => Casa o asterisco literal.

3.4.2.O ou "|": Casa um subpadrão ou outro.

Exemplos:

Boa-tarde | Boa-noite => Boa-tarde ou Boa-noite.
ftp:// | http:// => ftp:// ou http://.

José Antônio da Cunha
Gerência Educacional de Tecnologia da Informação – CEFET-RN
Av. Salgado Filho, 1559 Morro Branco CEP 59.000-000 Natal-RN
E-mail: jcunha@cefetrn.br

3.4.3.O grupo “(...)”: Agrupa metacaracteres, caracteres e até outros grupos.

Exemplos:

(Hello)+ => Hello, HelloHello, HelloHelloHello,...
Mensag(em | eiro) => Mensagem ou Mensageiro.

3.4.4.O retrovisor “\[1-9]”: Utilizado para atuar nos grupos, o retrovisor permite que você repita algo obtido em um determinado grupo. Pode-se fazer uso de 9 retrovisores. A ordem dos retrovisores deve ser percebido da esquerda para a direita.

Exemplos:

repete \1 => repete repete
(((t)e)r) = \1, \2, \3 => ter = ter, te, t

4. Montagem de Expressões

Nesta seção iremos aprender a montar nossas expressões regulares. Verificaremos que não se pode pensar em uma regex toda de uma vez, e sim uma parte de cada vez. Veremos abaixo dois exemplos práticos: e-mail e ponto flutuante, lembrando que tudo que será visto aqui sempre pode ser melhorado e especificado para a necessidade de cada um.

4.1. E-mail

Vamos imaginar um sítio que contenha um formulário de cadastro. Neste formulário existe um campo de e-mail a ser preenchido e, portanto validado.

Em primeiro lugar temos que pensar: O que deve existir antes do @? Podemos ter letras (minúsculas), hífen (-), sublinha (_) , ponto (.) e números; isso desde o início da linha, tendo que existir ao menos um desses elementos e sem limite pra terminar.

Com esse pensamento chegamos a seguinte expressão: ^[a-z.-_0-9]+@

Agora vamos imaginar o que vem após o ‘@’: normalmente temos o nome de um domínio adicionado de “.com”, ou “.org”, ou “.gov”, dentre outros. Portanto teremos o mesmo padrão obtido antes do ‘@’.

Após a segunda etapa temos a seguinte expressão: ^[a-z.-_0-9]+@[a-z.-_0-9]+

Para finalizar ainda falta o código do país, que pode ou não existir. Quando existir sabemos que a maioria utiliza apenas dois caracteres, logo depois de um ponto, e vem no final do e-mail, chegando a seguinte expressão:
\.[a-z]{2}$

Como esse código é opcional temos: (\.[a-z]{2})?$

Feito isso, obtemos nossa expressão final: ^[a-z.-_0-9]+@[a-z.-_0-9]+ (\.[a-z]{2})?$

4.2. Ponto Flutuante

Agora vamos tentar montar uma regex que case um número em ponto flutuante.

Inicialmente sabemos que um número em ponto flutuante pode ter ou não sinal, encontrando: [-+]?

José Antônio da Cunha
Gerência Educacional de Tecnologia da Informação – CEFET-RN
Av. Salgado Filho, 1559 Morro Branco CEP 59.000-000 Natal-RN
E-mail: jcunha@cefetrn.br

Depois temos que ter ao menos um algarismo entre de 0 a 9: [-+]?[0-9]+

Em um número ponto flutuante pode existir ou não um ponto: [-+]?[0-9]+\.?

E se no lugar de ponto eu também aceite a vírgula? Encontramos a seguinte regex: [-+]?[0-9]+(\.|,)?

E para finaliza, pode existir ou não a parte decimal, gerando nossa regex final: [-+]?[0-9]+(\.|,)?[0-9]*

5. Aplicações Práticas

As expressões regulares viraram uma importante ferramenta em vários aplicativos e linguagens de programação.

Citando alguns editores de texto os quais dão suporte as regex temos o Emacs, o MS Word, e o Vim, dentre outros. Pensando em termos de banco de dados, acredito que quase 100% dos SGBDs de hoje dão suporte às expressões, porém na internet só encontrei documentação fácil para PostGreSQL e MySQL.

Já entre as linguagens de programação temos Python, Perl, PHP, JavaScript, C# e Java dentre as principais que dão suporte às expressões.

Um grande problema que existe e que atrapalha bastante a utilização das regex é a falta de padronização entre os aplicativos e linguagens que permitem sua utilização. Portanto é aconselhável sempre dar uma olhada nas tabelas disponibilizadas pelas tecnologias para não haver engano.

A seguir irei realizar dois exemplos práticos com as regex utilizando as linguagens C# e Java.

5.1 C#

Imagine uma pequena empresa a qual guarda os dados dos seus funcionários dentro de um arquivo txt. Um dia o chefe chega pra você e te pede para montar um formulário igual ao da Figura 1, no qual ele seleciona o mês do ano e é listado ao lado as informações dos aniversariantes daquele mês.

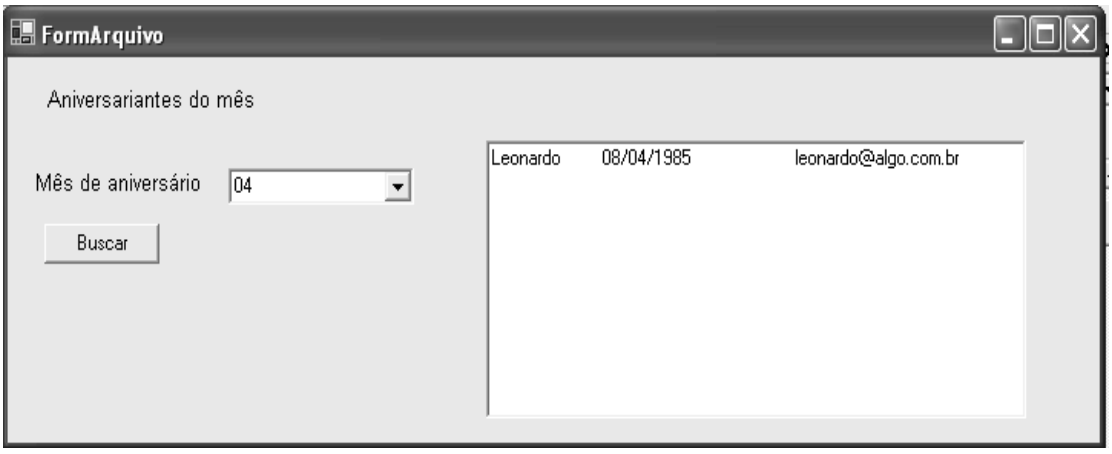

Figura 1 – Formulário dos aniversariantes do mês

Você tem várias opções para resolver o problema, porém uma ótima saída é utilizar a biblioteca Regex do C# e resolver o problema. Abaixo segue o código do evento Click do botão Buscar.

José Antônio da Cunha
Gerência Educacional de Tecnologia da Informação – CEFET-RN
Av. Salgado Filho, 1559 Morro Branco CEP 59.000-000 Natal-RN
E-mail: jcunha@cefetrn.br

```
private void btnBuscar_Click(object sender, System.EventArgs e)
{
        this.txtResultados.Text = "";
        FileStream fs = File.OpenRead("funcionarios.txt");
        StreamReader arq = new StreamReader(fs);
        string lendo = "";

        /*criando o padrão para capturar as linhas que contém o mês selecionado          no combo box */

        Regex reg = new Regex(@"[0-9]+/"+this.comboMes.SelectedItem.ToString()+
        "/[0-9]+");

        while(lendo != null)
        {
                Match a = reg.Match(lendo);

                /*se a string 'lendo' casar com o padrão, a linha será
                  colocada no txtResultados */
                if(a.Success)
                   this.txtResultados.Text += lendo+"\t\t";
                //fazendo a string lendo ser a próxima linha no arquivo
                lendo = arq.ReadLine();
        }
        fs.Close();
}
```

5.2 Java

Agora vamos imaginar que um comerciante guarda o valor dos seus produtos em um arquivo txt. Ele fará uma promoção de 10% para os produtos com preço entre R$ 100,00 e R$ 200,00. Pensando numa solução simples você pode utilizar o pacote java.util.regex do Java. Segue abaixo o código da classe BuscaSubstitue.java.

```
import java.io.FileInputStream;
import java.io.IOException;
import java.nio.CharBuffer;
import java.nio.MappedByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.charset.Charset;
import java.util.regex.*;

public class BuscaSubstitui {

        public static void main(String[] args) {
                // TODO Auto-generated method stub

            try {
                  //Criando um padrão para casar preços de R$ 100 a R$ 199
                  Pattern pattern = Pattern.compile("R\$ 1[0-9]{2}");
                  //Criando um objeto Matcher pra obter os casamentos
                  Matcher matcher = pattern.matcher(fromFile("arquivo.txt"));

                  StringBuffer buf = new StringBuffer();
                  boolean found = false;
                  while ((found = matcher.find())) {

                      String replaceStr = matcher.group();
                      //Alterando os valores casados para ganhar um desconto de 10%
```

José Antônio da Cunha
Gerência Educacional de Tecnologia da Informação – CEFET-RN
Av. Salgado Filho, 1559 Morro Branco CEP 59.000-000 Natal-RN
E-mail: jcunha@cefetrn.br

```
                    double novoValor = Double.parseDouble(replaceStr)*0.9;

                    //Substituindo o valor do preço
                    replaceStr = novoValor+"";
                    //Fazendo o novo valor ser adicionado ao restante do conteúdo do arquivo
                    matcher.appendReplacement(buf, replaceStr);
                }
                matcher.appendTail(buf);
                //Imprimindo o conteúdo com as substituições
                System.out.println(buf.toString());

            }catch (IOException e) {
                    // TODO: handle exception
            }
        }

    //Método para transformar o conteúdo do arquivo em um CharSequence
    public static CharSequence fromFile(String filename) throws IOException {
        FileInputStream fis = new FileInputStream(filename);
        FileChannel fc = fis.getChannel();
        MappedByteBuffer bbuf = fc.map(FileChannel.MapMode.READ_ONLY, 0, (int)fc.size());
        CharBuffer cbuf = Charset.forName("8859_1").newDecoder().decode(bbuf);
        return cbuf;
    }

}
```

5. CONCLUSÕES

Através deste artigo conseguimos ver como montar e aplicar as expressões regulares de forma simples e objetiva, mostrando que após o aprendizado o uso das regex é bastante fácil e útil de ser utilizado.

Vimos também que não se deve entender uma expressão regular com um todo, e sim entender suas partes em separado, para só assim ter uma visão ampla dela.

A maior dificuldade enfrentada no uso das regex é a falta de padronização no uso dos metacaracteres, já que cada aplicativo resolve fazer suas adaptações, fazendo com que sempre o usuário das regex tenham que ficar vendo as documentações destes aplicativos para não se confundir.

Mostramos dois exemplos simples em linguagens de grande uso hoje em dia pelos desenvolvedores: C# e Java, e mostramos que suas API's são bastante auto-explicativas e simples de se usar.

Espero que as publicações (em português) sobre as expressões regulares possam aumentar e o assunto se tornar mais popular por parte dos usuários de editores de texto e desenvolvedores.

6. REFERÊNCIAS BIBLIOGRÁFICAS

JARGAS, Aurélio Marinho. **Guia de Consulta Rápida Expressões Regulares.** Novatec Editora Ltda. 2001.

http://en.wikipedia.org/wiki/Regular_expression. Acessado em dezembro/2005.

http://regexlib.com. Acessado em outubro/2005.

http://www.postgresql.org/docs/8.1/static/functions-matching.html, Acessado em dezembro/2005.

José Antônio da Cunha
Gerência Educacional de Tecnologia da Informação – CEFET-RN
Av. Salgado Filho, 1559 Morro Branco CEP 59.000-000 Natal-RN
E-mail: jcunha@cefetrn.br

http://dev.mysql.com/doc/refman/4.1/pt/regexp.html. Acessado em outubro/2005.

http://javaalmanac.com/egs/java.util.regex. Acessado em Janeiro/2006.

LOTAR, Alfredo. **ASP.NET com C# Curso Prático**.Novatec Editora Ltda. 2003.

José Antônio da Cunha
Gerência Educacional de Tecnologia da Informação – CEFET-RN
Av. Salgado Filho, 1559 Morro Branco CEP 59.000-000 Natal-RN
E-mail: jcunha@cefetrn.br