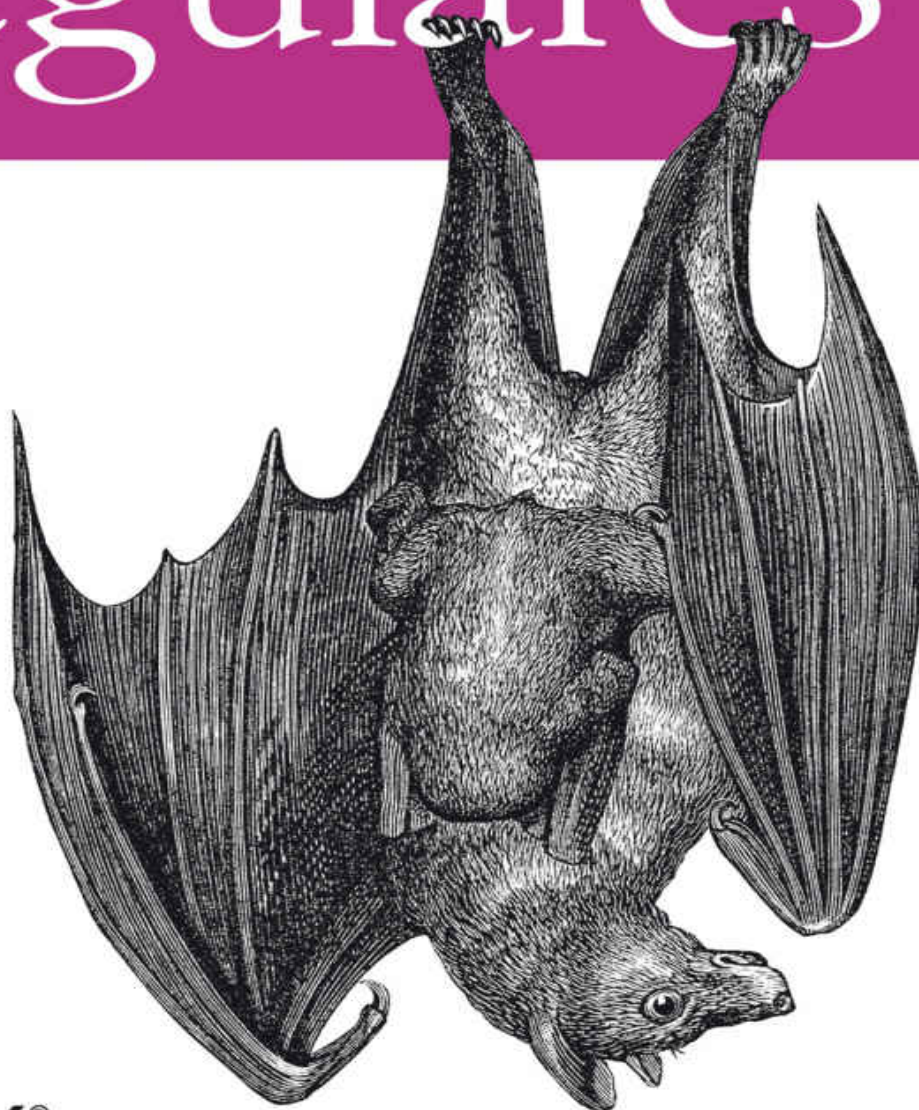


Desvendando as Expressões Regulares, Passo a Passo

Introdução às

Expressões Regulares



O'REILLY®
novatec

Michael Fitzgerald

Introdução às

Expressões Regulares

Michael Fitzgerald

O'REILLY®
Novatec

São Paulo | 2019

Authorized Portuguese translation of the English edition of titled Introducing Regular Expressions, First Edition ISBN 9781449392680 © 2012 Michael Fitzgerald. This translation is published and sold by permission of O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

Tradução em português autorizada da edição em inglês da obra *Introducing Regular Expressions*, First Edition ISBN 9781449392680 © 2012 Michael Fitzgerald. Esta tradução é publicada e vendida com a permissão da O'Reilly Media, Inc., detentora de todos os direitos para publicação e venda desta obra.

© Novatec Editora Ltda. [2012].

Todos os direitos reservados e protegidos pela Lei 9.610 de 19/02/1998. É proibida a reprodução desta obra, mesmo parcial, por qualquer processo, sem prévia autorização, por escrito, do autor e da Editora.

Editor: Rubens Prates

Tradução: Lúcia Ayako Kinoshita

Revisão gramatical: Marta Almeida de Sá

Editoração eletrônica: Carolina Kuwabata

ISBN: 978-85-7522-800-5

Histórico de edições impressas:

Setembro/2012 Primeira edição

Novatec Editora Ltda.

Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP – Brasil

Tel.: +55 11 2959-6529

E-mail: novatec@novatec.com.br

Site: www.novatec.com.br

Twitter: twitter.com/novateceditora

Facebook: facebook.com/novatec

LinkedIn: linkedin.com/in/novatec

Sumário

Sobre o autor

Prefácio

A quem se destina este livro

O que você precisa para usar este livro

Convenções usadas neste livro

Uso de exemplos de código de acordo com a política da O'Reilly

Como entrar em contato conosco

Agradecimentos

Capítulo 1 ■ O que é uma Expressão Regular?

Conhecendo o Regexpal

Correspondendo a um número de telefone no padrão norte-americano

Correspondendo a dígitos usando uma classe de caracteres

Usando shorthand de caracteres

Correspondendo a qualquer caractere

Grupos de captura e referências para trás

Usando quantificadores

Usando literais com escape

Uma amostra de aplicativos

O que você aprendeu no Capítulo 1

Notas técnicas

Capítulo 2 ■ Correspondência de padrões simples

Correspondendo a strings literais

Correspondendo a dígitos

Correspondendo a não-dígitos

Correspondendo a caracteres de palavra e não-caracteres de palavra

Correspondendo a espaços em branco

Correspondendo a qualquer caractere, mais uma vez

Inserindo marcação no texto

[Usando sed para marcação de texto](#)
[Usando Perl para marcação de texto](#)
[O que você aprendeu no Capítulo 2](#)
[Notas técnicas](#)

Capítulo 3 ■ Bordas

[Início e fim de uma linha](#)
[Bordas de palavra e não-bordas de palavra](#)
[Outras âncoras](#)
[Especificando um grupo de caracteres como literais](#)
[Adicionando tags](#)
[Adicionado tags com o sed](#)
[Adicionando tags usando Perl](#)
[O que você aprendeu no Capítulo 3](#)
[Notas técnicas](#)

Capítulo 4 ■ Alternância, grupos e retrovisores

[Alternância](#)
[Subpadrões](#)
[Capturando grupos e usando retrovisores](#)
[Grupos nomeados](#)
[Grupos de não-captura](#)
[Grupos atômicos](#)
[O que você aprendeu no Capítulo 4](#)
[Notas técnicas](#)

Capítulo 5 ■ Classes de caracteres

[Classes de caracteres negados](#)
[União e diferença](#)
[Classes de caracteres POSIX](#)
[O que você aprendeu no Capítulo 5](#)
[Notas técnicas](#)

Capítulo 6 ■ Correspondendo a caracteres Unicode e outros caracteres

[Correspondendo a um caractere Unicode](#)

[Usando vim](#)

[Correspondendo a caracteres com números octais](#)

[Correspondendo a propriedades de caracteres Unicode](#)

[Correspondendo a caracteres de controle](#)

[O que você aprendeu no Capítulo 6](#)

[Notas técnicas](#)

Capítulo 7 ■ Quantificadores

[Guloso, preguiçoso e possessivo](#)

[Correspondendo por meio de *, + e ?](#)

[Efetando a correspondência um número específico de vezes](#)

[Quantificadores preguiçosos](#)

[Quantificadores possessivos](#)

[O que você aprendeu no Capítulo 7](#)

[Notas técnicas](#)

Capítulo 8 ■ Lookarounds

[Lookaheads positivos](#)

[Lookaheads negativos](#)

[Lookbehinds positivos](#)

[Lookbehinds negativos](#)

[O que você aprendeu no Capítulo 8](#)

[Notas técnicas](#)

Capítulo 9 ■ Inserindo marcação HTML em um documento

[Correspondendo a tags](#)

[Transformando texto normal usando sed](#)

[Substituição usando sed](#)

[Lidando com Algarismos Romanos no sed](#)

[Lidando com um parágrafo específico usando sed](#)

[Lidando com as linhas do poema usando sed](#)

[Anexando tags](#)

[Usando um arquivo de comandos com o sed](#)

[Transformando texto normal usando Perl](#)

[Lidando com Algarismos Romanos usando Perl](#)

[Lidando com um parágrafo específico usando Perl](#)
[Lidando com as linhas do poema usando Perl](#)
[Usando um arquivo de comandos com Perl](#)
[O que você aprendeu no Capítulo 9](#)
[Notas técnicas](#)

Capítulo 10 ■ O fim do começo

[Aprendendo mais](#)
[Ferramentas, implementações e bibliotecas interessantes](#)
[Perl](#)
[PCRE](#)
[Ruby \(Oniguruma\)](#)
[Python](#)
[RE2](#)
[Correspondendo a um número de telefone no padrão norte-americano](#)
[Correspondendo a um endereço de email](#)
[O que você aprendeu no Capítulo 10](#)

Apêndice ■ Referência para Expressões Regulares

[Expressões regulares no QED](#)
[Metacaracteres](#)
[Shorthand de caracteres](#)
[Espaços em branco](#)
[Caracteres Unicode brancos](#)
[Caracteres de controle](#)
[Propriedades de caracteres](#)
[Nomes de script para propriedades de caracteres](#)
[Classes de Caracteres POSIX](#)
[Opções/modificadores](#)
[Tabela de códigos ASCII com Regex](#)
[Notas técnicas](#)

Glossário de expressões regulares

Sobre o autor

Michael Fitzgerald é programador e consultor e escreveu dez livros técnicos para a O'Reilly e John Wiley & Sons, bem como diversos artigos para a O'Reilly Network. Ele foi membro do comitê original que criou a linguagem de esquema RELAX NG para XML.

Prefácio

Este livro mostra como escrever expressões regulares por meio de exemplos. Seu objetivo é tornar o aprendizado de expressões regulares tão fácil quanto possível. De fato, o livro apresenta praticamente todos os conceitos por intermédio de exemplos, de modo que você pode facilmente reproduzi-los e experimentá-los por si mesmo.

As expressões regulares o ajudam a encontrar padrões em strings de texto. Mais precisamente, elas são strings de texto especialmente codificadas que correspondem a conjuntos de strings; na maioria das vezes, são strings que se encontram em documentos ou arquivos.

As expressões regulares começaram a ganhar evidência quando o matemático Stephen Kleene escreveu seu livro *Introduction to Metamathematics* (Nova York, Van Nostrand), publicado inicialmente em 1952, embora os conceitos já existissem desde o início da década de 1940. As expressões tornaram-se disponíveis aos cientistas da computação, de modo mais amplo, com o advento do sistema operacional Unix – trabalho de Brian Kernighan, Dennis Ritchie, Ken Thompson e outros da AT&T Bell Labs – e de seus utilitários, como o *sed* e o *grep*, no início da década de 1970.

A ocorrência mais antiga das expressões regulares em uma aplicação de computação que sou capaz de identificar encontra-se no editor QED. O QED, abreviação de Quick Editor, foi escrito para o *Berkeley Timesharing System*, que era executado no *Scientific Data Systems SDS 940*. Documentado em 1970, esse editor foi uma versão reescrita por Ken Thompson de outro editor anterior executado no *Compatible Time-Sharing System* do MIT; a versão resultou em uma das implementações práticas mais antigas, se não a primeira, das expressões regulares em computação. (A tabela A.1 no apêndice documenta os recursos de regex do QED.) Usarei diversas ferramentas para fazer a demonstração dos exemplos. Você achará a maioria deles útil e proveitosa, eu espero; alguns exemplos não

serão proveitosos porque não estarão prontamente disponíveis em seu sistema operacional Windows. Você pode pular aqueles que não forem práticos ou que não lhe parecerem atraentes. No entanto, eu recomendo a qualquer pessoa que esteja seriamente interessada na carreira de computação que aprenda a respeito das expressões regulares em um ambiente baseado em Unix. Eu trabalhei nesse tipo de ambiente durante 25 anos e continuo a aprender coisas novas todos os dias.

“Aqueles que não entendem o Unix estão condenados a reinventá-lo, precariamente.” – Henry Spencer

Algumas das ferramentas que mostrarei a você estão disponíveis online, acessíveis por meio de um navegador, e esta será a maneira mais fácil de usar para a maioria dos leitores. Outras serão usadas via prompt de comando ou de shell, e algumas serão executadas no desktop. As ferramentas poderão ser facilmente baixadas, caso você não as tenha. A maioria é gratuita ou não custa muito caro.

Este livro também não faz uso intensivo de jargões. Compartilharei os termos corretos com você quando necessário, mas em pequenas doses. Uso essa abordagem porque, ao longo dos anos, percebi que os jargões, com frequência, podem criar barreiras. Em outras palavras, procurarei não sobrecarregá-lo com a linguagem árida que descreve as expressões regulares. Faço isso por causa da filosofia básica deste livro: é possível fazer coisas produtivas antes de conhecer tudo a respeito de um determinado assunto.

Há uma série de diferentes implementações de expressões regulares. Você verá expressões regulares sendo usadas nas ferramentas de linha de comando do Unix, como *vi* (*vim*), *grep* e *sed*, dentre outras. Você encontrará as expressões regulares em linguagens de programação como Perl (é claro), Java, JavaScript, C# ou Ruby, e em muitas outras, e as encontrará em linguagens declarativas como o XSLT 2.0. Elas também se encontram presentes em aplicativos como o Notepad++, o Oxygen ou o TextMate, dentre muitos outros.

A maioria dessas implementações possui semelhanças e diferenças. Não abordarei todas essas diferenças neste livro, mas mencionarei uma boa quantidade delas. Se tentasse documentar

todas as diferenças que existem entre *todas* as implementações, eu enlouqueceria. Não me prenderei a esses tipos de detalhes neste livro. Você espera que o texto seja introdutório, conforme anunciado, e é isso que terá.

A quem se destina este livro

O público-alvo a quem este livro se destina se compõe de pessoas que nunca escreveram expressões regulares antes. Se você é novo no mundo das expressões regulares ou da programação, este livro é um bom lugar para começar. Em outras palavras, escrevo para o leitor que já ouviu falar de expressões regulares e está interessado nelas, mas que ainda não as compreende de verdade. Se você se enquadra nesse tipo de leitor, então este livro é bem adequado para você.

A ordem pela qual abordarei os recursos das regex será do simples em direção ao complexo. Em outras palavras, avançaremos passo a passo.

No entanto, se você já conhece alguma coisa a respeito de expressões regulares e como usá-las, ou se você é um programador experiente, pode ser que este livro não seja o ponto de partida desejado. Este é um livro introdutório para verdadeiros iniciantes que precisam ser conduzidos pela mão. Se você já escreveu algumas expressões regulares antes e se sente familiarizado com elas, pode começar por aqui, se quiser, mas eu pretendo ir mais devagar do que você provavelmente gostaria.

Eu recomendo diversos livros a serem lidos depois deste. Em primeiro lugar, experimente o livro *Dominando Expressões Regulares*, de Jeff Friedl, 3ª edição (ver <http://www.altabooks.com.br/dominando-expressoes-regulares.html>). O livro de Friedl oferece uma abordagem bem completa a respeito das expressões regulares, e eu o recomendo seriamente. Recomendo também o livro *Expressões Regulares Cookbook* (ver <http://novatec.com.br/livros/regexpcookbook/>) de Jan Goyvaerts e Steven Levithan. Jan Goyvaerts é o criador do RegexBuddy, uma poderosa aplicação para desktop (ver

<http://www.regexbuddy.com/>). Steven Levithan criou o Regexpal, um processador de expressões regulares online que será usado no primeiro capítulo deste livro (ver <http://www.regexpal.com>).

O que você precisa para usar este livro

Para obter o máximo deste livro, você precisará ter acesso a ferramentas que estão disponíveis nos sistemas operacionais Unix ou Linux, como o Darwin no Mac, uma variante do BSD (Berkeley Software Distribution) no Mac, ou o Cygwin em um microcomputador com Windows, que oferece muitas ferramentas GNU em sua distribuição (ver <http://www.cygwin.com> e <http://www.gnu.org>).

Haverá uma infinidade de exemplos para experimentar neste livro. Você pode simplesmente lê-los, se quiser, mas, para aprender de verdade, será preciso seguir o máximo de exemplos possíveis, pois eu acho que o tipo mais importante de aprendizado é sempre proveniente da prática; não basta ser um simples espectador. Você será apresentado a sites que ensinarão o que são as expressões regulares, colocando em destaque as correspondências efetuadas, as ferramentas de linha de comando do mundo Unix que fazem um trabalho pesado e as aplicações desktop que analisam expressões regulares ou as utilizam para efetuar buscas de textos.

Você encontrará exemplos deste livro no Github em <https://github.com/michaeljamesfitzgerald/Introducing-Regular-Expressions>.

Poderá também baixar um arquivo que contenha todos os arquivos de exemplos e de testes usados neste livro acessando <http://examples.oreilly.com/0636920012337/examples.zip>. Seria melhor criar um diretório ou uma pasta de trabalho em seu computador e depois baixar os arquivos para esse diretório antes de mergulhar de cabeça no livro.

Convenções usadas neste livro

As seguintes convenções tipográficas são usadas neste livro: *Itálico* Indica termos novos, URLs, endereços de email, nomes e extensões de arquivo e assim por diante.

Largura constante Usada para listagens de programas, bem como dentro dos parágrafos, para fazer referência a elementos de programas, tais como expressões e linhas de comando ou quaisquer outros elementos de programação.



Este ícone significa uma dica, sugestão ou observação geral.

Uso de exemplos de código de acordo com a política da O'Reilly

Este livro está aqui para ajudá-lo a fazer o seu trabalho. Em geral, você poderá usar o código-fonte que está neste livro em seus programas e em suas documentações. Não é necessário entrar em contato conosco para solicitar permissão, a menos que você vá reproduzir um trecho significativo do código. Por exemplo, escrever um programa que utilize diversos trechos de código deste livro não requer permissão. Vender ou distribuir um CD-ROM contendo exemplos dos livros da O'Reilly requer permissão. Responder a uma pergunta mencionando este livro ou citando um código de exemplo não requer permissão. Incorporar uma quantidade significativa de códigos de exemplo deste livro na documentação de seu produto requer permissão.

Apreciamos, mas não exigimos atribuição de créditos. Uma atribuição geralmente inclui o título, o autor, a editora e o ISBN. Por exemplo: “*Introducing Regular Expressions* by Michael Fitzgerald (O'Reilly). Copyright 2012 Michael Fitzgerald, 978-1-4493-9268-0.”

Se você achar que o seu uso de exemplos de códigos está além do uso permitido ou concedido acima, por favor, entre em contato com a O'Reilly pelo email permissions@oreilly.com.

Como entrar em contato conosco

Envie seus comentários e suas questões sobre este livro à editora escrevendo para: novatec@novatec.com.br.

Temos uma página web para este livro, na qual incluímos erratas, exemplos e quaisquer outras informações adicionais.

- Página da edição em português

http://www.novatec.com.br/catalogo/7522330_introexpreg • Página da
edição original em inglês
<http://shop.oreilly.com/product/0636920012337.do> Para obter mais
informações sobre os livros da Novatec, acesse nosso site em:
<http://www.novatec.com.br>.

Agradecimentos

Mais uma vez, gostaria de expressar minha admiração ao meu editor na O'Reilly, Simon St. Laurent, um homem muito paciente; não fosse por ele, este livro nunca teria visto a luz do dia. Agradeço a Seara Patterson Coburn e Roger Zauner por suas revisões proveitosas. E, como sempre, gostaria de expressar meu reconhecimento ao amor da minha vida, Cristi, que é minha *raison d'être*.

capítulo 1

O que é uma Expressão Regular?

Expressões regulares são strings de texto especialmente codificadas, utilizadas como padrões para corresponder a conjuntos de strings. Elas começaram a surgir na década de 1940 como uma maneira de descrever linguagens comuns, mas passaram realmente a ter destaque no mundo da programação na década de 1970. O primeiro lugar em que pude vê-las aparecendo foi no editor de textos QED, escrito por Ken Thompson.

“Uma expressão regular é um padrão que especifica um conjunto de strings de caracteres; diz-se que ela corresponde a determinadas strings.”

– Ken Thompson Posteriormente, as expressões regulares tornaram-se parte importante do conjunto de ferramentas que surgiram a partir do sistema operacional Unix – os editores *ed*, *sed* e *vi* (*vim*), o *grep*, o *AWK*, dentre outros. No entanto, as maneiras pelas quais as expressões regulares foram implementadas nem sempre foram tão regulares.



Este livro assume uma abordagem indutiva; em outras palavras, ele parte do específico em direção ao geral. Desse modo, em vez de encontrar um exemplo depois de um tratado, geralmente você terá um exemplo antes e um pequeno tratado a seguir. É um livro do tipo “aprenda na prática”.

As expressões regulares possuem fama de ser complicadas, mas tudo depende de como você fizer a abordagem. Há uma progressão natural de algo simples como: `\d`

um *shorthand* (abreviação) de caracteres que corresponde a qualquer dígito de 0 a 9, para algo um pouco mais complicado como: `^(\d{3}\)|^\d{3}[-.]?)?\d{3}[-.]?\d{4}$`

que é aonde chegaremos no final deste capítulo: uma expressão regular razoavelmente robusta que corresponde a um número de telefone de dez dígitos no padrão norte-americano, com ou sem parênteses ao redor do código de área, com ou sem hifens ou pontos separando os números. (Os parênteses devem ser pareados também; em outras palavras, não é possível ter somente um

parêntese.)



No capítulo 10, apresentaremos uma expressão regular um pouco mais sofisticada para um número de telefone, mas a expressão acima é suficiente para os propósitos deste capítulo.

Se você ainda não compreendeu como tudo isso funciona, não se preocupe: explicarei aos poucos a expressão completa, neste capítulo. Se você simplesmente seguir os exemplos (e também os demais ao longo do livro), escrever expressões regulares logo se tornará algo natural. Pronto para descobrir por si mesmo?

Às vezes, eu represento os caracteres Unicode neste livro usando seus *code points* – um número hexadecimal (base 16) de quatro dígitos. Esses *code points* são apresentados no formato *U+0000*. *U+002E*, por exemplo, representa o *code point* para ponto final (.).

Conhecendo o Regexpal

Inicialmente, deixe-me apresentá-lo ao site do Regexpal em <http://www.regexpal.com>. Acesse o site com um navegador, como por exemplo Google Chrome ou Mozilla Firefox. Você verá que o site tem a aparência apresentada na figura 1.1.

Note que há uma caixa de texto próxima ao topo e uma caixa de texto maior logo abaixo dessa. A caixa de texto superior serve para introduzir as expressões regulares e a de baixo conterá o texto de assunto ou texto-alvo. O texto-alvo é o texto ou o conjunto de strings no qual você quer efetuar as correspondências.

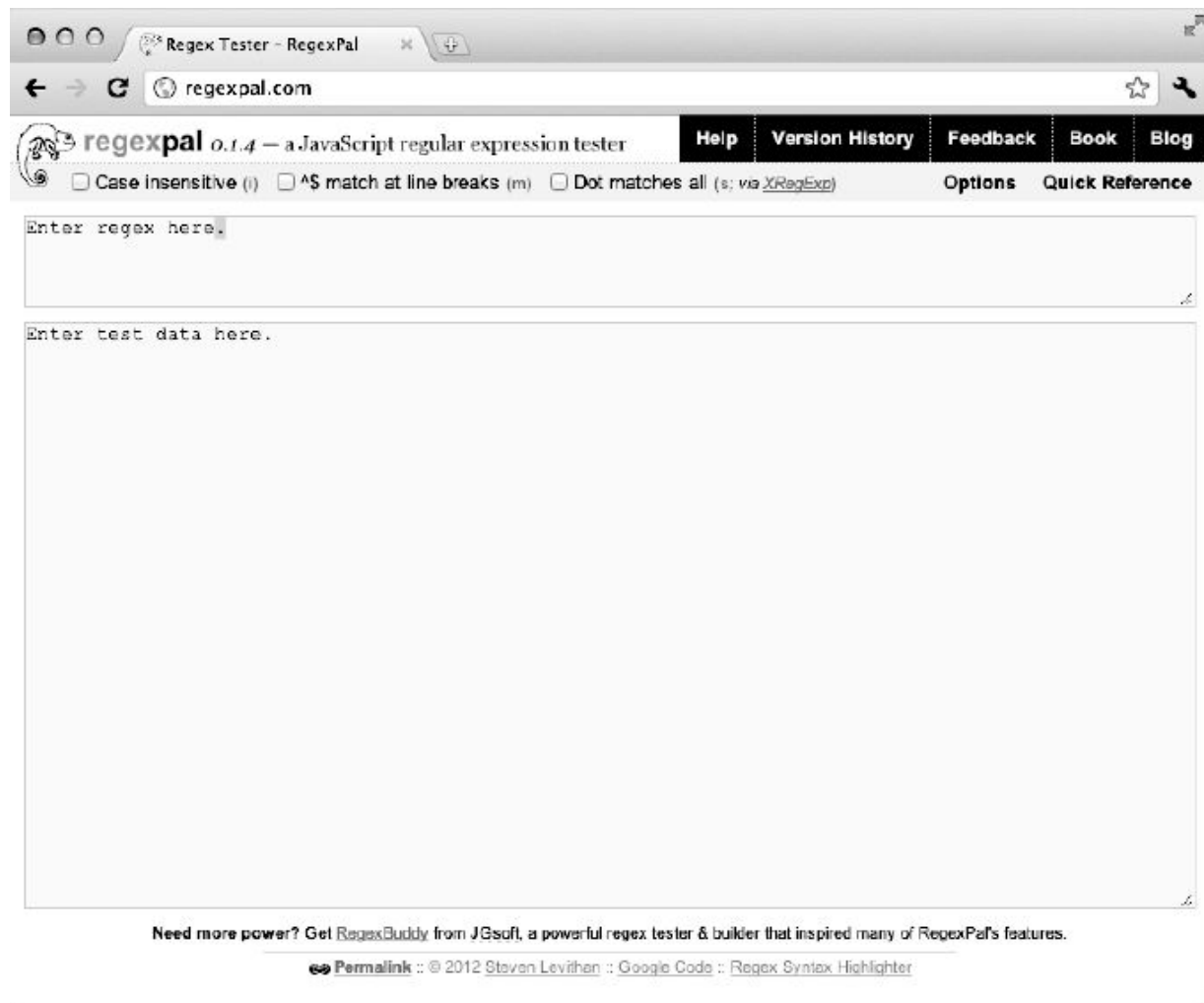


Figura 1.1 – Regexpal no navegador Google Chrome.



Ao final deste capítulo e de todos os capítulos seguintes, há uma seção intitulada “Notas técnicas”. Essas notas oferecem informações adicionais a respeito da tecnologia discutida no capítulo e indicam onde você poderá obter mais informações acerca dessa tecnologia. O fato de colocar as notas no final dos capítulos ajuda a manter a fluência do texto principal, evitando interrupções no caminho para discutir cada um dos detalhes.

Correspondendo a um número de telefone no padrão norte-americano

Agora faremos um número de telefone no padrão norte-americano corresponder a uma expressão regular. Digite o número de telefone que aparece aqui, na caixa de texto inferior do Regexpal: 707-827-

7019

Você o reconhece? É o número da O'Reilly Media.

Vamos fazer este número corresponder a uma expressão regular. Há várias maneiras de fazer isso, mas, para começar, digite simplesmente o número na caixa de texto superior, exatamente como está escrito na caixa de texto inferior (tenha paciência, não suspire): 707-827-7019

Você deverá ver o número de telefone digitado na caixa de texto inferior destacado em amarelo, do começo ao fim. Se for isso que estiver vendo (conforme mostrado na figura 1.2), então você está no caminho certo.

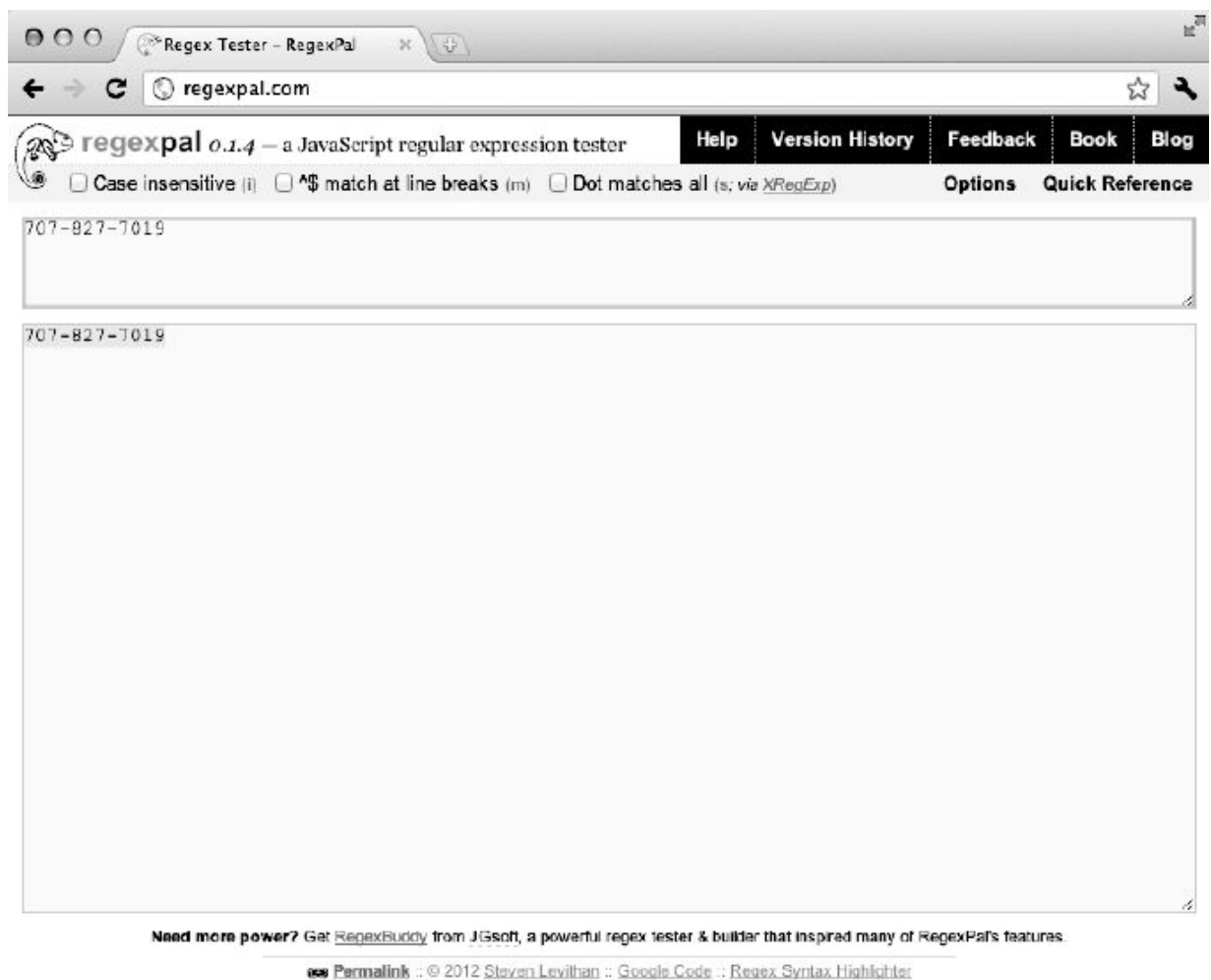


Figura 1.2 – Número de telefone com dez dígitos, em destaque no Regexpal.

Quando eu mencionar cores neste livro, em relação a algo que você



deverá ver em uma imagem ou tela, como por exemplo no texto em destaque no Regexpal, elas deverão aparecer online e nas versões eletrônicas deste livro, mas, infelizmente, não aparecerão na versão em papel. Portanto, se você estiver lendo este livro em papel, me desculpe, mas seu mundo terá gradações de cinza quando eu mencionar cores.

O que você fez nessa expressão regular foi usar algo chamado *string literal* para corresponder a uma string no texto-alvo. Uma string literal é uma representação literal de uma string.

Agora apague o número na caixa de texto superior e substitua-o somente pelo número 7. Viu o que aconteceu? Desta vez, somente os setes ficam destacados. O caractere literal (número) 7 na expressão regular coincide com as quatro ocorrências do número 7 no texto em que você está fazendo as correspondências.

Correspondendo a dígitos usando uma classe de caracteres

E se você quisesse corresponder a todos os algarismos do número de telefone de uma só vez? Ou se quisesse corresponder a qualquer número?

Experimente digitar o seguinte, exatamente como mostrado, novamente na caixa de texto superior: [0-9]

Todos os números (mais precisamente, os *dígitos*) da parte inferior ficarão destacados em amarelo e azul, alternadamente. O que a expressão regular [0-9] está dizendo ao processador de regex (abreviação de *regular expression*) é o seguinte: “corresponda a qualquer dígito que esteja no intervalo de 0 a 9”.

A associação literal com os colchetes não é feita porque eles são tratados de forma especial, como *metacaracteres*. Um metacaractere tem um significado especial nas expressões regulares e constitui um caractere reservado. Uma expressão regular no formato [0-9] é chamada de *classe de caracteres*, ou às vezes de *conjunto de caracteres*.

Você pode limitar o conjunto de dígitos de forma mais precisa e obter o mesmo resultado usando uma lista mais específica para

fazer a correspondência, como a que se segue: [012789]

Isso fará com que haja correspondência somente dos dígitos listados, ou seja, 0, 1, 2, 7, 8 e 9. Experimente digitá-los na caixa de texto superior. Novamente, todos os dígitos na caixa de texto inferior ficarão destacados com cores alternadas.

Para fazer a correspondência de qualquer número de telefone de dez dígitos no padrão norte-americano, cujas partes estejam separadas por hifens, você poderia fazer o seguinte: [0-9][0-9][0-9]-[0-9][0-9][0-9]-[0-9][0-9][0-9]

Funciona, mas é excessivamente extenso. Há uma solução melhor usando algo que se chama *shorthand*.

Usando shorthand de caracteres

Outra maneira de fazer correspondência de dígitos, que você já viu no início do capítulo, é usando o `\d`, o qual, sozinho, fará a correspondência de todos os dígitos arábicos, da mesma maneira que `[0-9]`. Experimente digitá-lo na caixa de texto superior e, assim como ocorreu com as expressões regulares anteriores, os dígitos abaixo ficarão em destaque. Esse tipo de expressão regular chama-se *shorthand (abreviação) de caracteres*. (Também é chamado de *escape de caracteres*, mas esse termo pode ser um pouco enganoso, por isso eu o evito. Explicarei o porquê mais tarde.) Para corresponder a qualquer dígito do número de telefone, você também poderia fazer isto: `\d\d\d-\d\d\d-\d\d\d\d` Repetir o `\d` três e quatro vezes na sequência resultará na correspondência exata de três e quatro dígitos na sequência. O hífen na expressão regular acima foi introduzido como caractere literal e a correspondência será feita dessa maneira.

E o que dizer desses hifens? Como fazer sua correspondência? Você pode usar um hífen literal (`-`), como já foi mostrado, ou poderia usar uma letra *D* maiúscula com escape (`\D`), que corresponde a qualquer caractere que *não* seja um dígito.

Este exemplo usa `\D` no lugar do hífen literal.

\d\d\d\D\d\d\d\D\d\d\d\d Mais uma vez, todo o número do telefone deverá ficar em destaque, desta vez incluindo os hifens.

Correspondendo a qualquer caractere

Você também poderia usar um ponto para corresponder a esses hifens chatinhos: \d\d\d.\d\d\d.\d\d\d\d O ponto atua essencialmente como um curinga e corresponde a qualquer caractere (exceto a um final de linha em determinadas situações). No exemplo acima, a expressão regular corresponde ao hífen, mas poderia também corresponder a um sinal de porcentagem (%): 707%827%7019

ou a uma barra vertical (|): 707|827|7019

ou a qualquer outro caractere.



Conforme mencionei antes, o caractere ponto normalmente não corresponderá a um caractere de mudança de linha, como o *line feed* (U+000A). No entanto, há maneiras possíveis de fazer corresponder uma mudança de linha a um ponto, que mostrarei posteriormente. Essa opção normalmente é chamada de *dotall*.

Grupos de captura e referências para trás

Você agora fará a correspondência somente de uma parte do número de telefone usando algo conhecido como *grupo de captura*. Então você fará referência ao conteúdo do grupo com um *retrovisor* (*backreference*). Para criar um grupo de captura, coloque um \d entre parênteses para inseri-lo em um grupo e depois coloque um \1 para fazer uma referência àquilo que foi anteriormente capturado: (\d)\d\1

O \1 faz uma referência ao que foi capturado antes pelo grupo entre parênteses. Como resultado, a expressão regular acima corresponde ao prefixo 707. Aqui está um detalhamento dela:

- (\d) corresponde ao primeiro dígito e o captura (o número 7);
- \d corresponde ao próximo dígito (o número 0), mas não o captura, porque não está entre parênteses;
- \1 referencia o dígito capturado (o número 7).

Essa expressão corresponderá somente ao código de área. Não se preocupe se você não estiver entendendo tudo neste momento.

Você verá muitos exemplos de grupos mais adiante neste livro.

Você poderia fazer a correspondência de todo o número de telefone usando um grupo e vários retrovisores: `(\d)0\1\D\d\d\1\D\1\d\d\d` Mas isso não está tão elegante quanto poderia ser. Vamos experimentar algo que funciona melhor ainda.

Usando quantificadores

Aqui está outra maneira de fazer a correspondência de um número de telefone usando uma sintaxe diferente: `\d{3}-?\d{3}-?\d{4}`

Os números entre chaves indicam ao processador de regex *exatamente* quantas ocorrências desses dígitos você quer que ele procure. As chaves com números são uma espécie de *quantificador*. As chaves propriamente ditas são consideradas metacaracteres.

O ponto de interrogação (?) é outro tipo de quantificador. Ele vem depois do hífen na expressão regular acima e indica que esse é opcional – ou seja, pode haver zero ou uma ocorrência do hífen (uma ou nenhuma). Há outros quantificadores, como o sinal de mais (+), que significa “um ou mais”, ou o asterisco (*), que significa “zero ou mais”.

Ao usar quantificadores, você pode deixar uma expressão regular mais concisa ainda: `(\d{3,4}[-.]?)+`

Novamente, o sinal de mais indica que a quantidade pode ocorrer uma ou mais vezes. Essa expressão regular corresponderá a três ou quatro dígitos, seguidos por um hífen ou um ponto opcional, agrupados por parênteses, uma ou mais vezes (+).

Sua cabeça está girando? Espero que não. Aqui está uma análise, caractere por caractere, da expressão regular anterior:

Caractere	Descrição
(Abre um grupo de captura
\	Início do <i>shorthand</i> de caracteres (escapa o caractere seguinte)

d	Fim do <i>shorthand</i> de caracteres (corresponda a qualquer dígito no intervalo de 0 a 9 com \d)
{	Abre o quantificador
3	Quantidade mínima para corresponder
,	Separa as quantidades
4	Quantidade máxima para corresponder
}	Fecha o quantificador
[Abre a classe de caracteres
.	Ponto (corresponde a um ponto literal)
-	Caractere literal para corresponder ao hífen
]	Fecha a classe de caracteres
?	Quantificador zero ou um
)	Fecha o grupo de captura
+	Quantificador um ou mais

Tudo isso funciona, mas não está totalmente correto porque a expressão corresponderá também a outros grupos de três ou quatro dígitos, estejam ou não no formato de um número de telefone. Sim, aprendemos com nossos erros, mais do que com nossos acertos.

Então vamos dar uma pequena melhoria: `(\d{3}[-.]?){2}\d{4}`

Esta expressão corresponde a duas sequências de três dígitos cada, que não estão entre parênteses, seguidas por um hífen opcional, e depois seguidas exatamente por quatro dígitos.

Usando literais com escape

Finalmente, aqui está uma expressão regular que permite que a primeira sequência de três dígitos esteja opcionalmente entre parênteses e faz com que o código de área também seja opcional:

`^(\\d{3})|^\\d{3}[-.]?\\d{3}[-.]?\\d{4}$`

Para garantir que a expressão é fácil de ser decifrada, vamos dar uma olhada nela, caractere por caractere, também:

Caractere	Descrição
<code>^</code>	(circunflexo) No início da expressão regular ou depois da barra vertical () significa que o número de telefone estará no início de uma linha
<code>(</code>	Abre um grupo de captura
<code>\\</code>	É um abre parênteses literal
<code>\\d</code>	Corresponde a um dígito
<code>{3}</code>	É um quantificador que, depois do <code>\\d</code> , corresponde exatamente a três dígitos
<code>)</code>	Corresponde a um fecha parênteses literal
<code> </code>	(barra vertical) Indica alternância, ou seja, um dado conjunto de alternativas. Em outras palavras, diz: “corresponda a um código de área com ou sem parênteses”

Caractere	Descrição
^	Corresponde ao início de uma linha
\d	Corresponde a um dígito
{3}	É um quantificador que corresponde exatamente a três dígitos
[.-]?	Corresponde a um ponto ou hífen opcional
)	Fecha o grupo de captura
?	Torna o grupo opcional, ou seja, o prefixo no grupo não é necessário
\d	Corresponde a um dígito
{3}	Corresponde exatamente a três dígitos
[.-]?	Corresponde a outro ponto ou hífen opcional
\d	Corresponde a um dígito
{4}	Corresponde exatamente a quatro dígitos
\$	Corresponde ao fim de linha

Essa última expressão regular corresponde a um número de telefone de dez dígitos no padrão norte-americano, com ou sem parênteses, hifens ou pontos. Experimente diferentes formatos do número para ver com quais deles haverá correspondência (e com quais não haverá).



O grupo de captura na expressão regular anterior não é necessário. O grupo é necessário, mas a parte referente à captura não é. Há uma maneira melhor de se fazer isso: um grupo de não-captura. Quando revisarmos essa expressão regular no último capítulo deste livro, você entenderá por que.

Uma amostra de aplicativos

Para concluir este capítulo, mostrarei a expressão regular para números de telefone em diversos aplicativos.

O TextMate é um editor disponível somente no Mac que utiliza a mesma biblioteca de expressões regulares usada na linguagem de programação Ruby. Você pode utilizar as expressões regulares por meio do comando *Find* (localizar), conforme mostrado na figura 1.3. Clique na caixa de seleção ao lado de *Regular expression*.

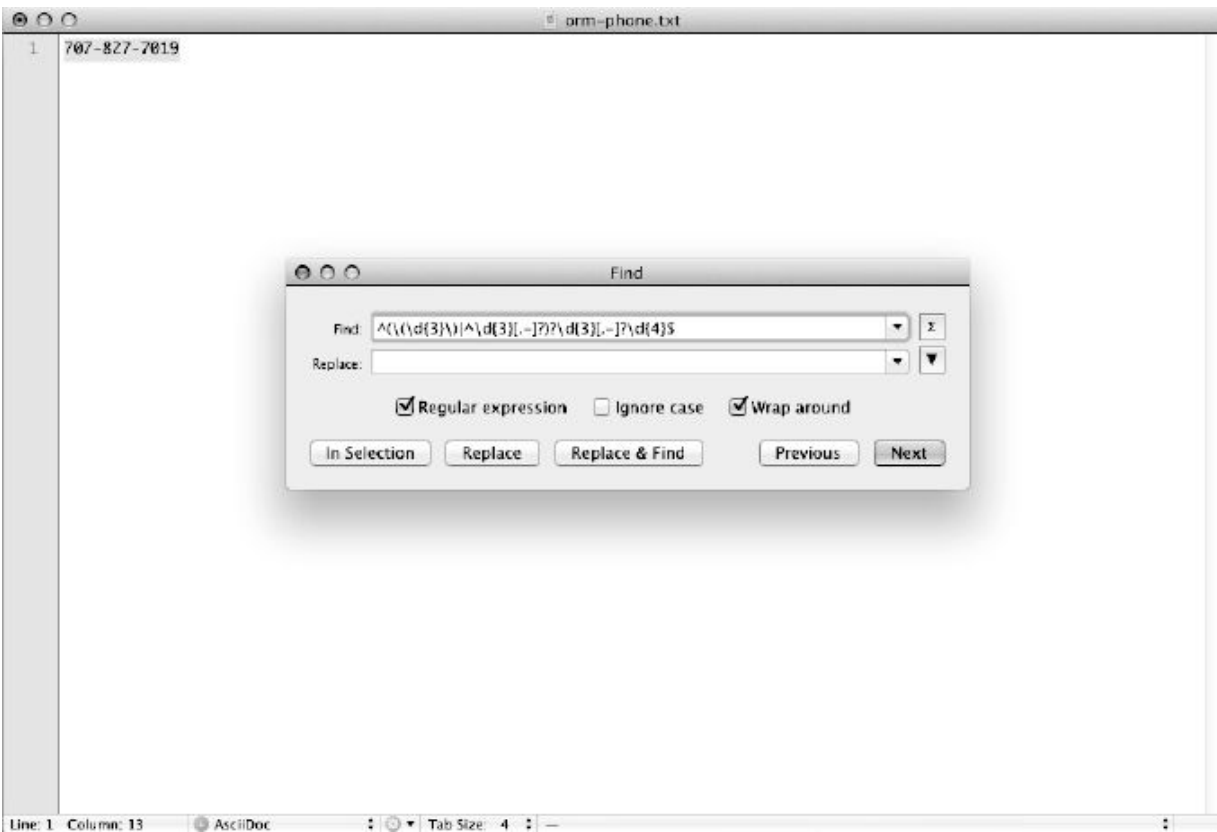


Figura 1.3 – Regex para número de telefone no TextMate.

O Notepad++ está disponível no Windows e é um editor popular, gratuito, que usa a biblioteca de expressões regulares PCRE. Você pode acessar as expressões regulares por meio das caixas de diálogo *Find* (localizar) e *Replace* (substituir) (Figura 1.4) e clicando no botão ao lado de *Regular expression*.

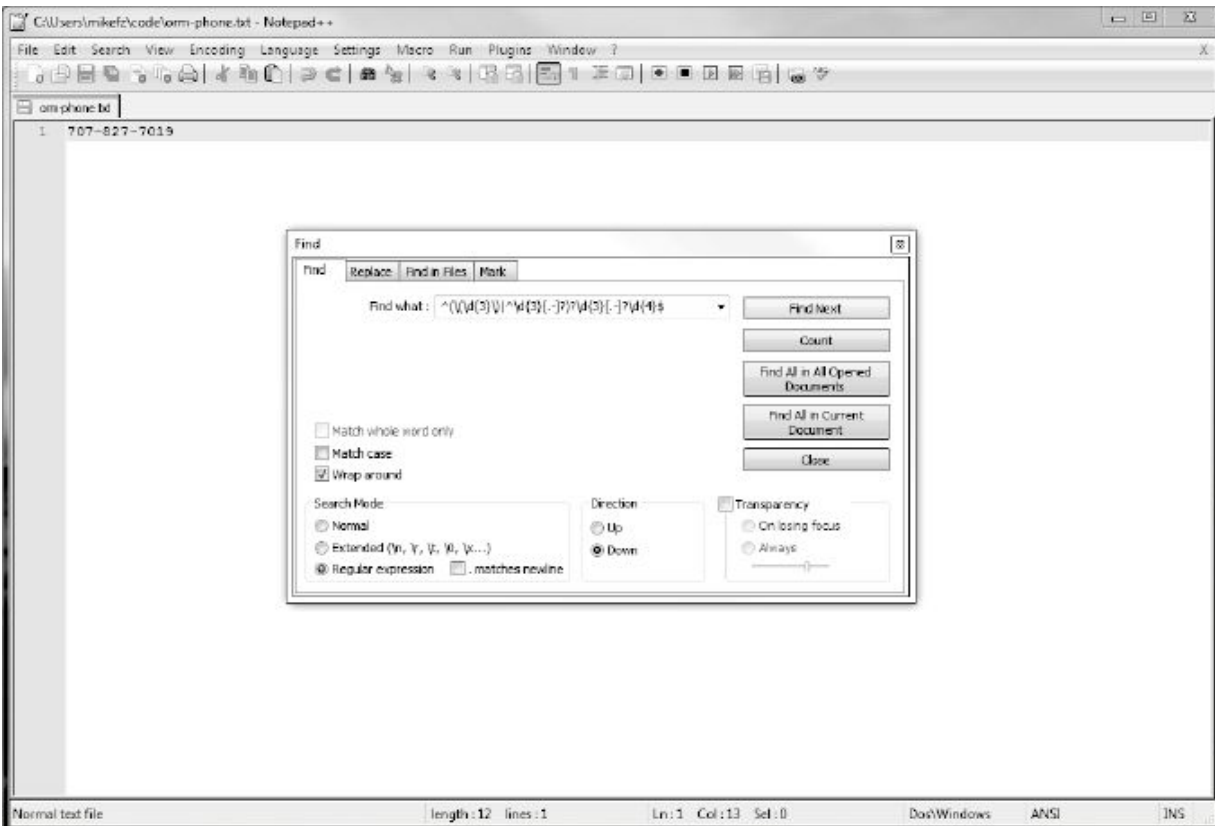


Figura 1.4 – Regex para número de telefone no Notepad++.

O Oxygen é um editor XML também popular e poderoso que usa a sintaxe das expressões regulares do Perl 5. Você pode ter acesso às expressões regulares por meio da caixa de diálogo *Find/Replace* (localizar/substituir), conforme mostrado na figura 1.5, ou utilizando o construtor de expressões regulares para XML Schema. Para usar expressões regulares com *Find/Replace*, clique na caixa de seleção ao lado de *Regular expression*.

E aqui termina a introdução. Parabéns. Você cobriu muitos assuntos neste capítulo. No próximo capítulo, focaremos na correspondência de padrões simples.

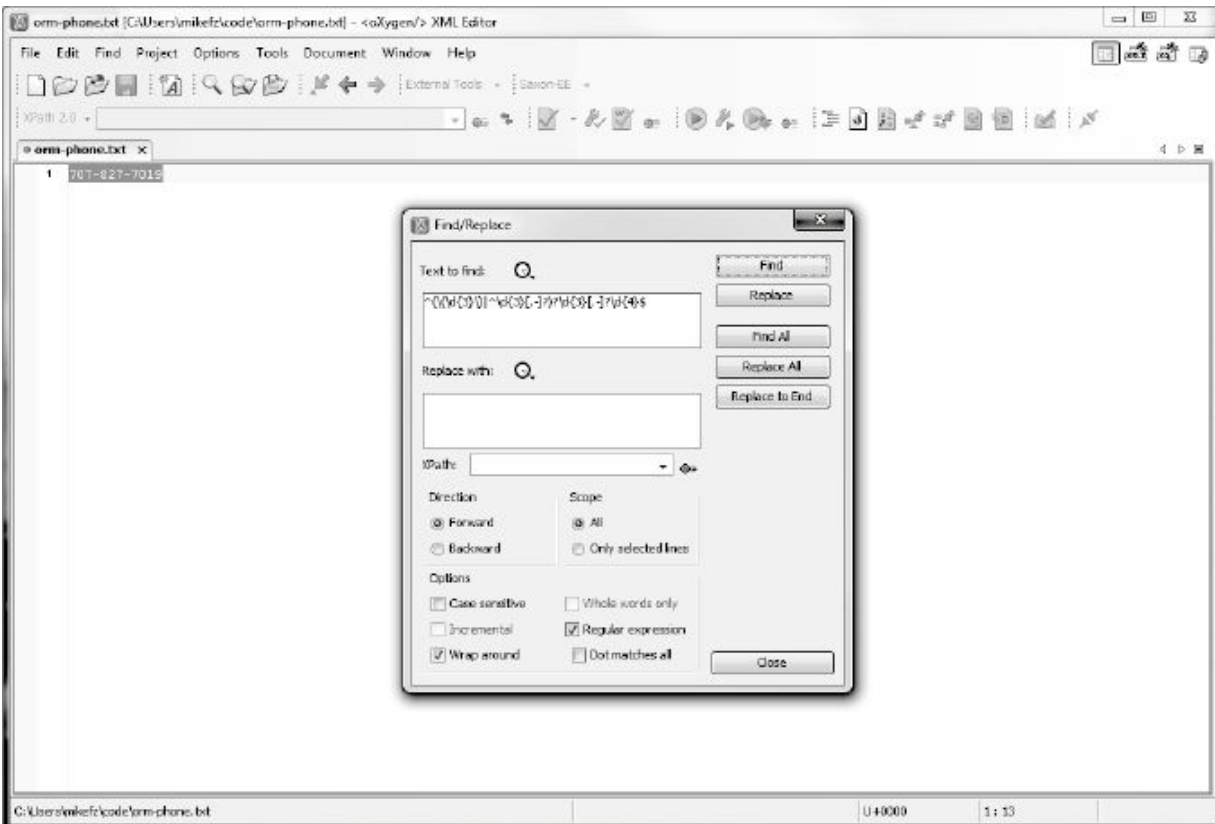


Figura 1.5 – Regex para número de telefone no Oxygen.

O que você aprendeu no capítulo 1

- O que é uma expressão regular.
- Como usar o Regexpal, um processador simples de expressões regulares.
- Como corresponder a strings literais.
- Como corresponder a dígitos usando uma classe de caracteres.
- Como corresponder a um dígito usando um *shorthand* de caracteres.
- Como corresponder a um não-dígito usando um *shorthand* de caracteres.
- Como usar um grupo de captura e um retrovisor (*backreference*).
- Como corresponder a uma quantidade exata de um conjunto de strings.
- Como corresponder a um caractere de forma opcional (zero ou

uma) ou uma ou mais vezes.

- Como corresponder a strings no início ou no final de uma linha.

Notas técnicas

- O Regexpal (<http://www.regexpal.com>) é uma implementação de regex baseada em web, que usa JavaScript. Não é a implementação mais completa e ele não faz tudo que as expressões regulares podem fazer; no entanto, é uma ferramenta de aprendizagem clara, simples e muito fácil de ser usada, fornecendo vários recursos para começar.
- Você pode baixar o navegador Chrome do site <https://www.google.com/chrome> ou o Firefox do site <http://www.mozilla.org/en-US/firefox/new/>.
- Por que há tantas maneiras de fazer coisas com expressões regulares? Um dos motivos é porque as expressões regulares possuem uma qualidade maravilhosa chamada *composicionalidade*. Uma linguagem, seja ela formal, de programação ou de esquema, que tenha a característica de composicionalidade (James Clark fornece uma boa explicação em <http://www.thaiopensource.com/relaxng/design.html#section:5>) é uma linguagem que permite que você pegue suas partes atômicas e seus métodos de composição e recombine-os facilmente de diversas maneiras. Depois de aprender as diferentes partes que compõem as expressões regulares, você aumentará sensivelmente sua capacidade de fazer correspondências de strings de qualquer tipo.
- O TextMate está disponível no site <http://www.macromates.com>. Para mais informações sobre expressões regulares no TextMate, consulte o site http://manual.macromates.com/en/regular_expressions.
- Para mais informações sobre o Notepad, consulte o site <http://notepad-plus-plus.org>. Para documentação sobre o uso de expressões regulares no Notepad, consulte o site http://sourceforge.net/apps/mediawiki/notepad-plus/index.php?title=Regular_Expressions.

- Descubra mais sobre o Oxygen no site <http://www.oxygenxml.com>. Para mais informações sobre o uso de regex com *Find/Replace*, consulte o site <http://www.oxygenxml.com/doc/ug-editor/topics/find-replace-dialog.html>. Para informações sobre o uso do construtor de expressões regulares para XML Schema, consulte o site <http://www.oxygenxml.com/doc/ug-editor/topics/XML-schema-regexp-builder.html>.

capítulo 2

Correspondência de padrões simples

Expressões regulares têm tudo a ver com corresponder e encontrar padrões em textos, variando de padrões simples a outros bem mais complexos. Este capítulo o conduzirá em um passeio pelos métodos mais simples para efetuar correspondência de padrões usando:

- Strings literais
- Dígitos

- Letras

- Caracteres de qualquer tipo No primeiro capítulo, utilizamos o RegexPal de Steven Levithan para fazer uma demonstração das expressões regulares. Neste capítulo, usaremos o site do RegExr de Grant Skinner, que se encontra no endereço <http://gskinner.com/regexr> (Figura 2.1).



Cada página deste livro o conduzirá cada vez mais para o interior da floresta de expressões regulares. Mas sinta-se à vontade para parar e aspirar o aroma da sintaxe. O que eu quero dizer é que você deve começar experimentando coisas novas assim que tomar conhecimento delas. Experimente. Falhe rapidamente. Recupere o controle. Continue. A melhor maneira de consolidar algo que foi aprendido é *fazendo* alguma coisa com ele.

Antes de continuar, gostaria de enfatizar a ajuda que o RegExr oferece. Na parte superior à direita do RegExr, você verá três guias. Observe aquelas relativas à *Samples* e *Community*. A guia *Samples* oferece ajuda para a sintaxe de diversas expressões regulares, e a guia *Community* mostra um grande número de expressões regulares resultantes de contribuições, as quais receberam classificações. Você encontrará um bocado de informações úteis nessas guias que poderão ajudá-lo. Além disso, ao passar o mouse sobre a expressão regular ou sobre o texto-alvo no RegExr, surgem popups que fornecem informações úteis. Esses recursos constituem um dos motivos pelos quais o RegExr está entre os meus verificadores online favoritos de regex.

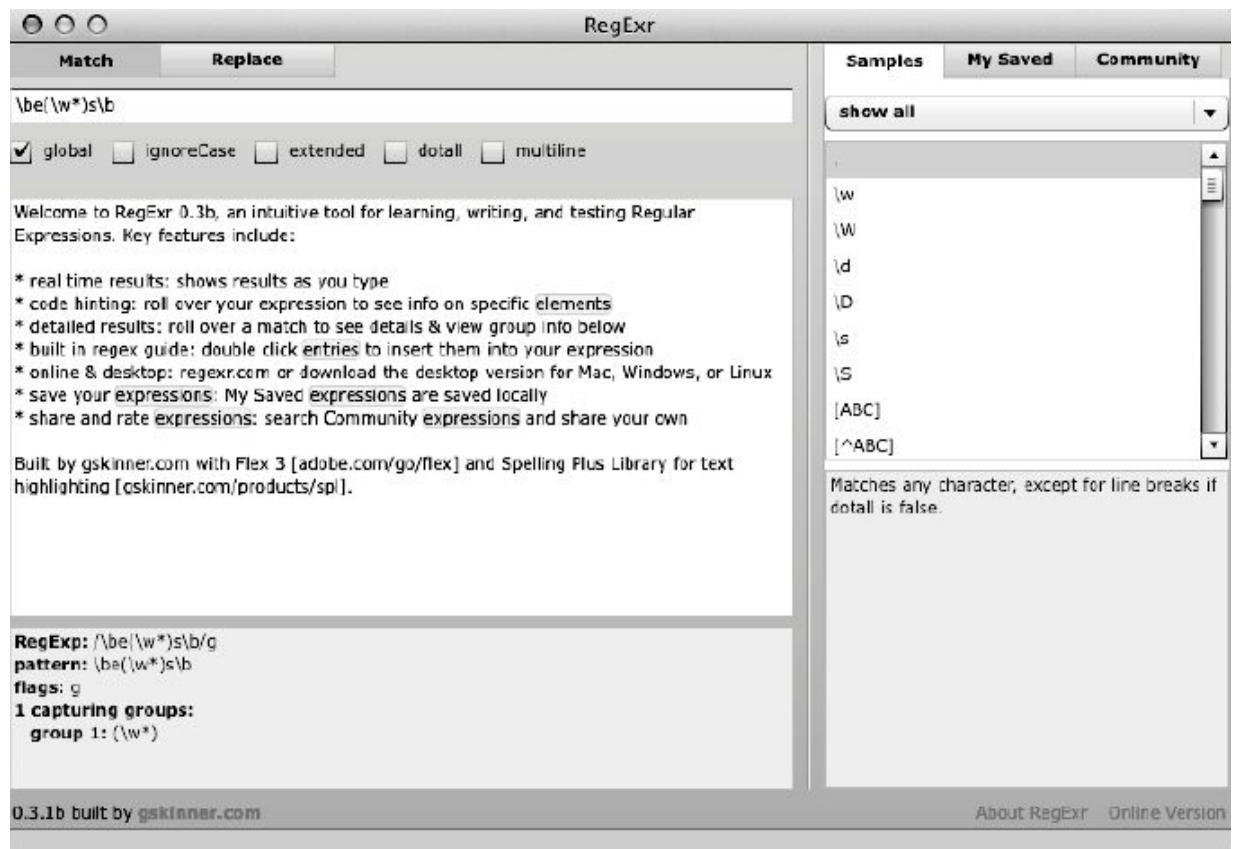


Figura 2.1. RegExr de Grant Skinner no Firefox.

Neste capítulo, apresentaremos nosso texto principal, “The Rime of the Ancyent Marinere”, de Samuel Taylor Coleridge, publicado inicialmente em *Lyrical Ballads* (London, J. & A. Arch, 1798). Trabalharemos com esse poema nos próximos capítulos, começando com uma versão simples do texto original e terminando com uma versão que contém marcações em HTML5. O texto completo do poema está armazenado em um arquivo chamado *rime.txt*; neste capítulo, usaremos o arquivo *rime-intro.txt*, que contém somente as primeiras linhas.

As linhas a seguir são do arquivo *rime-intro.txt*: THE ANCYENT MARINERE, IN SEVEN PARTS.

ARGUMENT.

How a Ship having passed the Line was driven by Storms to the cold Country towards the South Pole; and how from thence she made her course to the tropical Latitude of the Great Pacific Ocean; and of the strange things that befell; and in what manner the Ancyent Marinere came back to his own Country.

I.

1 It is an ancyent Marinere, 2 And he stoppeth one of three: 3 "By thy long grey beard and thy glittering eye 4 "Now wherefore stoppest me?

Copie e cole as linhas que aparecem aqui na caixa de texto inferior do RegExr. Você encontrará o arquivo *rime-intro.txt* no Github em <https://github.com/michaeljamesfitzgerald/Introducing-Regular-Expressions>. Você também encontrará o mesmo arquivo como parte do arquivo de download que se encontra em <http://examples.oreilly.com/0636920012337/examples.zip>. O texto online também pode ser encontrado no Project Gutenberg, mas sem as linhas numeradas (ver <http://www.gutenberg.org/ebooks/9622>).

Correspondendo a strings literais

O recurso mais óbvio e direto das expressões regulares consiste em corresponder a strings usando um ou mais caracteres literais, conhecidos como *strings literais* ou simplesmente *literais*.

O modo de corresponder a strings literais é usando caracteres normais, literais. Parece familiar, não é mesmo? Esse método é semelhante àquele utilizado quando fazemos uma pesquisa em um processador de textos ou quando submetemos uma palavra-chave a uma ferramenta de buscas. Quando você busca uma string de texto, caractere por caractere, você está fazendo uma pesquisa utilizando uma string literal.

Se quiser corresponder à palavra *Ship*, por exemplo, que é uma palavra (cadeia de caracteres) presente logo no início do poema, basta digitar a palavra *Ship* na caixa de texto que se encontra na parte superior do RegExr, e a palavra ficará em destaque na caixa de texto inferior. (Não se esqueça de digitar a palavra começando com letra maiúscula.) Apareceu um texto em destaque com a cor azul clara? Você deverá ver esse texto destacado na caixa de texto inferior. Se não o estiver vendo, verifique novamente o que foi digitado.



Por padrão, a correspondência de strings diferencia letras maiúsculas de minúsculas no RegExr. Se você não quiser corresponder a letras minúsculas e maiúsculas, clique na caixa de seleção ao lado de *ignoreCase* na parte superior à esquerda do RegExr. Se você selecionar

esta opção, haverá correspondência tanto da palavra *Ship* quanto de *ship* caso ambas estejam presentes no texto-alvo.

Correspondendo a dígitos

Na caixa de texto superior à esquerda do RegExr, digite este *shorthand* de caracteres para corresponder a dígitos: `\d` Essa expressão faz coincidir todos os dígitos arábicos na caixa de texto inferior porque a opção *global* está selecionada. Desmarque essa caixa de seleção e o `\d` corresponderá somente à primeira ocorrência do dígito. (ver figura 2.2.) Agora, em vez de `\d`, utilize uma classe de caracteres que fará a correspondência da mesma maneira. Digite o seguinte intervalo de dígitos na caixa de texto superior do RegExr: `[0-9]`

Como você pode observar na figura 2.3, embora a sintaxe seja diferente, usar `\d` produz o mesmo resultado que `[0-9]`.



Você aprenderá mais sobre classes de caracteres no capítulo 5.

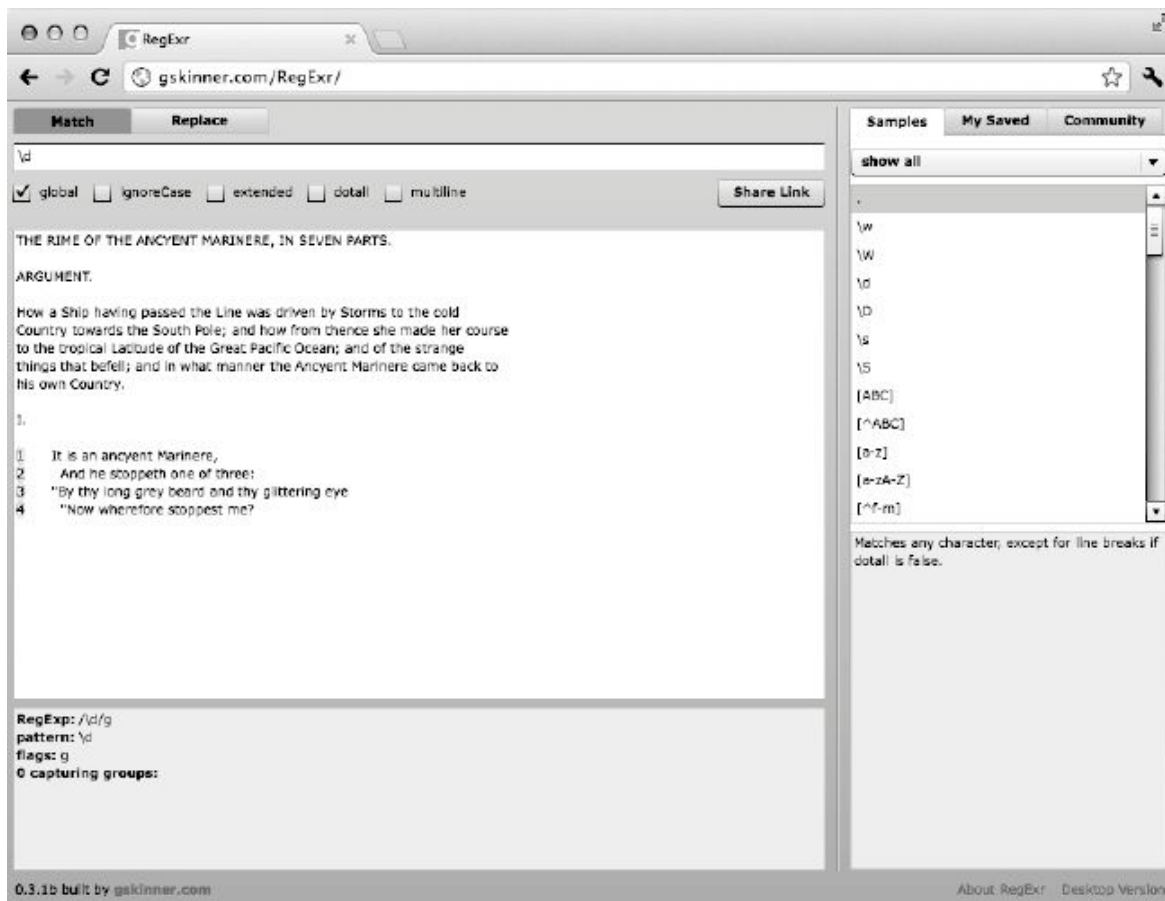


Figura 2.2 – Correspondendo a todos os dígitos no RegExr usando \d.

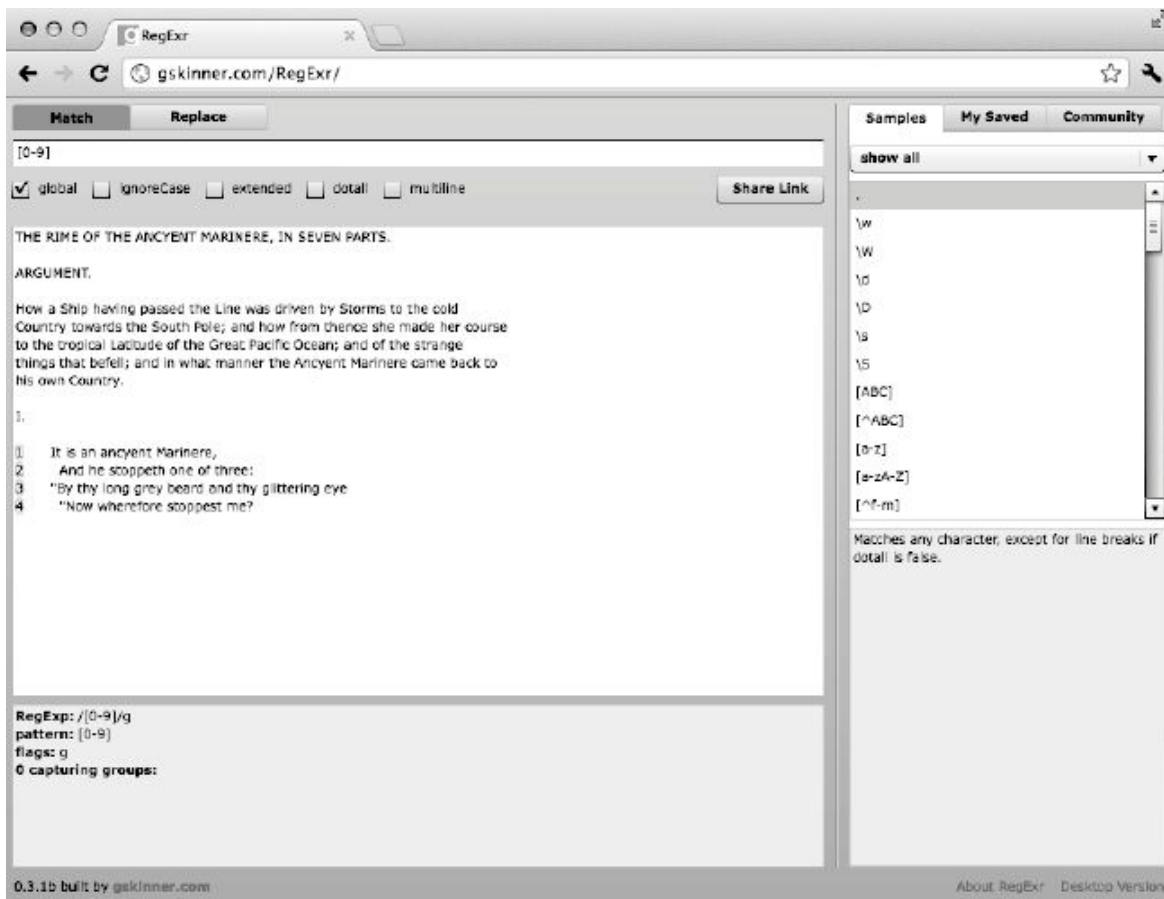


Figura 2.3 – Correspondendo a todos os dígitos no RegExr usando [0-9].

A classe de caracteres [0-9] é um *intervalo*, o que significa que ela corresponde ao intervalo de dígitos que vai de 0 a 9. Você também poderia fazer a correspondência dos dígitos de 0 a 9 listando todos os dígitos: [0123456789]

Se quiser corresponder somente aos dígitos binários 0 e 1, você pode usar a seguinte classe de caracteres: [01]

Experimente digitar [12] no RegExr e observe o resultado. Usando uma classe de caracteres, é possível selecionar os dígitos exatos em relação aos quais você deseja fazer a correspondência. O *shorthand* de caracteres para dígitos (\d) é mais conciso e mais simples, mas não tem nem o poder nem a flexibilidade de uma classe de caracteres. Eu uso classes de caracteres quando não é possível usar o \d (nem sempre é suportado) e quando preciso ser bastante específico a respeito dos dígitos em relação aos quais preciso fazer a correspondência; caso contrário, eu uso o \d porque

essa expressão tem uma sintaxe mais simples e mais conveniente.

Correspondendo a não-dígitos

Como acontece frequentemente com os *shorthands*, você pode fazer a inversão – ou seja, pode ir para a direção contrária. Por exemplo, se quiser corresponder a caracteres que não são dígitos, utilize este *shorthand* com uma letra *D* maiúscula: `\D`

Experimente usar esse *shorthand* no RegExr agora. Uma letra *D* maiúscula, em vez de minúscula, corresponde a caracteres que não são dígitos (ver figura 2.4). Esse *shorthand* é igual à classe de caracteres a seguir, que é uma classe negada (uma classe negada diz essencialmente “não corresponda a estes elementos” ou “corresponda a todos os elementos, exceto a estes”): `[^0-9]`

que por sua vez é igual a: `[^\d]`

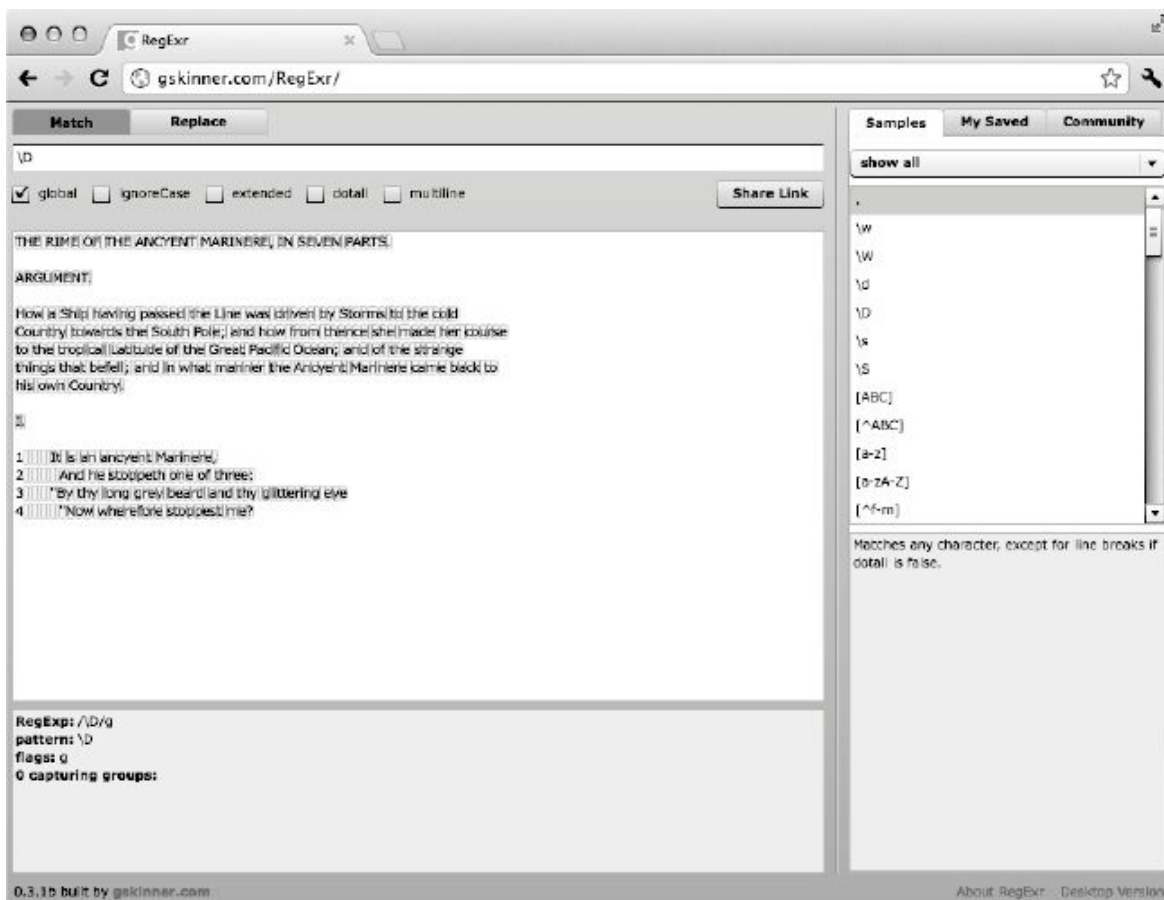


Figura 2.4 – Correspondendo a não-dígitos no RegExr usando `\D`.

Correspondendo a caracteres de palavra e não-caracteres de palavra

No RegExr, troque `\D` por: `\w` Este *shorthand* corresponderá a todos os caracteres de palavra (caso a opção *global* ainda estiver selecionada). A diferença entre `\D` e `\w` é que `\D` corresponde aos espaços em branco, aos sinais de pontuação, às aspas, aos hifens, às barras para frente, aos colchetes e a outros caracteres similares, enquanto o `\w` não – ele corresponde a letras e números.

Em inglês, o `\w` faz essencialmente a mesma correspondência efetuada pela classe de caracteres abaixo: `[a-zA-Z0-9]`



Você aprenderá como corresponder a caracteres que vão além do conjunto de letras usadas no inglês no capítulo 6.

Para corresponder a um não-caractere de palavra, utilize a letra maiúscula `W`: `\W`

Este *shorthand* corresponde a espaços em branco, sinais de pontuação e outros tipos de caracteres que não são utilizados em palavras neste exemplo. É o mesmo que usar a seguinte classe de caracteres: `[^a-zA-Z0-9]`

Devo admitir que as classes de caracteres permitem maior controle sobre a correspondência, mas às vezes você não quer ou não precisa digitar todos esses caracteres. Isso é conhecido como princípio do “vence o que exigir menos digitação”. Mas às vezes é preciso digitar todas essas coisas para especificar exatamente o que você deseja. A escolha é sua.

Somente para se divertir, experimente digitar no RegExr: `[\w]` e também `[\W]`

Deu para perceber as diferenças quanto às correspondências efetuadas?

A tabela 2.1 fornece uma lista estendida dos *shorthands* de caracteres. Nem todos eles funcionam em todos os processadores de regex.

Tabela 2.1 – Shorthands de caracteres

Shorthands de caracteres	Descrição
\a	Alerta
\b	Borda de palavra
[b]	Backspace
\B	Não-borda de palavra
\cx	Caractere Control
\d	Dígito
\D	Não-dígito
\dxxx	Valor de um caractere em decimal
\f	Alimentação de formulário
\r	Retorno de carro
\n	Mudança de linha
\oxxx	Valor de um caractere em octal
\s	Branco
\S	Não-branco
\t	Tabulação horizontal
\v	Tabulação vertical
\w	Caractere de palavra
\W	Não-caractere de palavra
\0	Caractere nulo
\xxx	Valor de um caractere em hexadecimal

Shorthands de caracteres	Descrição
\uxxxx	Valor de um caractere em Unicode

Correspondendo a espaços em branco

Para corresponder a espaços em branco, você pode usar este *shorthand*: \s Experimente usar essa expressão no RegExr e observe o que será realçado (ver figura 2.5). A classe de caracteres a seguir faz a mesma correspondência efetuada por \s: [\t\n\r]

Em outras palavras, essa classe corresponde a:

- Espaços em branco
- Tabulações (\t)
- Mudanças de linha (\n)
- Retornos de carro (\r)



Espaços em branco e tabulações aparecem em destaque no RegExr, mas as mudanças de linha e os retornos de carro não.

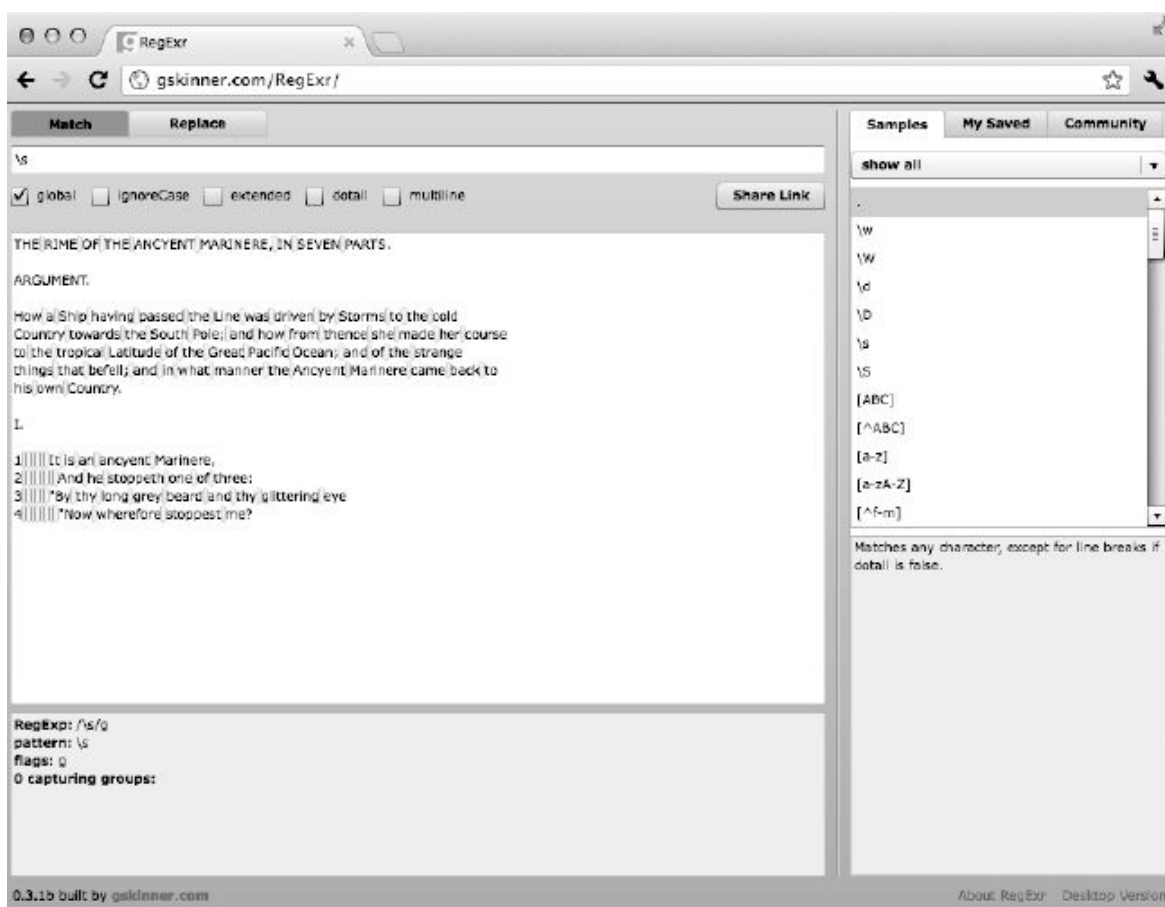


Figura 2.5 – Correspondendo a espaços em branco no RegExr usando \s.

Como você pode imaginar, o `\s` tem seu *compañero*. Para corresponder a um caractere não branco, utilize: `\S`

Esta expressão corresponde a todos os caracteres, exceto aos espaços em branco. Ela corresponde à classe de caracteres: `[\t\n\r]`

Ou:

`[^\s]`

Experimente usar as expressões acima no RegExr e observe o resultado.

Além dos caracteres cujas correspondências foram efetuadas pelo `\s`, há outros caracteres brancos menos comuns. A tabela 2.2 lista os *shorthands* de caracteres para caracteres brancos que são comuns e alguns que são mais raros.

Tabela 2.2 – Shorthands de caracteres para espaços em branco

Shorthand de caracteres	Descrição
<code>\f</code>	Alimentação de formulário
<code>\h</code>	Branco horizontal
<code>\H</code>	Não branco horizontal
<code>\n</code>	Mudança de linha
<code>\r</code>	Retorno de carro
<code>\t</code>	Tabulação horizontal
<code>\v</code>	Tabulação vertical (branco)
<code>\V</code>	Não-branco vertical

Correspondendo a qualquer caractere, mais uma vez

Há uma maneira de corresponder a *qualquer* caractere usando

expressões regulares, e essa maneira consiste em utilizar o ponto, também conhecido como *full stop* (U+002E). O ponto corresponde a todos os caracteres, exceto aos caracteres de fim de linha, a não ser em determinadas circunstâncias.

No RegExr, desmarque a caixa de seleção ao lado da opção *global*. A partir de agora, toda expressão regular efetuará a primeira correspondência identificada no texto-alvo.

Para corresponder a um único caractere, qualquer caractere, basta digitar um só ponto na caixa de texto superior do RegExr.

Na figura 2.6, você verá que o ponto corresponde ao primeiro caractere do texto-alvo, ou seja, à letra *T*.

Se quisesse corresponder à expressão *THE RIME* completa, você poderia usar oito pontos:

Mas esse método não é muito prático, de modo que eu não recomendo a utilização de uma série de pontos como essa frequentemente, se é que algum dia eu recomendaria. Em vez de oito pontos, utilize um quantificador: `.{8}`

e a expressão acima corresponderá às primeiras duas palavras e ao espaço entre elas, mas é basicamente isso. Para entender o que eu quero dizer com *basicamente*, clique na caixa de seleção ao lado de *global* e observe como esse método é inútil. A expressão corresponde a sequências de oito caracteres, do início ao fim, deixando de fora somente os últimos caracteres do texto-alvo.

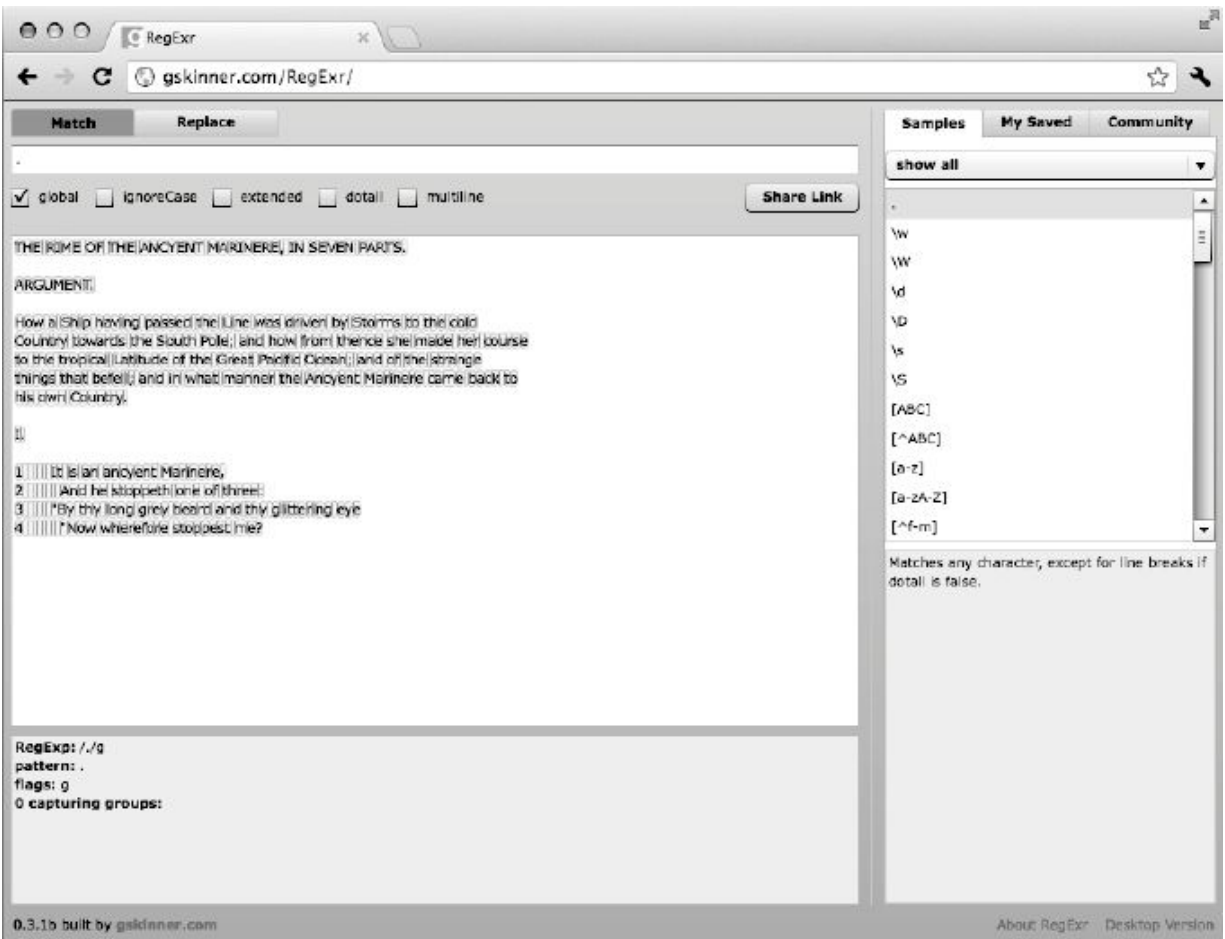


Figura 2.6 – Correspondendo a um único caractere no RegExr usando “.”

Vamos experimentar um método diferente usando bordas e letras de início e fim. Digite a seguinte expressão na caixa de texto superior do RegExr e note que haverá uma pequena diferença: `\bA.{5}T\b`. Essa expressão tem um pouco mais de especificidade. (Tente dizer *especificidade* três vezes em voz alta.) Ela corresponde à palavra *ANCYENT*, uma forma ortográfica arcaica para a palavra *ancient* (antigo, em inglês). Como?

- O *shorthand* `\b` corresponde a uma borda de palavra, sem consumir nenhum caractere.
- Os caracteres `A` e `T` também delimitam a sequência de caracteres.
- `{5}` corresponde a quaisquer cinco caracteres.
- `\b` corresponde à outra borda de palavra.

Essa expressão regular na verdade corresponderia tanto à *ANCYENT*

quanto à *ANCIENT*.

Agora experimente usar um *shorthand*: `\b\w{7}\b` Por último, falarei sobre corresponder a zero ou mais caracteres: `.*`

que é o mesmo que: `[\n]`

ou

`[\n\r]`

O ponto usado com o quantificador um ou mais (+) é similar à expressão acima: `.+`

Experimente usar essas expressões no RegExr e qualquer uma delas corresponderá à primeira linha (desabilite a caixa de seleção ao lado de *global*). Isso acontece porque normalmente o ponto não corresponde aos caracteres de mudança de linha, como um caractere de alimentação de formulário (U+000A) ou de retorno de carro (U+000D). Clique na caixa de seleção ao lado de *dotall* no RegExr, e então o `.*` ou o `.+` corresponderão a *todo* o texto na caixa de texto inferior. (*dotall* significa que um ponto corresponderá a todos os caracteres, inclusive aos de mudança de linha).

O motivo pelo qual isso acontece é que esses quantificadores são gulosos (*greedy*); em outras palavras, eles correspondem ao máximo de caracteres possíveis. Mas não se preocupe com isso ainda. No capítulo 7 explicaremos os quantificadores e a gulodice em maiores detalhes.

Inserindo marcação no texto

“The Rime of the Ancyent Marinere” é somente um texto comum. E se você quisesse mostrá-lo na Web? E se você quisesse marcá-lo como HTML5 usando expressões regulares, em vez de fazê-lo manualmente? Como você faria isso?

Em alguns dos capítulos seguintes, mostrarei algumas maneiras de fazer essa marcação. Começarei aos poucos, neste capítulo, e depois acrescentarei mais e mais marcações à medida que avançarmos.

No RegExr, clique na guia Replace, selecione *multiline* (múltiplas

linhas) e, na primeira caixa de texto, digite: (^T.*\$) Começando no início do arquivo, essa expressão efetuará a correspondência da primeira linha do poema e em seguida fará a captura desse texto em um grupo usando parênteses. Na caixa de texto seguinte, digite: <h1>\$1</h1> A regex de substituição insere um elemento *h1* ao redor do grupo capturado, representado por \$1. Você pode ver o resultado na caixa de texto mais abaixo. O \$1 é um retrovisor, no estilo do Perl. Na maioria das implementações, incluindo o Perl, você utilizará este estilo: \1; mas o RegExr suporta somente \$1, \$2, \$3 e assim por diante. Você aprenderá mais acerca de grupos e retrovisores no capítulo 4.

Usando sed para marcação de texto

Em uma linha de comando, você também poderia fazer a marcação usando *sed*. O *sed* é um editor de stream do Unix que aceita expressões regulares e que permite fazer transformações em textos. Esse editor foi inicialmente desenvolvido no começo da década de 1970 por Lee McMahon no Bell Labs. Se você está trabalhando no Mac ou tem um pacote Linux, então você já tem o editor.

Experimente usar o *sed* em um prompt de shell (como, por exemplo, em uma janela de Terminal no Mac) usando esta linha: `echo Hello | sed s/Hello/Goodbye/`

É isto o que deverá acontecer:

- O comando `echo` mostra a palavra *Hello* na saída padrão (que geralmente é a sua tela), mas a barra vertical (|) faz o pipe dessa saída para o comando *sed* que vem na sequência.

- O pipe direciona a saída do `echo` para a entrada do *sed*.
- O comando `s` (substituição) do *sed* muda a palavra *Hello* para *Goodbye*, e a palavra *Goodbye* é apresentada na tela.

Se você ainda não possui o *sed* em sua plataforma, ao final deste capítulo, você encontrará algumas notas técnicas contendo orientações acerca de informações para instalação. Você verá ali discussões a respeito de duas versões do *sed*: BSD e GNU.

Agora experimente fazer isto: em um prompt de shell ou de

comando, digite: `sed -n 's/^/<h1>/;s$/<\h1>/p;q' rime.txt` E a saída será: `<h1>THE RIME OF THE ANCYENT MARINERE, IN SEVEN PARTS.</h1>` Eis aqui o que a regex fez, em detalhes:

- A linha começa invocando o programa *sed*.

- A opção `-n` anula o comportamento padrão do *sed*, que consiste em ecoar cada linha da entrada na saída. Isso porque você quer ver somente a linha afetada pela regex, ou seja, a linha 1.
- `s/^/<h1>/` coloca uma tag *h1* inicial no começo (^) da linha.
- O ponto e vírgula (;) separa os comandos.
- `s$/<\h1>/` coloca uma tag *h1* final no fim (\$) da linha.
- O comando `p` mostra a linha afetada (linha 1). Este comando contrasta com o `-n`, que ecoa todas as linhas, independentemente de ter sido afetada.
- Por último, o comando `q` faz sair do programa, de modo que o *sed* processa somente a primeira linha.
- Todas as operações anteriores são executadas em relação ao arquivo *rime.txt*.

Outra maneira de escrever esta linha é usando a opção `-e`. A opção `-e` concatena os comandos de edição, um após o outro. Eu prefiro o método com ponto e vírgula, é claro, porque é mais conciso.

```
sed -ne 's/^/<h1>/' -e 's$/<\h1>/p' -e 'q' rime.txt
```

Você poderia também reunir esses comandos em um arquivo, como foi feito, por exemplo, no arquivo *h1.sed* mostrado aqui (esse arquivo está no repositório de códigos mencionado anteriormente):

```
#!/usr/bin/sed
s/^/<h1>/
s$/<\h1>/
q
```

Para executá-lo, digite `sed -f h1.sed rime.txt` em um prompt no mesmo diretório ou na mesma pasta em que se encontra o arquivo *rime.txt*.

Usando Perl para marcação de texto

Por último, mostrarei como executar um processo semelhante usando Perl. Perl é uma linguagem de programação de uso geral, criada por Larry Wall nos idos de 1987. É uma linguagem conhecida por suportar fortemente as expressões regulares e por sua

capacidade de processamento de textos.

Descubra se você já tem Perl instalado em seu sistema digitando a seguinte linha em um prompt de comando e teclando Return ou Enter: `perl -v` Esse comando deve retornar a versão de Perl que está em seu sistema ou uma mensagem de erro (consulte a seção de “Notas técnicas”).

Para obter a mesma saída mostrada no exemplo usando *sed*, digite esta linha em um prompt: `perl -ne 'if ($. == 1) { s/^/<h1>/; s/$/<\h1>/m; print; }' rime.txt` e, como ocorreu no exemplo com *sed*, você obterá este resultado: `<h1>THE RIME OF THE ANCYENT MARINERE, IN SEVEN PARTS.</h1>` Eis o que aconteceu no comando Perl, novamente em detalhes:

- `perl` chama o programa Perl.

- A opção `-n` percorre a entrada (o arquivo *rime.txt*).
- A opção `-e` permite submeter código de programação na linha de comando, em vez de usar um arquivo (como no *sed*).
- O comando `if` verifica se você está na linha 1. O `$.` é uma variável especial do Perl que corresponde à linha atual.
- O primeiro comando de substituição `s` encontra o começo da primeira linha (`^`) e insere uma tag *h1* inicial nesse local.
- O segundo comando de substituição procura pelo final da linha (`$`) e insere uma tag *h1* final.
- O modificador ou flag `m` ou *multiline* no final do comando de substituição indica que você está tratando essa linha de forma distinta e separada; conseqüentemente, o `$` corresponde ao final da linha 1, e não ao final do arquivo.
- Por último, o comando mostra o resultado na saída padrão (a tela).
- Todas as operações acima são executadas em relação ao arquivo *rime.txt*.

Você também poderia armazenar todos esses comandos em um arquivo de programa, como foi feito no arquivo *h1.pl*, que se encontra no arquivo contendo os exemplos.

```
#!/usrbin/perl -n if ($. == 1) {  
s/^\<h1>/; s/$/<\h1>/m; print; }
```

E, no mesmo diretório em que se encontra o arquivo *rime.txt*, execute o programa desta maneira: `perl h1.pl rime.txt` Há várias maneiras de fazer coisas com Perl. Não estou dizendo que essa é a maneira mais eficiente de adicionar essas tags. É somente uma das maneiras. É bem provável que, quando este livro estiver publicado, eu encontre outras maneiras mais eficientes de fazer coisas com Perl (e com outras ferramentas). Espero que você também encontre. No capítulo seguinte, falaremos sobre bordas e sobre aquilo que é conhecido como *asserções de largura zero*.

O que você aprendeu no capítulo 2

- Como corresponder a strings literais.
- Como corresponder a dígitos e a não-dígitos.
- O que é o modo *global*.
- Como os *shorthands* de caracteres se comparam às classes de caracteres.
- Como corresponder a caracteres de palavra e a não-caracteres de palavra.
- Como corresponder a espaços em branco.
- Como corresponder a qualquer caractere usando ponto.
- O que é o modo *dotall*.
- Como inserir marcações HTML em uma linha de texto usando RegExr, *sed* e Perl.

Notas técnicas

- O RegExr se encontra no site <http://www.gskinner.com/RegExr> e possui também uma versão desktop (<http://www.gskinner.com/RegExr/desktop/>). O RegExr foi feito em Flex 3 (<http://www.adobe.com/products/flex.html>) e baseia-se na implementação de expressões regulares do ActionScript (<http://www.adobe.com/devnet/actionsript.html>). Suas expressões

regulares são semelhantes àsquelas usadas pelo JavaScript (ver https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/RegExp).

- O Git é um sistema rápido de controle de versões (<http://git-scm.com>). O GitHub é um repositório de projetos baseado em web que usa o Git (<http://github.com>). Eu sugiro o uso do repositório do GitHub para acessar os exemplos deste livro somente se você se sentir à vontade com o Git ou com outros sistemas modernos de controle de versão, como o Subversion ou o Mercurial.
- O HTML5 (<http://www.w3.org/TR/html5/>) é a quinta grande revisão do HTML do W3C, a linguagem de marcação para publicação na World Wide Web. Ela permaneceu em draft por diversos anos e muda regularmente, mas é amplamente aceita como a aparente herdeira do HTML 4.01 e do XHTML.
- O `sed` está prontamente disponível nos sistemas Unix/Linux, incluindo o Mac (versão Darwin ou BSD). Está também disponível no Windows em pacotes como o Cygwin (<http://www.cygwin.com>) ou individualmente em <http://gnuwin32.sourceforge.net/packages/sed.htm> (atualmente, na versão 4.2.1; ver <http://www.gnu.org/software/sed/manual/sed.html>).
- Para usar os exemplos em Perl presentes neste capítulo, pode ser que seja necessário instalar Perl em seu sistema. Ele vem por padrão com o Mac OS X Lion e em geral está presente nos sistemas Linux. Se estiver usando Windows, você poderá adquirir o Perl instalando os pacotes Cygwin adequados (ver <http://www.cygwin.com>) ou efetuando o download do pacote mais recente a partir do site da ActiveState (acesse o endereço <http://www.activestate.com/activeperl/downloads>).

Para informações detalhadas a respeito da instalação do Perl, acesse o site <http://learn.perl.org/installing/> ou <http://www.perl.org/get.html>.

Para descobrir se você já tem Perl instalado, digite o comando abaixo em um prompt de shell. Para isso, abra uma janela de comandos ou de shell em seu sistema, como, por exemplo, uma janela de Terminal (em Aplicativos/Utilitários) no Mac ou uma

janela de linha de comandos no Windows (clique em Iniciar e depois digite `cmd` na caixa de texto na parte inferior do menu).

Digite a linha abaixo no prompt: `perl -v` Se o Perl estiver ativo e funcionando em seu sistema, então esse comando retornará informações sobre a versão do Perl. Em meu Mac executando Lion, eu instalei a versão mais recente de Perl (5.16.0 por ocasião desta publicação) a partir do código-fonte e o compilei (ver <http://www.cpan.org/src/5.0/perl-5.16.0.tar.gz>). Quando digito o comando acima, eu recebo a seguinte informação: This is perl 5, version 16, subversion 0 (v5.16.0) built for darwin-2level Copyright 1987-2012, Larry Wall Perl may be copied only under the terms of either the Artistic License or the GNU General Public License, which may be found in the Perl 5 source kit.

Complete documentation for Perl, including FAQ lists, should be found on this system using "`man perl`" or "`perldoc perl`". If you have access to the Internet, point your browser at <http://www.perl.org/>, the Perl Home Page.

Tanto o `perl` quanto o `perldoc` são instalados em `usr/local/bin` quando compilados e gerados a partir do código-fonte, e você pode adicioná-lo em seu path. Para informações sobre configuração de sua variável de path, consulte o site <http://java.com/en/download/help/path.xml>.

capítulo 3

Bordas

Este capítulo tem como foco as asserções. As asserções marcam fronteiras ou bordas, mas não consomem caracteres, ou seja, não fazem com que caracteres sejam retornados como resultado. Elas também são conhecidas como *asserções de largura zero*. Uma asserção de largura zero não corresponde a nenhum caractere propriamente dito, mas a uma localização em uma string. Algumas delas, como `^` e `$`, também são conhecidas como âncoras.

As bordas sobre as quais falarei neste capítulo correspondem a: • Início e fim de uma linha ou string.

- Bordas de palavra (dois tipos).
- Início e fim de um assunto.
- Bordas que delimitam e especificam strings literais.

Para começar, usarei o RegExr novamente, porém, desta vez, para variar, usarei o navegador Safari (mas você pode usar qualquer navegador que preferir). Usarei também o mesmo texto utilizado da última vez: as doze primeiras linhas do arquivo *rime.txt*. Abra o navegador Safari, acesse o site <http://gskinner.com/regexr> e copie as primeiras doze linhas do arquivo *rime.txt*, contido no arquivo de códigos, para a caixa de texto inferior.

Início e fim de uma linha

Como você já viu diversas vezes, para corresponder ao início de uma linha ou de uma string, utilize o circunflexo (U+005E): `^`

Dependendo do contexto, um `^` corresponderá ao início de uma linha ou de uma string, às vezes, de todo um documento. O contexto depende de seu aplicativo e de quais opções você estiver usando nesse aplicativo.

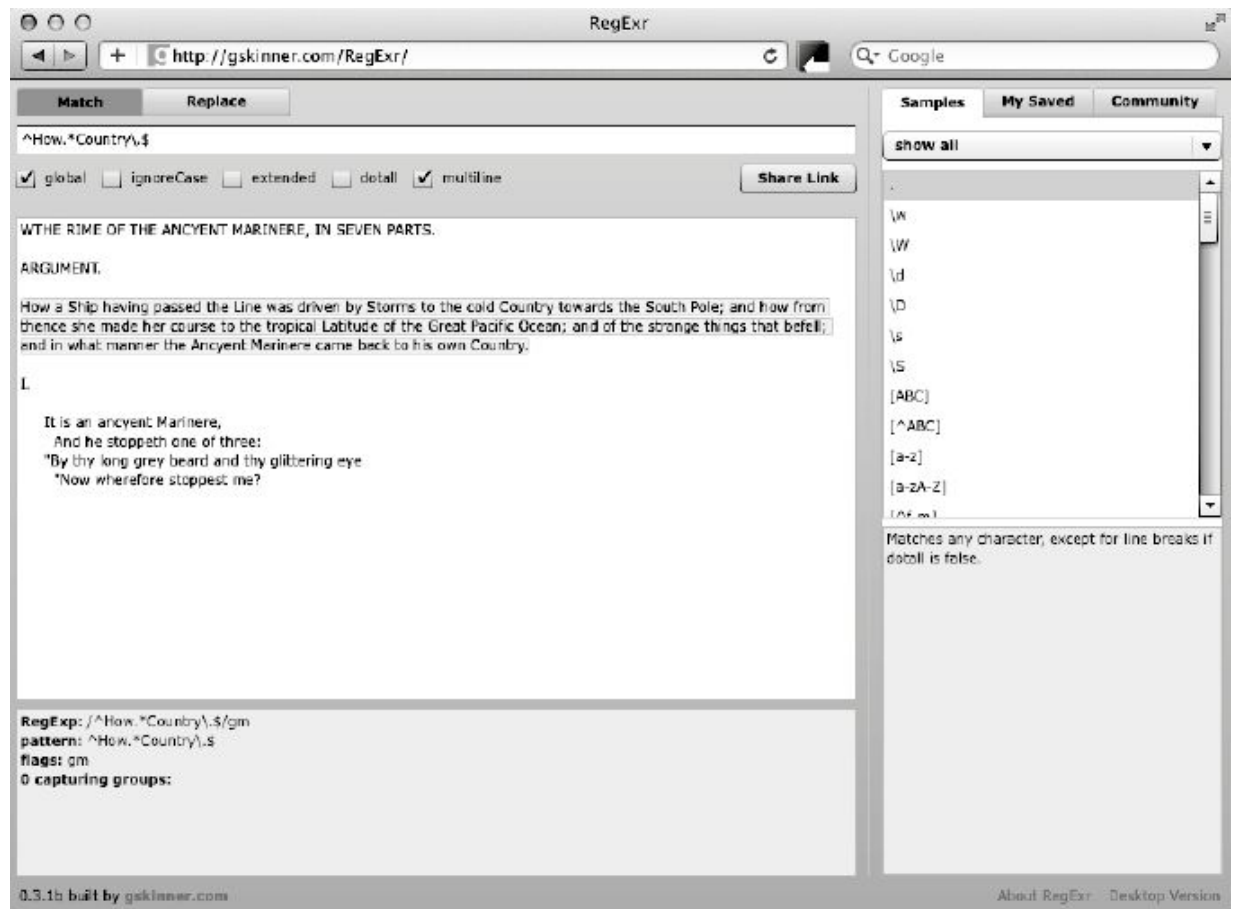


Figura 3.1 – RegExr no Safari.

Para corresponder ao final de uma linha ou string, como você já sabe, basta utilizar o sinal de cifrão: \$

No RegExr, certifique-se de que a opção *multiline* esteja selecionada. A opção *global* fica selecionada por padrão quando o RegExr é aberto, mas você pode deixá-la selecionada ou não neste exemplo. Quando a opção *multiline* não está selecionada, o texto-alvo como um todo é considerado uma única string.

Na caixa de texto superior, digite esta expressão regular: ^How.*Country\\.\$

A expressão acima corresponderá à linha toda que começa com a palavra *How*. Observe que o ponto no final da expressão encontra-se precedido por uma barra invertida. A barra escapa o ponto, de modo que ele é interpretado como um literal. Se o ponto não fosse escapado, a que ele iria corresponder? A qualquer caractere. Se

quiser fazer a correspondência de um ponto literal, você precisa escapá-lo ou colocá-lo em uma classe de caracteres (ver capítulo 5).

Se a opção *multiline* for desabilitada, o que acontece? O realce desaparece. Com a opção *multiline* desabilitada e com *dotall* selecionado, digite: `^THE.*\?$`

e você verá que esta expressão corresponde a todo o texto.

A opção *dotall* significa que o ponto corresponderá a mudanças de linha, além de corresponder a todos os demais caracteres. Remova a seleção de *dotall*, e a expressão não corresponderá a nada. No entanto, a expressão a seguir `^THE.*`

corresponderá à primeira linha. Clique na opção *dotall* novamente, e todo o texto será correspondido outra vez. O `\?$` não é necessário para fazer a correspondência até o final do texto.

Bordas de palavra e não-bordas de palavra

Você já viu o `\b` sendo usado inúmeras vezes. Ele marca uma borda de palavra. Experimente digitar: `\bTHE\b` e você verá que essa expressão corresponde às duas ocorrências de *THE* na primeira linha (com a opção *global* selecionada). Da mesma maneira que `^` ou `$`, `\b` é uma asserção de largura zero. Pode parecer que ele corresponde a coisas como um espaço em branco ou o início de uma linha, mas, na verdade, ele corresponde a um nada com largura zero. Você notou que os espaços ao redor do segundo *THE* não se encontram destacados? Isso ocorre porque esses espaços não fazem parte da correspondência. Não é a coisa mais fácil do mundo para se compreender, mas você entenderá melhor ao observar o que o `\b` faz e o que ele não faz.

Você também pode fazer correspondências de não-bordas de palavra. Uma não-borda de palavra corresponde a locais que não são equivalentes a uma borda de palavra, como uma letra ou um número dentro de uma palavra ou de uma string. Para corresponder a uma não-borda de palavra, experimente usar isto: `\Be\B`

e observe as correspondências efetuadas (ver figura 3.2). Você verá que a expressão corresponde a uma letra e minúscula quando essa estiver cercada por outras letras ou não-caracteres de palavra. Sendo uma asserção de largura zero, a expressão não corresponde aos caracteres ao redor, mas reconhece quando o `e` literal está cercado por não-bordas de palavra.

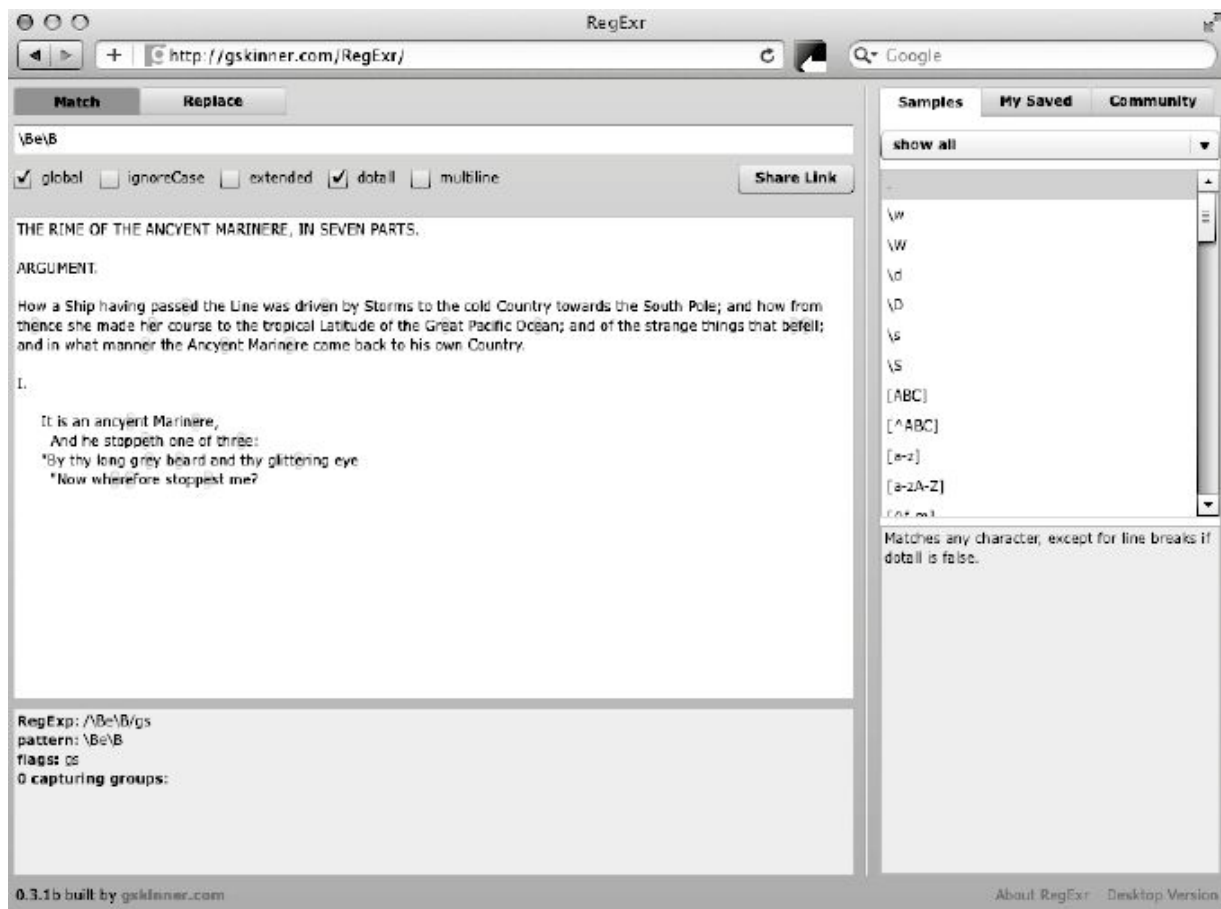


Figura 3.2 – Correspondendo a não-bordas de palavra usando `\B`.

Em alguns aplicativos, outra maneira de especificar uma borda de palavra é usando `\<` para o início de uma palavra e `\>` para o final de uma palavra. Essa é uma sintaxe mais antiga e não está disponível nos aplicativos mais novos de regex. Ela é útil em algumas ocasiões porque, diferentemente do `\b`, que corresponde a *qualquer* borda de palavra, essa sintaxe permite corresponder ao início ou ao fim de uma palavra.

Se você tiver `vi` ou `vim` em seu sistema, poderá experimentar essa sintaxe nesses editores. Basta seguir estes passos. As instruções

são fáceis mesmo que você nunca tenha usado *vim* antes. Em uma janela de comando ou de shell, mude para o diretório no qual se encontra o arquivo contendo o poema e abra-o usando: `vim rime.txt`. Depois, digite o seguinte comando de pesquisa `/` e tecele Enter ou Return. A barra para frente (`/`) é a maneira de começar uma pesquisa no *vim*. Observe o cursor e você verá que essa pesquisa encontrará os finais de palavras. Tecle `n` para repetir a busca. A seguir, digite `^` seguido de Enter ou Return. Desta vez, a pesquisa encontrará os inícios de palavras. Para sair do *vim*, basta digitar `zz`.

Essa sintaxe também funciona com o *grep*. Desde o início da década de 1970, o *grep*, assim como o *sed*, tem sido um dos pilares do Unix. (Na década de 1980, eu tinha um colega de trabalho que possuía uma placa de carro personalizada contendo a palavra GREP.) Experimente digitar este comando a partir de um prompt de shell: `grep -Eoc '\<(THE|The|the)\>' rime.txt`. A opção `-E` indica que você quer utilizar as expressões regulares estendidas (EREs, ou *Extended Regular Expressions*) em vez de usar as expressões regulares básicas (BREs, ou *Basic Regular Expressions*) que são usadas por padrão pelo *grep*. A opção `-o` significa que você quer mostrar, no resultado, somente a parte da linha que corresponde ao padrão, e a opção `-c` indica que somente um contador do resultado será retornado. O padrão entre aspas simples corresponde a *THE*, *The* ou *the* enquanto palavras inteiras. É isso que o `\<` e o `\>` ajudam a encontrar.

Esse comando retornará 259

que é o número de palavras encontradas.

Por outro lado, se o `\<` e o `\>` não forem incluídos, você obterá um resultado diferente. Experimente fazer isso desta maneira: `grep -Eoc '(THE|The|the)' rime.txt` e você obterá uma quantidade diferente: 327

Por quê? Porque o padrão corresponderá não somente a palavras inteiras, mas também a *qualquer* sequência de caracteres que contenha a palavra. Portanto, este é um dos motivos pelos quais o `\<` e o `\>` podem ser úteis.

Outras âncoras

O *shorthand* a seguir, que corresponde ao início de um assunto, é semelhante à âncora `^`: `\A` Esta expressão não está disponível em todas as implementações de *regex*, mas você a tem disponível em Perl e em PCRE (*Perl Compatible Regular Expressions*), por exemplo. Para corresponder ao final de um assunto, você pode utilizar o companheiro do `\A`, que é o: `\Z`

Além disso, em alguns contextos você pode usar o: `\Z` O *pcgrep* é uma versão do *grep* para a biblioteca PCRE. (Consulte a seção de “Notas técnicas” para descobrir onde obtê-lo.) Uma vez instalado, para experimentar essa sintaxe com o *pcgrep*, você poderia fazer algo do tipo: `pcgrep -c '\A\s*(THE|The|the)' rime.txt` o qual retornará um contador (-c) de 108 ocorrências da palavra *the* (nos três casos) que ocorrem próximas ao início de uma linha, precedidas por espaço em branco (zero ou mais). A seguir, digite este comando: `pcgrep -n '(MARINERE|Marinere)(.)?\Z' rime.txt` Haverá correspondência tanto de *MARINERE* quanto de *Marinere* no final de uma linha (assunto), seguida por qualquer caractere opcional, que neste caso corresponde tanto a um sinal de pontuação quanto à letra S. (Os parênteses ao redor do ponto não são essenciais.) Você verá esta saída: 1:THE RIME OF THE ANCYENT MARINERE, 10: It is an ancyent Marinere, 38: The bright-eyed Marinere.

63: The bright-eyed Marinere.

105: "God save thee, ancyent Marinere!

282: "I fear thee, ancyent Marinere!

702: He loves to talk with Mariners A opção -n usada com o *pcgrep* faz com que sejam mostrados os números de linha no início de cada linha da saída. As opções de linha de comando do *pcgrep* são bastante semelhantes às opções do *grep*. Para vê-las, digite: `pcre --help`

Especificando um grupo de caracteres como literais

Você pode usar estas sequências para especificar um conjunto de caracteres como literais: `\Q`

e

\E

Para demonstrar como isso funciona, digite os seguintes metacaracteres na caixa de texto inferior do RegExr: `.^$*+?|(){}[]\-`

Esses quinze metacaracteres são tratados como caracteres especiais em expressões regulares e são usados para codificar um padrão. (O hífen é tratado de modo especial, indicando um intervalo quando usado dentro de colchetes que representam uma classe de caracteres. Nos demais casos, o hífen não é considerado um caractere especial.) Se você tentar fazer a correspondência desses caracteres na caixa de texto superior do RegExr, nada acontecerá. Por quê? Porque o RegExr pensa (se é que ele pode pensar) que você está digitando uma expressão regular, e não caracteres literais. Agora experimente digitar: `\Q$E`

e a expressão corresponderá ao `$` porque qualquer coisa que estiver entre `\Q` e `\E` é interpretado como sendo um caractere literal (ver figura 3.3). (Lembre-se de que você pode colocar uma `\` antes de um metacaractere para transformá-lo em um literal.)

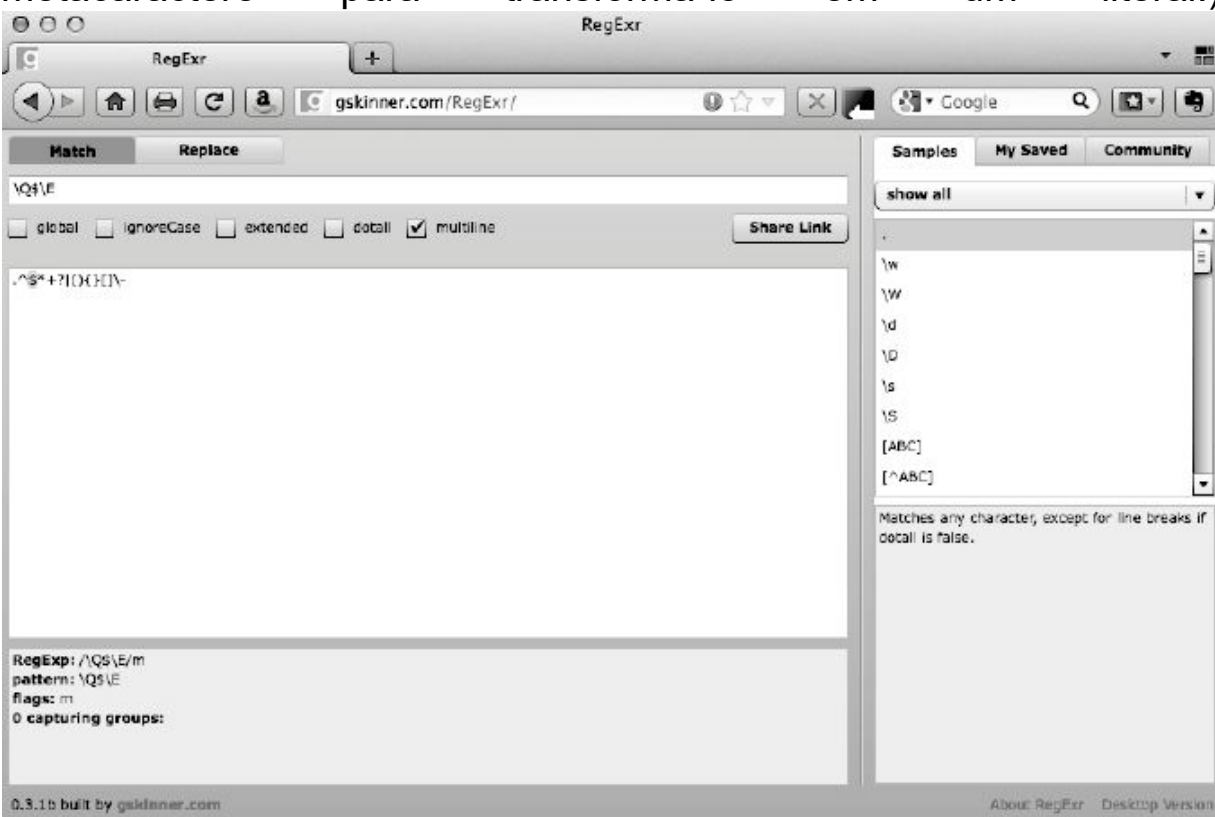


Figura 3.3 – Especificando metacaracteres como sendo literais.

Adicionando tags

No RegExr, remova a seleção da opção *global* e selecione *multiline*, clique na guia Replace e depois na primeira caixa de texto (identificada com o número 1 na figura 3.4), então digite: `^(.*)$`

Essa expressão corresponderá à primeira linha de texto e fará a sua captura. Em seguida, na caixa de texto seguinte (identificada com o número 2), digite a linha a seguir ou algo semelhante: `<!DOCTYPE html>\n<html lang="en">\n<head><title>Rime</title></head>\n<body>\n <h1>$1</h1>` Ao digitar o texto de substituição, você notará que o texto de assunto (mostrado na caixa de texto identificada com o número 3) será alterado na caixa de texto de resultados (identificada com o número 4) para incluir a marcação que você adicionou (ver figura 3.4).

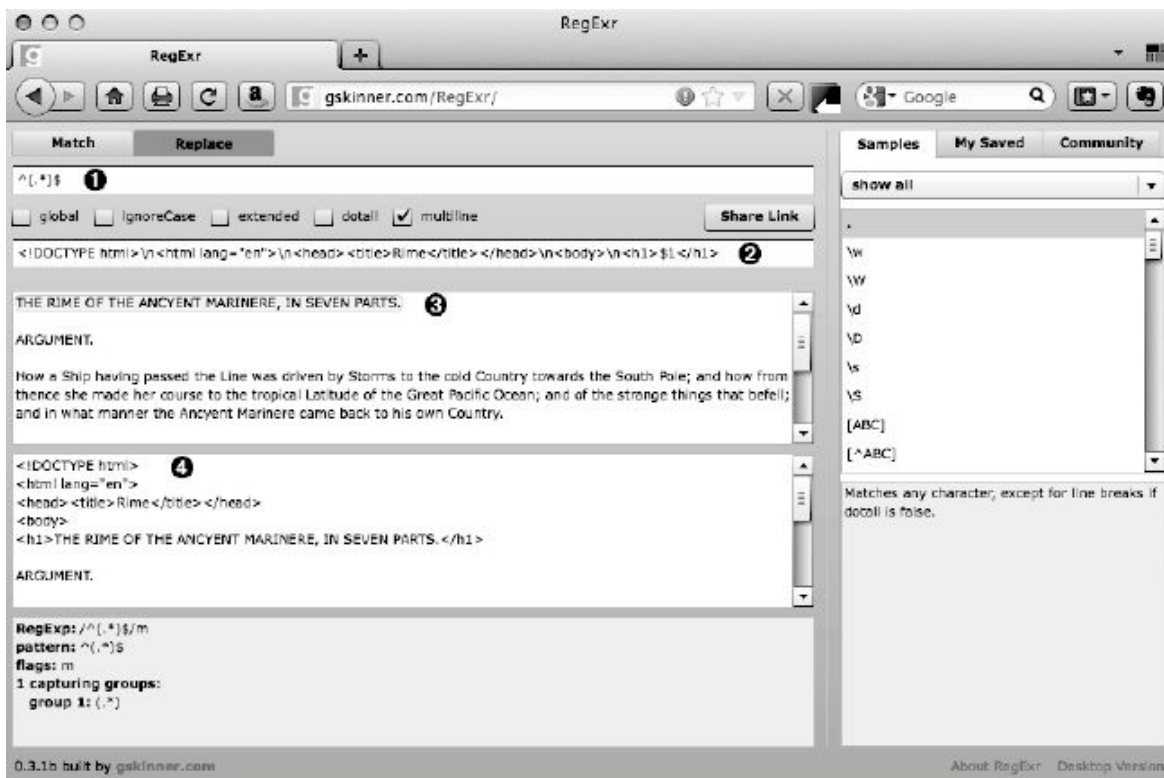


Figura 3.4 – Adicionando marcações com o RegExr.

O RegExr tem um bom desempenho para demonstrar uma maneira de inserir a marcação, mas é limitado quanto ao que pode fazer. Por exemplo, o RegExr não consegue salvar nenhum resultado em um arquivo. Precisamos ir além do navegador para ser capaz de fazer

isso.

Adicionado tags com o sed

Em uma linha de comando, você poderia também fazer algo similar ao que acabamos de fazer no RegExr usando *sed*, que você viu no capítulo anterior. O comando de inserção (*i*) do *sed* permite inserir um texto acima ou antes de um determinado local em um documento ou em uma string. A propósito, o contrário de *i* no *sed* é um *a*, o qual concatena texto abaixo ou depois de determinada localização. Usaremos o comando de concatenação mais tarde.

O comando a seguir insere o doctype do HTML5 e várias outras tags, começando na linha 1: `sed '1 i\ <!DOCTYPE html>\ <html lang="en">\ <head>\ <title>Rime</title>\ </head>\ <body> s/^/<h1>/ s/$/</h1>/'`

q' rime.txt As barras invertidas (**) no final das linhas permitem que você insira mudanças de linhas na cadeia e evite que o comando seja executado prematuramente. As barras invertidas na frente das aspas *escapam* as aspas, de modo que elas são interpretadas como caracteres literais, e não como parte do comando.

Se você executar esse comando *sed* corretamente, a saída será algo do tipo: `<!DOCTYPE html> <html lang="en"> <head> <title>The Rime of the Ancyent Mariner (1798)</title> </head> <body> <h1>THE RIME OF THE ANCYENT MARINERE, IN SEVEN PARTS.</h1>` Esses mesmos comandos do *sed* estão salvos no arquivo *top.sed*, contido no arquivo de exemplos. Você pode executar os mesmos comandos acima utilizando o arquivo por meio deste comando: `sed -f top.sed rime.txt` Você deverá obter a mesma saída apresentada no comando anterior. Se quiser salvar a saída em um arquivo, você pode redirecionar a saída para um arquivo, por exemplo, desta maneira: `sed -f top.sed rime.txt > temp` Além de mostrar o resultado na tela, a parte relativa ao redirecionamento do comando (`> temp`) fará com que a saída seja salva no arquivo *temp*.

Adicionando tags usando Perl

Vamos tentar fazer a mesma coisa usando Perl. Sem dar explicações para tudo o que está acontecendo, experimente digitar isto: `perl -ne 'print "<!DOCTYPE html>\ <html lang=\"en\">\ <head>`


```
<title>Rime</title></head>\    <body>\    "    if    $.    ==    1;
s/^/<h1>/;s/$/<\h1>/m;print;exit;' rime.txt
```

 Compare com o comando `sed`. De que modo são semelhantes? De que modo são diferentes? O comando `sed` é um pouco mais simples, mas o Perl é muito mais poderoso, em minha opinião.

Eis uma explicação de como o comando acima funciona: • A variável `$.`, que é testada pelo comando `if`, representa a linha atual. O comando `if` retorna `true`, indicando que passou no teste para saber se a linha atual é a linha 1.

- Quando o Perl encontra a linha 1 com o `if`, ele imprime o doctype e algumas tags HTML. É preciso escapar as aspas, do mesmo modo que foi feito com o `sed`.
- A primeira substituição insere uma tag `h1` inicial no começo da linha, e a segunda insere uma tag `h1` final no fim da linha. O `m` no final da segunda substituição significa que o modificador *multiline* está sendo usado. Isso é feito para que o comando reconheça o final da primeira linha. Sem o `m`, o `$` corresponderia ao final do arquivo.
- O comando `print` mostra o resultado das substituições.
- O comando `exit` sai imediatamente do Perl. Do contrário, por causa da opção `-n`, o Perl faria um laço passando por todas as linhas do arquivo, o que não queremos fazer neste script.

Há muita coisa para digitar aqui, de modo que eu coloquei todo esse código em Perl em um arquivo e o chamei de *top.pl*, o qual também pode ser encontrado no arquivo de códigos.

```
#!/usr/bin/perl -n if ($ == 1) {
print "<!DOCTYPE html>\ <html lang=\"en\">\ <head>\ <title>The Rime of the Ancyent
Mariner (1798)</title>\ </head>\ <body>\ ";
s/^/<h1>/; s/$/<\h1>/m; print; exit; }
```

Execute desta maneira: `perl top.pl rime.txt` Você obterá um resultado semelhante ao do comando anterior, embora esse seja formado de um modo um pouco diferente. (Você pode redirecionar a saída usando `>`, como foi feito com `sed`.) O próximo capítulo cobre alternância, grupos e retrovisores, entre outras coisas. Vejo você lá.

O que você aprendeu no capítulo 3

- Como usar âncoras no início ou no fim de uma linha com `^` ou `$`.
- Como usar bordas de palavra e não-bordas de palavra.
- Como corresponder ao início ou ao fim de um assunto usando `\A` e `\Z` (ou `\z`).
- Como especificar strings como sendo literais usando `\Q` e `\E`.
- Como adicionar tags em um documento usando RegExr, *sed* e Perl.

Notas técnicas

- O *vi* é um editor do Unix que usa expressões regulares e foi desenvolvido em 1976 pelo cofundador da Sun, Bill Joy. O editor *vim* é um substituto do *vi*, desenvolvido principalmente por Bram Moolenaar (ver <http://www.vim.org>). Um texto antigo de Bill Joy e Mark Horton sobre o *vi* pode ser encontrado no site <http://docs.freebsd.org/44doc/usd/12.vi/paper.html>. A primeira vez que eu usei o *vi* foi em 1983, e eu o uso quase todos os dias. Esse editor permite que eu faça mais coisas e de modo mais rápido que qualquer outro editor. E ele é tão poderoso que estou sempre descobrindo novos recursos que não conhecia antes, mesmo tendo contato com o editor há quase trinta anos.
- O *grep* é um utilitário de linha de comando do Unix para busca e apresentação de strings usando expressões regulares. Criado por Ken Thompson em 1973, diz-se que o *grep* evoluiu a partir do comando *g/re/p* (global/regular expression/print) do editor *ed*. Ele foi superado, mas não eliminado pelo *egrep* (ou *grep -E*), que utiliza expressões regulares estendidas (EREs) e possui metacaracteres adicionais, como `|`, `+`, `?`, `(` e `)`. O *fgrep* (*grep -F*) faz buscas em arquivos usando strings literais; metacaracteres como `$`, `*` e `|` não possuem significados especiais. O *grep* está disponível em sistemas Linux, bem como no Darwin do Mac OS X. Você também pode obtê-lo como parte do pacote Cygwin GNU (<http://www.cygwin.com>) ou fazer download dele a partir do site <http://gnuwin32.sourceforge.net/packages/grep.htm>.

- O PCRE (<http://www.pcre.org>) ou Perl Compatible Regular Expressions é uma biblioteca de funções em C (8 bits e 16 bits) para expressões regulares compatíveis com Perl 5 que inclui alguns recursos de outras implementações. O *pcregrep* é uma ferramenta para 8 bits, no estilo do *grep*, que permite que você utilize os recursos da biblioteca PCRE na linha de comando. Você pode obter o *pcregrep* para o Mac no Macports (<http://www.macports.org>) executando o comando `sudo port install pcre`. (O Xcode é um pré-requisito; ver <https://developer.apple.com/technologies/tools/>. É necessário fazer login.)

capítulo 4

Alternância, grupos e retrovisores

Você já viu os grupos em ação. Os grupos circundam texto entre parênteses para auxiliar na execução de algumas operações, tais como:

- efetuar alternância, ou seja, uma escolha dentre dois ou mais padrões opcionais;
- criar subpadrões;
- capturar um grupo para referência posterior usando um retrovisor;
- aplicar uma operação a um padrão agrupado, como, por exemplo, uma quantificação;
- usar grupos de não-captura;
- agrupamento atômico (avançado).

Usaremos alguns exemplos bem planejados, além do texto “The Rime of the Ancyent Marinere” novamente, que se encontra no arquivo *rime.txt*. Desta vez, usarei a versão do RegExr para desktop, bem como outras ferramentas, como o *sed*. Você pode fazer o download da versão desktop do RegExr a partir do site <http://www.regexr.com>, para Windows, Mac ou Linux (foi escrito com Adobe AIR). Clique no link para Desktop Version na página do RegExr (canto inferior à direita) para mais informações.

Alternância

Dito de maneira simples, a *alternância* oferece uma escolha de padrões alternativos com os quais a correspondência será efetuada. Por exemplo, suponhamos que você queira descobrir quantas ocorrências do artigo *the* estão presentes em “The Rime of the Ancyent Marinere”. O problema é que a palavra ocorre como *THE*, *The* e *the* no poema. Você pode usar a alternância para lidar com essa peculiaridade.

Abra a aplicação RegExr na versão desktop clicando duas vezes no seu ícone. Essa versão se parece muito com a versão online, mas tem a vantagem de executar localmente em seu computador, de modo que você não estará sujeito aos problemas de rede que às vezes ocorrem quando aplicações web são usadas.

Eu copiei e coleí o poema todo na versão desktop do RegExr para o próximo exercício. Estou usando o RegExr em um Mac rodando OS X Lion.

Na caixa de texto superior, digite o padrão: (the|The|THE) e você verá todas as ocorrências de *the* destacadas no poema na caixa de texto inferior (ver figura 4.1). Use a barra de rolagem para ver mais resultados.

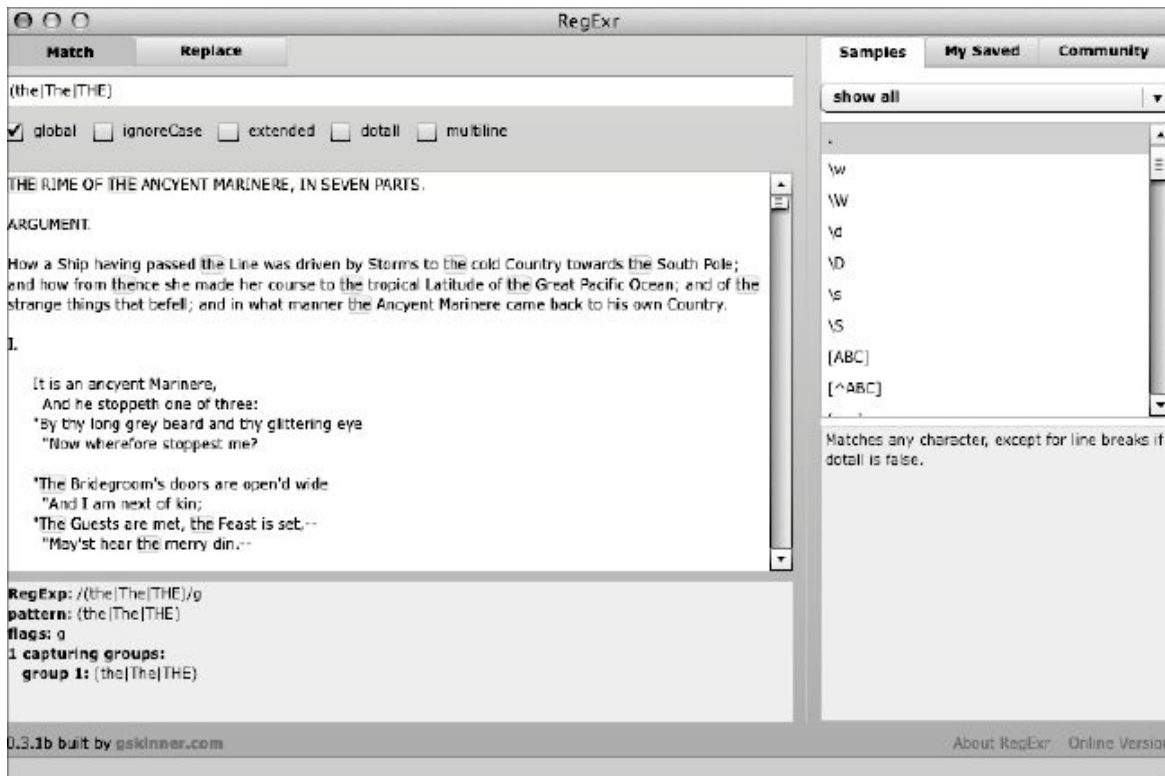


Figura 4.1 – Usando alternância na versão desktop do RegExr.

Podemos tornar esse grupo mais conciso aplicando uma opção. As opções permitem especificar a maneira pela qual você gostaria de buscar um padrão. Por exemplo, a opção (?i) faz com que seu padrão não diferencie letras maiúsculas de minúsculas; em vez de usar o padrão original com alternância, você pode fazer o seguinte: (?i)the Experimente digitar essa expressão no RegExr para ver como ela funciona. Você também pode tornar a busca não sensível ao caso selecionando a opção *ignoreCase* no RegExr, mas ambos os métodos funcionam. Essa e outras opções ou modificadores encontram-se listados na tabela 4.1.

Tabela 4.1 – Opções em expressões regulares

Opção	Descrição	Suportado por
(?d)	Linhas Unix	Java
(?i)	Não sensível ao caso	PCRE, Perl, Java
(?J)	Permitir nomes duplicados	PCRE*
(?m)	Múltiplas linhas	PCRE, Perl, Java
(?s)	Linha única (dotall)	PCRE, Perl, Java
(?u)	Diferenciar caso em Unicode	Java
(?U)	Correspondência preguiçosa (<i>lazy</i>) por padrão	PCRE
(?x)	Ignorar brancos, comentários	PCRE, Perl, Java
(?-...)	Desabilitar ou desligar opções	PCRE

* Consulte a seção “Named Subpatterns” em <http://www.pcre.org/pcre.txt>.

Vamos usar a alternância com *grep*. As opções na tabela 4.1, a propósito, não funcionam com *grep*, então você vai usar o padrão original de alternância. Para contar o número de linhas nas quais a palavra *the* ocorre, independentemente de usar letras maiúsculas ou minúsculas, uma ou mais vezes, utilize: `grep -Ec "(the|The|THE)" rime.txt` e você obterá o resultado a seguir: 327

Esse resultado não conta a história toda. Fique ligado.

Aqui está uma análise do comando *grep*: • A opção `-E` significa que você deseja usar as expressões regulares estendidas (EREs), e não as expressões regulares básicas (BREs). O uso dessa opção evita, por exemplo, a necessidade de escapar os parênteses e a barra vertical, como em `\(THE\|The\|the\)`, conforme você faria se usasse BREs.

- A opção `-c` retorna um contador das linhas correspondidas (e não das palavras correspondidas).
- Os parênteses agrupam as opções ou alternativas entre *the*, *The*

ou *THE*.

- A barra vertical separa as opções possíveis, que são avaliadas da esquerda para a direita.

Para obter o número de palavras usadas, a abordagem a seguir fará com que cada uma das ocorrências da palavra seja retornada, uma em cada linha: `grep -Eo "(the|The|THE)" rime.txt | wc -l` Este comando retorna: 412

E aqui está uma análise um pouco mais detalhada: • A opção `-o` indica que somente a parte da linha que corresponde ao padrão deverá ser mostrada, embora isso não seja visível por causa do pipe (`|`) para `wc`.

- A barra vertical, nesse contexto, faz o pipe da saída do comando `grep` para a entrada do comando `wc`. O `wc` é um comando para contagem de palavras, e `-l` conta o número de linhas da entrada.

Por que a diferença tão grande entre 327 e 412? Porque `-c` fornece um contador de linhas com as quais houve correspondência, mas pode ter havido mais de uma correspondência por linha. Se você usar `-o` com `wc -l`, então cada ocorrência das diversas formas da palavra aparecerá em uma linha separada e será contabilizada, resultando no número maior.

Para executar essa mesma correspondência com Perl, escreva seu comando desta maneira: `perl -ne 'print if (the|The|THE)' rime.txt` Ou, melhor, você pode fazer isso com a opção `(?i)` mencionada anteriormente, mas sem a alternância: `perl -ne 'print if (?i)the' rime.txt` Ou, melhor ainda, concatene o modificador `i` depois do último delimitador de padrão: `perl -ne 'print if thei' rime.txt` e você obterá o mesmo resultado. Quanto mais simples, melhor. Para uma lista de modificadores adicionais (também conhecidos como *flags*), consulte a tabela 4.2. Além disso, compare com as opções (similares, mas com uma sintaxe diferente) que estão na tabela 4.1.

*Tabela 4.2 – Modificadores do Perl (flags)**

Modificador	Descrição
a	Corresponder <code>\d</code> , <code>\s</code> , <code>\w</code> e POSIX somente no intervalo ASCII

c	Manter a posição atual depois de falha na correspondência
d	Usar regras padrão, nativas da plataforma
g	Correspondência global
i	Correspondência não sensível ao caso
l	Usar regras da localidade atual
m	Strings de múltiplas linhas
p	Conservar a string correspondida
s	Tratar strings como única linha
u	Usar regras Unicode ao corresponder
x	Ignorar brancos e comentários

* Ver <http://perldoc.perl.org/perlre.html#Modifiers>.

Subpadrões

Na maioria das vezes, quando nos referimos a *subpadrões* em expressões regulares, estamos nos referindo a um grupo ou grupos dentro de grupos. Um subpadrão é um padrão dentro de um padrão. Geralmente, uma condição em um subpadrão pode ser correspondida quando um padrão anterior é correspondido, mas nem sempre é assim. Os subpadrões podem ser configurados de diversas maneiras, mas estamos preocupados aqui principalmente com aqueles que são definidos dentro de parênteses.

Em certo sentido, o padrão que você viu antes: `(the|The|THE)` possui três subpadrões: *the* é o primeiro subpadrão, *The* é o segundo e *THE* é o terceiro, mas corresponder ao segundo padrão, neste caso, não depende da correspondência do primeiro. (A correspondência é feita em primeiro lugar com o padrão mais à esquerda.) Mas aqui está um caso no qual o(s) subpadrão(ões) depende(m) do padrão anterior: `(t|T)h(e|eir)` Falando de modo claro,

essa expressão corresponderá aos caracteres literais *t* ou *T*, seguidos por um *h* seguido por uma letra *e* ou pelas letras *eir*. Sendo assim, esse padrão corresponderá a qualquer uma das palavras abaixo: • *the* • *The* • *their* • *Their* Neste caso, o segundo subpadrão (*e|eir*) é dependente do primeiro (*tT*).

Os subpadrões não exigem parênteses. Aqui está um exemplo de subpadrões usados com classes de caracteres: `\b[tT]h[ceinry]*\b` Esse padrão, além de corresponder a *the* ou a *The*, pode corresponder também a palavras como *thee*, *thy* e *thence*. As duas bordas de palavra (`\b`) significam que o padrão corresponderá a palavras inteiras, e não a letras dentro de outras palavras.

Aqui está uma análise completa desse padrão: • `\b` corresponde a uma borda inicial de palavra.

- `[tT]` é uma classe de caracteres que corresponde a uma letra *t* minúscula ou a uma letra *T* maiúscula. Podemos considerar esse como sendo o primeiro subpadrão.
- Em seguida, o padrão corresponde (ou tenta corresponder) a uma letra *h* minúscula.
- O segundo ou último subpadrão também é expresso na forma de uma classe de caracteres `[ceinry]`, seguido por um quantificador `*` indicando zero ou mais ocorrências.
- Por último, outra fronteira de palavra `\b` finaliza o padrão.



Um aspecto interessante da situação das expressões regulares é que a terminologia, embora seja geralmente próxima em termos de significado, também pode variar bastante. Ao definir *subpadrão* e outros termos neste livro, eu analisei inúmeras fontes e tentei reuni-las sob o mesmo teto. Mas suspeito de que haja pessoas que argumentarão que uma classe de caracteres não é um subpadrão. Minha posição é a de que elas podem funcionar como subpadrões, por isso eu as incluí.

Capturando grupos e usando retrovisores

Quando um padrão agrupa todo um conteúdo ou uma parte dele entre parênteses, ele captura esse conteúdo e o armazena temporariamente na memória. Você pode reutilizar esse conteúdo,

se quiser, usando um retrovisor, da seguinte maneira: \1

ou:

\$1

em que \1 ou \$1 faz referência ao primeiro grupo capturado, \2 ou \$2 faz referência ao segundo grupo capturado, e assim por diante. O *sed* aceita somente o formato \1, mas o Perl aceita ambos.



Originalmente, o *sed* suportava retrovisores no intervalo de \1 a \9, mas essa limitação aparentemente não existe mais.

Você já viu os retrovisores em ação, mas vou demonstrá-los novamente. Usaremos a expressão a seguir para reorganizar a ordem das palavras em uma linha do poema, com minhas desculpas a Samuel Taylor Coleridge. Na caixa de texto superior do RegExr, depois de clicar na guia Replace, digite este padrão: (It is) (an ancyent Marinere) Faça a rolagem do texto de assunto (terceira área de texto) até poder ver a linha que se encontra em destaque, e então, na segunda caixa de texto, digite: \$2 \$1

e você verá a linha reorganizada na caixa de texto mais abaixo: an ancyent Marinere It is, (Ver figura 4.2)

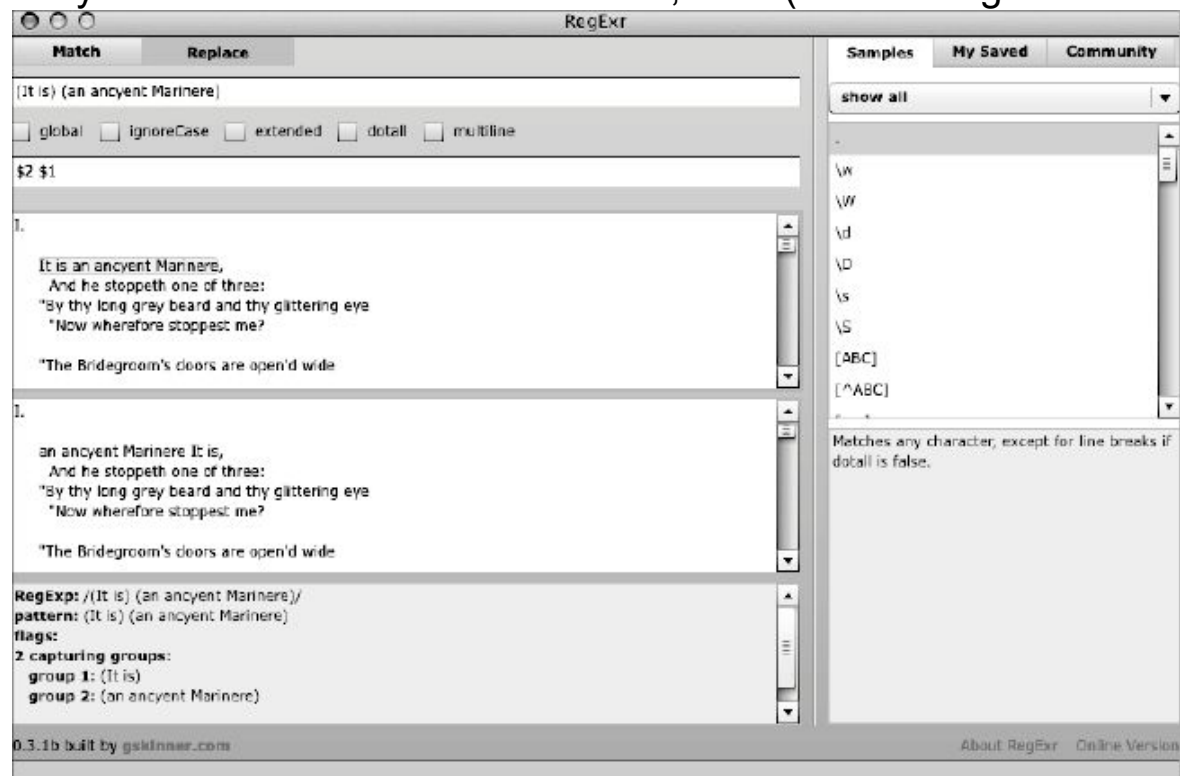


Figura 4.2 – Fazendo referências para trás usando \$1 e \$2.

Aqui está uma maneira de obter o mesmo resultado usando `sed`: `sed -E 's/(It is) (an ancyent Marinere)/\2 \1/p' rime.txt` A saída será: `an ancyent Marinere It is`, exatamente como no `RegExr`. Vamos analisar o comando `sed` para ajudá-lo a compreender tudo o que está acontecendo:

- A opção `-E` mais uma vez invoca as EREs, de modo que você não precisa colocar escape nos parênteses, por exemplo.

- A opção `-n` inibe o comportamento padrão, que é o de mostrar todas as linhas.
- O comando de substituição busca uma correspondência para o texto “`It is an ancyent Marinere`”, capturando-o em dois grupos.
- O comando de substituição também substitui a correspondência, reorganizando o texto capturado na saída, usando os retrovisores `\2`, primeiro, e depois o `\1`.
- O `p` no final do comando de substituição significa que você quer que a linha seja mostrada.

Um comando similar no Perl fará a mesma coisa: `perl -ne 'print if s/(It is) (an ancyent Marinere)/\2 \1/' rime.txt` Observe que o comando acima utiliza o estilo de sintaxe `\1`. Você pode usar a sintaxe `$1` também, é claro: `perl -ne 'print if s/(It is) (an ancyent Marinere)/$2 $1/' rime.txt` Eu gosto do modo como o Perl permite mostrar uma linha selecionada, sem ter de passar por laços.

Gostaria de ressaltar alguns pontos acerca da saída: `an ancyent Marinere It is`. As letras maiúsculas ficam misturadas na transformação. O Perl corrige isso com o `\u` e o `\l`, desta maneira: `perl -ne 'print if s/(It is) (an ancyent Marinere)/\u$2 \l$1/' rime.txt` Agora o resultado tem uma aparência muito melhor: `An ancyent Marinere it is`. E aqui está a explicação:

- A sintaxe `\l` não corresponde a nada, mas transforma o caractere que se segue para letra minúscula.

- A sintaxe `\u` transforma o caractere que vem a seguir em letra maiúscula.
- A diretiva `\U` (não mostrada) converte toda a string de texto que se segue em letras maiúsculas.
- A diretiva `\L` (não mostrada) converte toda a string de texto que se

segue em letras minúsculas.

Essas diretivas permanecem ativas até que outra diretiva seja encontrada (como, por exemplo, `\l` ou `\E`, o final de uma string especificada como literal). Faça experiências com essas sintaxes para ver como elas funcionam.

Grupos nomeados

Grupos nomeados são grupos capturados com nomes. Você pode acessar esses grupos pelo nome posteriormente, em vez de acessá-los usando um número inteiro. Mostrarei como isso pode ser feito usando Perl: `perl -ne 'print if s/(?<one>It is) (?<two>an ancyent Marinere)/\u${two}'`

`\u${one}' rime.txt` Vamos analisar este comando: • Adicionar `?<one>` e `?<two>` dentro dos parênteses nomeia os grupos como *one* e *two*, respectivamente.

- `${one}` faz referência ao grupo que se chama *one*, e `${two}`, ao grupo que se chama *two*.

Você também pode reutilizar os grupos nomeados dentro do padrão no qual o grupo foi nomeado. Vou demonstrar o que eu quero dizer. Suponhamos que você esteja procurando uma string que contenha seis zeros juntos: `000000`

É um exemplo simplório, mas serve para mostrar como isso funciona. Dê um nome a um grupo de três zeros usando este padrão (o *z* é arbitrário): `(?<z>0{3})` Você pode usar o grupo novamente, do seguinte modo: `(?<z>0{3})k<z>` Ou assim:

`(?<z>0{3})k'z'`

Ou assim:

`(?<z>0{3})g{z}`

Experimente usar essas expressões no RegExr para obter resultados rápidos. Todos os exemplos acima funcionarão. A tabela 4.3 mostra muitas das possibilidades para a sintaxe de grupos nomeados.

Tabela 4.3 – Sintaxe de grupos nomeados

Sintaxe	Descrição

(?<nome>...)	Um grupo nomeado
(?nome...)	Outro grupo nomeado
(?P<nome>...)	Um grupo nomeado em Python
\k<nome>	Referência pelo nome em Perl
\k'nome'	Referência pelo nome em Perl
\g{nome}	Referência pelo nome em Perl
\k{nome}	Referência pelo nome em .NET
(?P=nome)	Referência pelo nome em Python

Grupos de não-captura

Também existem grupos que são grupos de não-captura, ou seja, grupos que não armazenam seu conteúdo na memória. Às vezes, isso pode ser uma vantagem, especialmente se você não tem nenhuma intenção de fazer referência ao grupo. Pelo fato de não armazenar seu conteúdo, é possível que você tenha uma performance melhor, embora problemas de performance sejam dificilmente perceptíveis na execução de exemplos simples como os deste livro.

Você se lembra do primeiro grupo que foi discutido neste capítulo? Aqui está ele novamente: (the|The|THE) Você não precisa fazer referências para trás em nenhuma ocasião, portanto, você poderia escrever um grupo de não-captura desta maneira: (?:the|The|THE) Voltando ao início deste capítulo, você poderia adicionar uma opção para fazer com que o padrão não diferencie letras maiúsculas de minúsculas, desta maneira (embora a opção faça com que seja eliminada a necessidade de um grupo): (?:the|The|THE) Ou você poderia fazer isso desta maneira: (?:the|The|THE) Ou, melhor ainda, para fechar com *chave de ouro*: (?:the|The|THE) A letra correspondente à opção, i, pode ser inserida entre o ponto de interrogação e os dois-pontos.

Grupos atômicos

Outro tipo de grupo de não-captura é o *grupo atômico*. Se você está usando uma ferramenta de regex que faz backtracking, esse grupo vai desabilitar o backtracking – não para toda a expressão regular, mas somente para a parte que está incluída no grupo atômico. O formato da sintaxe é: `(?>the)` Em que ocasiões você iria usar grupos atômicos? Uma das coisas que pode tornar o processamento de regex realmente lento é o backtracking. O motivo pelo qual isso acontece é que, pelo fato de testar todas as possibilidades, o backtracking consome tempo e recursos de processamento. Às vezes, um bocado de tempo é consumido. Quando a situação fica realmente ruim, chamamos isso de *backtracking catastrófico*.

Você pode desabilitar todo o backtracking usando uma ferramenta que seja do tipo não-backtracking, como o `re2` (<http://code.google.com/p/re2/>), ou desabilitando-o para partes de sua expressão regular com grupamento atômico.



Meu foco neste livro é introduzir a sintaxe. Falarei muito pouco sobre ajuste de performance aqui. Grupos atômicos têm a ver principalmente com performance, em minha opinião.

No capítulo 5, você aprenderá a respeito das classes de caracteres.

O que você aprendeu no capítulo 4

- Que a alternância permite uma escolha entre dois ou mais padrões.
- O que são modificadores e como usá-los em um padrão.
- Diferentes tipos de subpadrões.
- Como usar grupos de captura e retrovisores.
- Como usar grupos nomeados e como referenciá-los.
- Como usar grupos de não-captura.
- Um pouco sobre grupamento atômico.

Notas técnicas

- O runtime do Adobe AIR permite que você use HTML, JavaScript, Flash e ActionScript para construir aplicações web que executam como aplicações cliente standalone, sem a necessidade de se usar um navegador. Obtenha mais informações acessando o site <http://www.adobe.com/products/air.html>.
- O Python (<http://www.python.org>) é uma linguagem de programação de alto nível, fácil de ser compreendida. Ela contém uma implementação de expressões regulares (ver <http://docs.python.org/library/re.html>).
- O .NET (<http://www.microsoft.com/net>) é um framework de programação para a plataforma Windows. Esse framework também contém uma implementação de expressões regulares (ver <http://msdn.microsoft.com/en-us/library/hs600312.aspx>).
- Explicações mais avançadas acerca de agrupamento atômico estão disponíveis em <http://www.regular-expressions.info/atomic.html> e em <http://stackoverflow.com/questions/6488944/atomic-group-and-non-capturing-group>.

capítulo 5

Classes de caracteres

Falarei mais sobre as classes de caracteres ou as *expressões entre colchetes*, como às vezes são chamadas. As classes de caracteres ajudam a fazer correspondências de caracteres específicos ou de sequências de caracteres específicos. Essas classes podem ser tão amplas ou abrangentes quanto os *shorthands* de caracteres – por exemplo, o *shorthand* de caracteres `\d` corresponderá aos mesmos caracteres que: 0-9

Mas você pode usar as classes de caracteres para ser mais específico ainda. Nesse aspecto, elas são mais versáteis que os *shorthands*.

Experimente usar esses exemplos em qualquer processador de regex de sua preferência. Eu usarei o Rubular no Opera e o Reggy no desktop.

Para executar este teste, insira a string abaixo na caixa de texto de assunto ou de texto-alvo na página web: ! " # \$ % & ' () * + , - . /

0 1 2 3 4 5 6 7 8 9

:: ; < = > ? @

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

[\] ^ _ `

a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~

Não é necessário digitar tudo isso. Você encontrará esse texto armazenado no arquivo *asciigraphic.txt* no arquivo de códigos que acompanha este livro.

Para começar, utilize uma classe de caracteres para corresponder a um conjunto de caracteres do inglês – neste caso, as vogais do inglês: `[aeiou]`

As vogais minúsculas devem ficar destacadas na caixa de texto inferior (ver figura 5.1). Como você faria para que as vogais maiúsculas ficassem destacadas? Como você faria para deixar as duas destacadas, ou seja, para corresponder às duas?

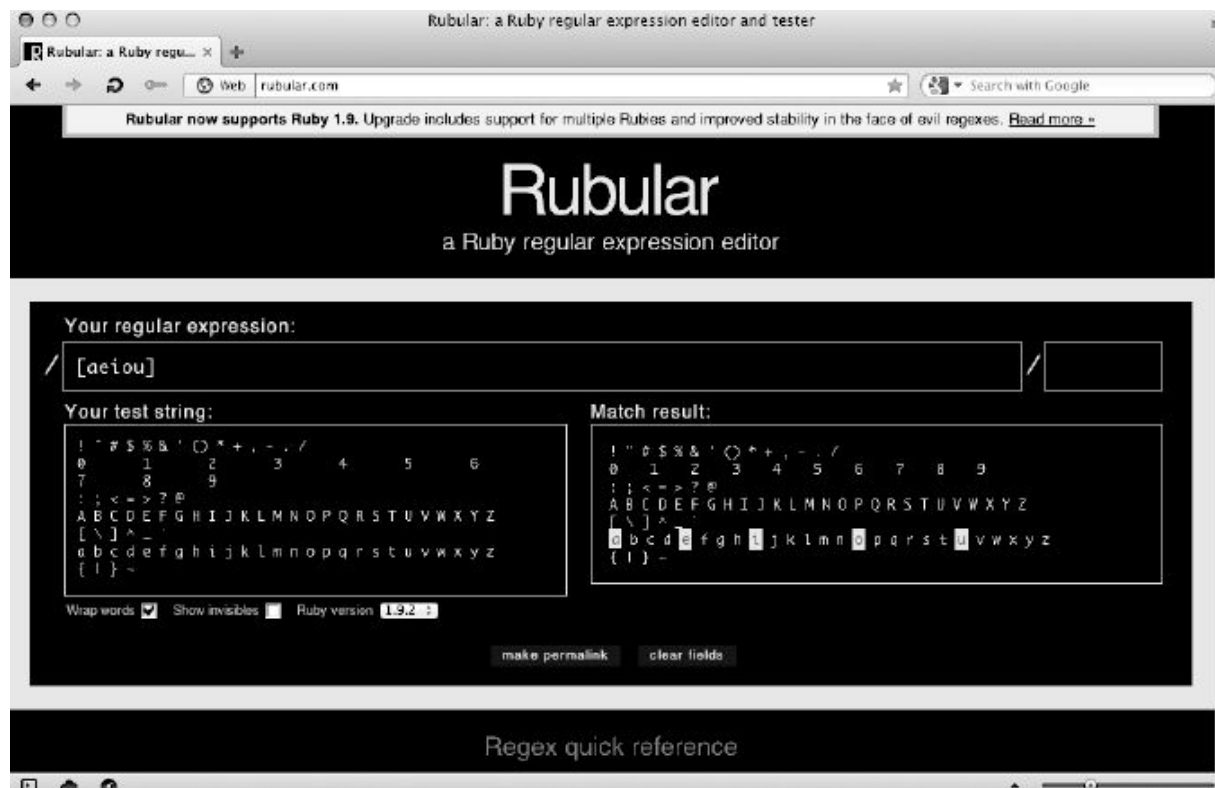


Figura 5.1 – Classe de caracteres no Rubular no navegador Opera.

Usando classes de caracteres, você também pode corresponder a um intervalo de caracteres: [a-z]

Esta expressão corresponde às letras minúsculas de *a* até *z*. Experimente fazer a correspondência usando um intervalo menor de caracteres, algo do tipo de *a* até *f*: [a-f]

É claro que você também pode especificar um intervalo de dígitos: [0-9]

Ou um intervalo menor ainda, como, por exemplo, 3, 4, 5 e 6: [3-6]

Agora, vamos expandir seu horizonte. Se quisesse fazer a correspondência de números pares no intervalo de 10 a 19, você poderia combinar duas classes de caracteres, lado a lado, desta maneira: \b[1][24680]\b Ou você poderia ir mais adiante ainda e procurar pelos números pares no intervalo de 0 a 99 usando isto (sim, conforme aprendemos na escola, zero é par): \b[24680]\b\b[1-9][24680]\b Se quisesse criar uma classe de caracteres que corresponde a dígitos hexadecimais, como você faria isso? Aqui vai uma dica: [a-fA-F0-9]

Você também pode usar *shorthands* dentro de uma classe de caracteres. Por exemplo, para corresponder a espaços em branco e caracteres de palavra, você poderia criar uma classe de caracteres como esta: `[\w\s]`

que é igual a: `[_a-zA-Z \t\n\r]`

mas muito mais fácil de ser digitada.

Classes de caracteres negados

Você já viu essa sintaxe diversas vezes, portanto, serei breve. Uma classe de caracteres negados faz a correspondência de caracteres que não correspondem ao conteúdo da classe. Por exemplo, se não quisesse corresponder às vogais, você poderia escrever (experimente usar esta expressão em seu navegador e então observe a figura 5.2): `^[^aeiou]`

Em sua essência, o circunflexo (^) no início de uma classe significa “não, eu não quero estes caracteres”. (O circunflexo *deve* ser colocado no início.)

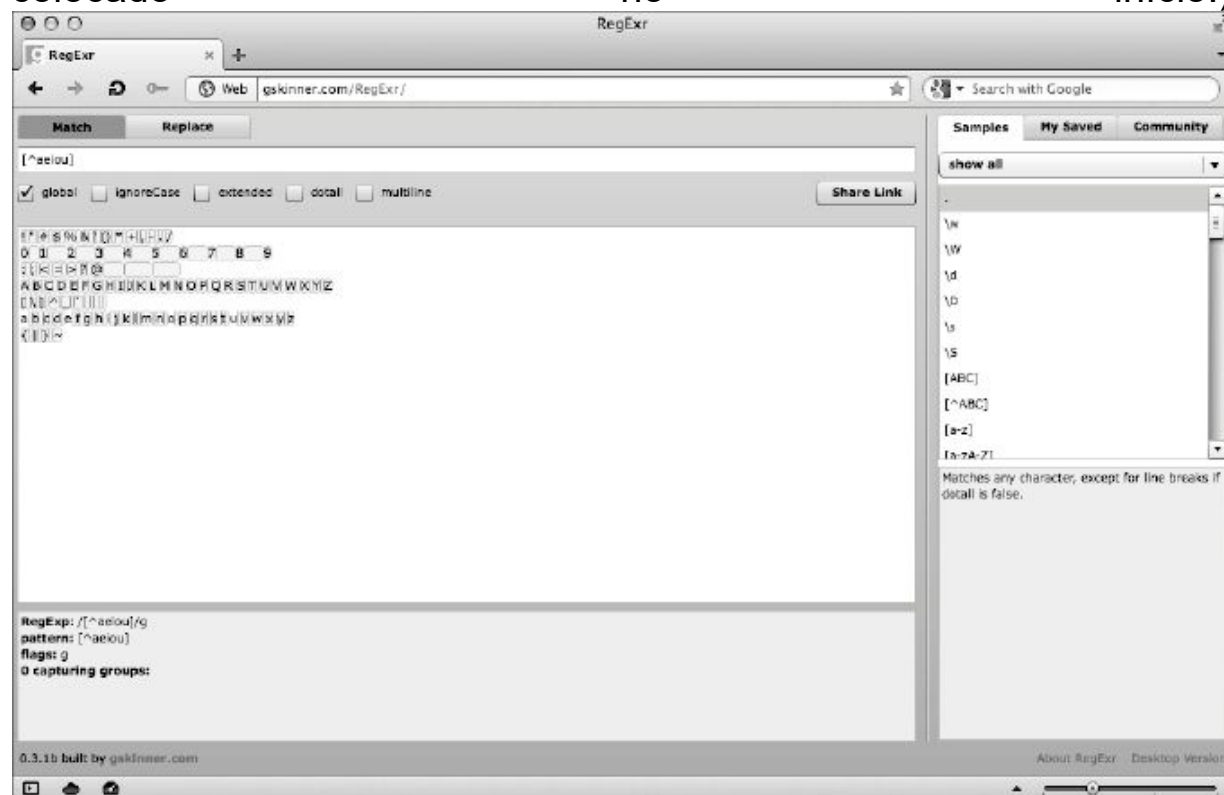


Figura 5.2 – Classe de caracteres negados no Regexpal no navegador Opera.

União e diferença

As classes de caracteres podem atuar como conjuntos. De fato, outro nome para uma classe de caracteres é *conjunto de caracteres*. Essa funcionalidade não é suportada por todas as implementações. Mas o Java a suporta.

Mostrarei agora uma aplicação desktop do Mac que se chama Reggy (consulte “Notas técnicas”). Em *Preferences* (Figura 5.3), eu alterei *Regular Expression Syntax* para Java, e em *Font* (sob *Format*), eu alterei o tamanho para 24 pontos para ficar mais legível.

Se quiser fazer a união de dois conjuntos de caracteres, você pode fazê-lo desta maneira: `[0-3[6-9]]`

A regex corresponderia ao intervalo de 3 a 6 ou de 6 a 9. A figura 5.4 mostra como o uso dessa expressão é apresentado no Reggy.



Figura 5.3 – Preferências do Reggy.



Figura 5.4 – União de dois conjuntos de caracteres no Reggy.

Para corresponder a uma diferença (essencialmente a uma subtração), use: `[a-z&&[^m-r]]`

que faz a correspondência de todas as letras de *a* até *z*, exceto de *m* até *r* (ver figura 5.5).



Figura 5.5 – Diferença entre dois conjuntos de caracteres no Reggy.

Classes de caracteres POSIX

POSIX ou Portable Operating System Interface é uma família de padrões mantida pelo IEEE. Essa família inclui um padrão para expressões regulares (ISO/IEC/IEEE 9945:2009), o qual disponibiliza um conjunto de classes de caracteres com nomes que assumem o formato: `[[:XXXX:]]`

em que `xxxx` é um nome, como, por exemplo, `digit` ou `word`.

Para corresponder a caracteres alfanuméricos (letras e dígitos), experimente usar: `[[:alnum:]]`

A figura 5.6 mostra a classe de alfanuméricos no Rubular.

Uma alternativa a essa classe é utilizar simplesmente o *shorthand* `\w`. O que é mais fácil de digitar, a classe de caracteres POSIX ou o *shorthand*? Você sabe aonde quero chegar: quanto menos teclas forem digitadas, melhor. Admito que eu não uso as classes POSIX com muita frequência, mas, ainda assim, vale a pena conhecê-las.

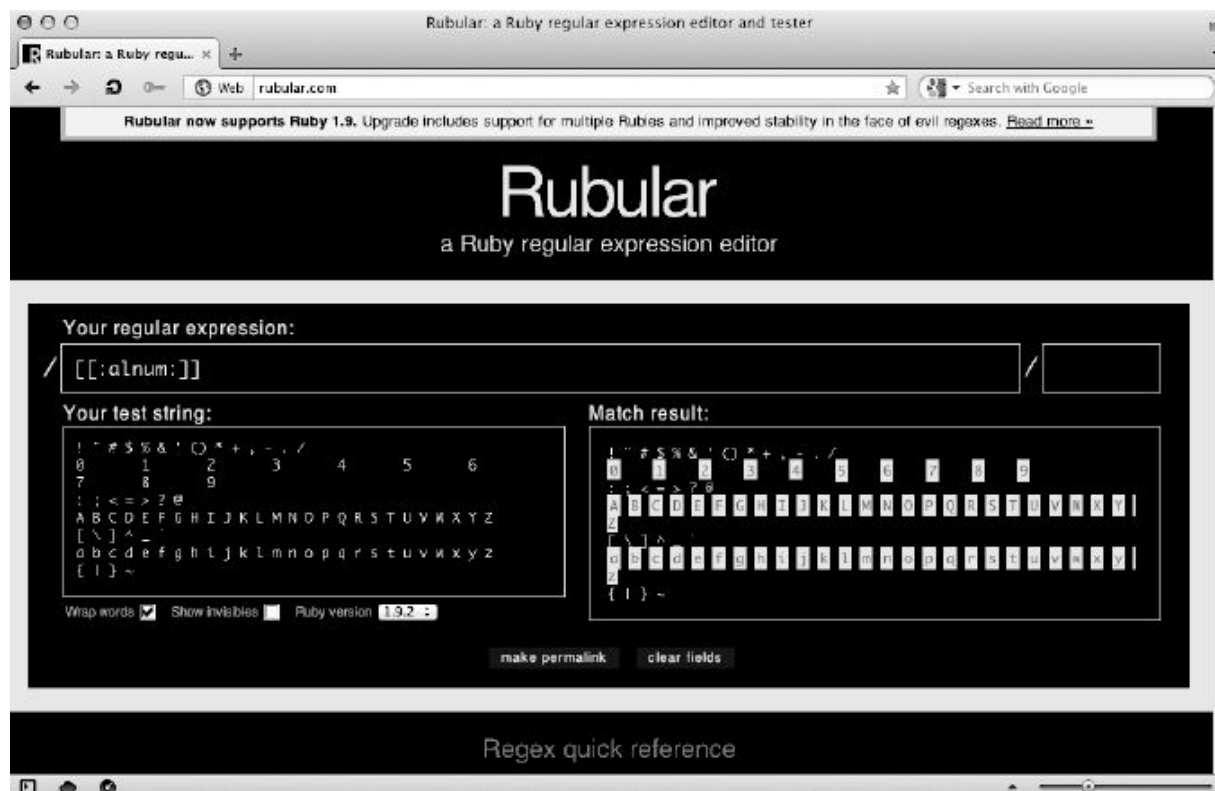


Figura 5.6 – Classe de caracteres alfanuméricos POSIX no Reggy.

Para caracteres alfabéticos, tanto maiúsculos quanto minúsculos, use: `[:alpha:]`

Se você quiser fazer a correspondência de caracteres no intervalo ASCII, escolha: `[:ascii:]`

Obviamente, também há classes POSIX de caracteres negados, no formato: `[:^XXXX:]`

Desse modo, se quisesse fazer a correspondência de caracteres não alfabéticos, você poderia usar: `[:^alpha:]`

Para corresponder a caracteres de espaço e de tabulação, use: `[:space:]`

Ou para corresponder a todos os caracteres brancos, temos o: `[:blank:]`

Há várias dessas classes de caracteres POSIX, as quais se encontram na tabela 5.1.

Tabela 5.1 – Classes de caracteres POSIX

Classe de caracteres	Descrição
----------------------	-----------

<code>::alnum::</code>	Caracteres alfanuméricos (letras e dígitos)
<code>::alpha::</code>	Caracteres alfabéticos (letras)
<code>::ascii::</code>	Caracteres ASCII (todos os 128)
<code>::blank::</code>	Caracteres brancos
<code>::ctrl::</code>	Caracteres de controle
<code>::digit::</code>	Dígitos
<code>::graph::</code>	Caracteres gráficos
<code>::lower::</code>	Letras minúsculas
<code>::print::</code>	Caracteres imprimíveis
<code>::punct::</code>	Caracteres de pontuação
<code>::space::</code>	Espaços em branco
<code>::upper::</code>	Letras maiúsculas
<code>::word::</code>	Caracteres de palavra
<code>::xdigit::</code>	Dígitos hexadecimais

O próximo capítulo é dedicado à correspondência de caracteres Unicode e outros tipos de caracteres.

O que você aprendeu no capítulo 5

- Como criar uma classe ou um conjunto de caracteres usando uma expressão com colchetes.
- Como criar um ou mais intervalos dentro de uma classe de caracteres.
- Como corresponder a números pares no intervalo de 0 a 99.
- Como corresponder a um número hexadecimal.
- Como usar *shorthand* de caracteres dentro de uma classe de

caracteres.

- Como negar uma classe de caracteres.
- Como efetuar união e diferença com classes de caracteres.
- O que são classes de caracteres POSIX.

Notas técnicas

- A aplicação desktop Reggy para Mac pode ser baixada gratuitamente a partir do site <http://www.reggyapp.com>. O Reggy mostra as correspondências efetuadas mudando a cor do texto que foi alvo das correspondências. O padrão é azul, mas você pode alterar essa cor em *Preferences*, acessível pelo menu do Reggy. Em *Preferences*, selecione Java em *Regular Expression Syntax*.
- O navegador Opera Next, atualmente em versão beta, pode ser baixado a partir do site <http://www.opera.com/browser/next/>.
- O Rubular é um editor online de expressões regulares do Ruby, criado por Michael Lovitt, e que suporta as versões 1.8.7 e 1.9.2 do Ruby (ver <http://www.rubular.com>).
- Leia mais a respeito de números pares, dentre os quais o zero está incluído, no site <http://mathworld.wolfram.com/EvenNumber.html>.
- A implementação de expressões regulares em Java (1.6) está documentada no site <http://docs.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html>.
- Você pode encontrar mais informações sobre o IEEE e sua família de padrões POSIX no site <http://www.ieee.org>.

capítulo 6

Correspondendo a caracteres Unicode e outros caracteres

Você terá oportunidades para fazer correspondências de caracteres ou intervalos de caracteres que estão fora do escopo do conjunto ASCII. O ASCII, abreviação de *American Standard Code for Information Interchange*, define um conjunto de caracteres do inglês – as letras de A a Z maiúsculas e minúsculas, mais caracteres de controle e outros. O ASCII tem estado presente há muito tempo: o conjunto de 128 caracteres baseado no latim foi padronizado em 1968. Isso aconteceu muito antes de existir algo como um microcomputador, foi antes do VisiCalc, antes do mouse, antes da Internet, mas eu ainda consulto as tabelas ASCII online regularmente.

Eu me lembro de que, na época em que comecei minha carreira, muitos anos atrás, trabalhei com um engenheiro que guardava uma tabela de códigos ASCII na carteira. Só por garantia. Tabela de Códigos ASCII: não saia de casa sem ela.

Portanto, não vou contestar a importância do ASCII, mas atualmente ele está ultrapassado, especialmente por causa do padrão Unicode (<http://www.unicode.org>), o qual atualmente representa mais de 100 mil caracteres. No entanto, o Unicode não deixa o ASCII para trás; ele incorpora o ASCII em sua tabela de códigos de Latim Básico (ver <http://www.unicode.org/charts/PDF/U0000.pdf>).

Neste capítulo, você sairá dos domínios do ASCII e adentrará no mundo não tão novo assim do Unicode.

O primeiro texto encontra-se no arquivo *voltaire.txt* que está contido no arquivo de códigos e consiste em uma citação de Voltaire (1694–1778), o filósofo iluminista francês.

Qu'est-ce que la tolérance? c'est l'apanage de l'humanité. Nous sommes tous pétris de faiblesses et d'erreurs; pardonnons-nous réciproquement nos sottises, c'est la première loi de la nature.

Aqui está uma tradução para o português: O que é a tolerância? É o

apanágio da humanidade. Somos todos constituídos por fraquezas e erros; perdoarmo-nos reciprocamente nossas tolices é a primeira lei da natureza.

Correspondendo a um caractere Unicode

Há inúmeras maneiras de especificar um caractere Unicode, também conhecido como *code point*. (Para os propósitos deste livro, um caractere Unicode é um caractere que está fora do intervalo ASCII, embora isso não seja estritamente correto.) Comece inserindo a citação de Voltaire no Regexpal (<http://www.regexpal.com>) e depois digite esta expressão regular: `\u00e9`

O `\u` é seguido por um valor hexadecimal `00e9` (não há diferenciação de letras maiúsculas e minúsculas, ou seja, `00E9` funciona igualmente). O valor `00e9` é equivalente ao valor decimal 233, que está bem fora do intervalo ASCII (0–127).

Observe que a letra é (letra e minúscula com acento agudo) fica destacada no Regexpal (ver figura 6.1). Isso acontece porque a letra é é igual ao *code point* U+00E9 em Unicode, com o qual a expressão `\u00e9` efetuou a correspondência.

O Regexpal usa a implementação de expressões regulares do JavaScript. O JavaScript também permite que esta sintaxe seja utilizada: `\xe9`

Experimente usar essa expressão no Regexpal e veja como ela corresponde aos mesmos caracteres correspondidos por `\u00e9`.



Figura 6.1 – Correspondendo a U+00E9 no Regexpal.

Vamos usar essa expressão com uma ferramenta diferente de regex. Acesse o endereço <http://regexhero.net/tester/> com um navegador. O Regex Hero está codificado em .NET e possui uma sintaxe um pouco diferente. Insira o conteúdo do arquivo *basho.txt* na área de texto intitulada *Target String*. Esse arquivo contém um *haiku* famoso escrito pelo poeta japonês Matsuo Basho (que, coincidentemente, faleceu exatamente uma semana antes do nascimento de Voltaire).

Aqui está o poema em japonês: 古池

蛙飛び込む

水の音

—芭蕉 (1644–1694) E aqui está uma tradução para o português: No antigo lago uma rã salta som de água.

—Basho (1644–1694) Para corresponder a uma parte do texto em japonês, digite o seguinte na área de texto intitulada Regular Expression: \u6c60

Este é o *code point* para o caractere japonês (chinês) correspondente

a *lago*. O caractere aparecerá destacado abaixo (ver figura 6.2).

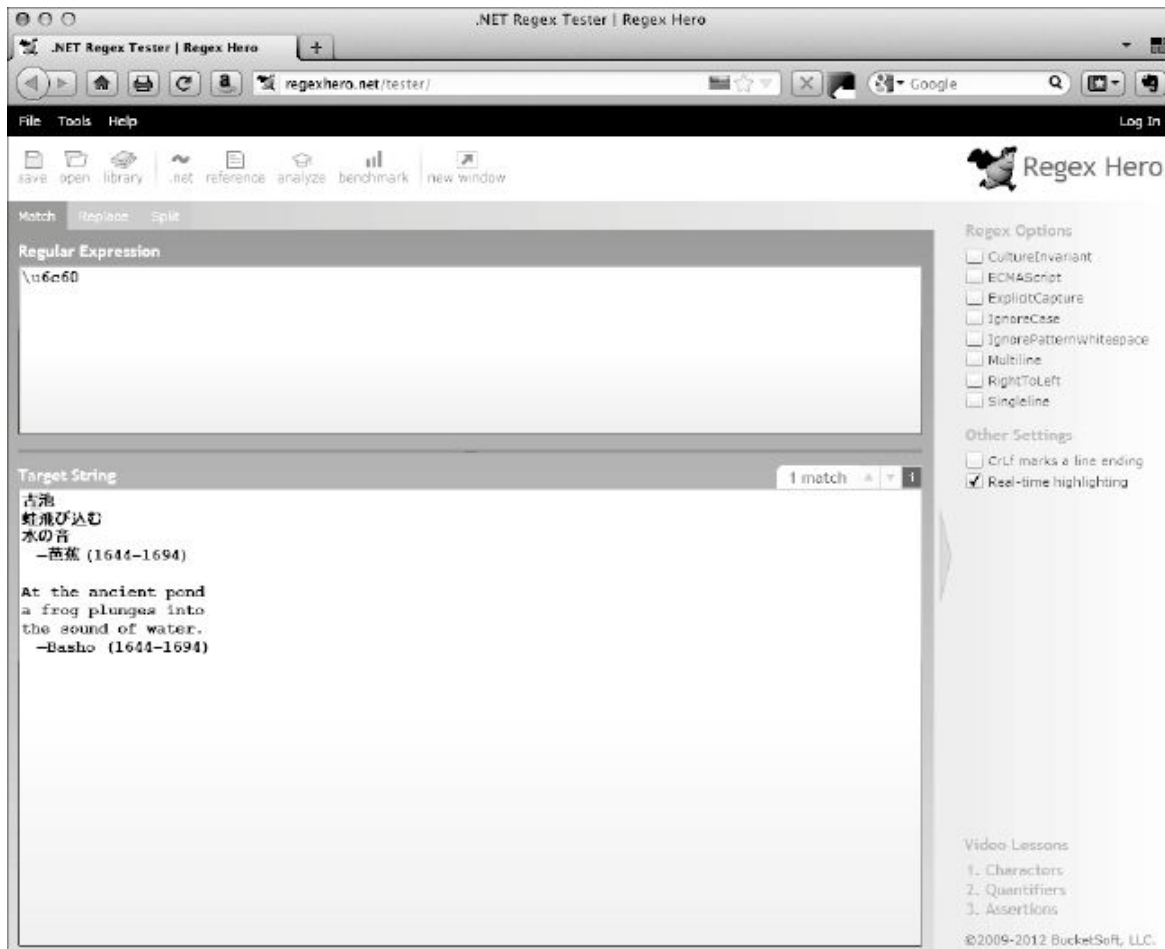


Figura 6.2 – Correspondendo a U+6c60 no Regex Hero.

Aproveitando a ocasião, experimente corresponder ao traço eme (*em dash*) (—) com: \u2014

Ou ao traço ene (*en dash*) (–) com: \u2013

Agora observe esses caracteres em um editor.

Usando vim

Se o *vim* estiver disponível em seu sistema, você pode abrir o arquivo *basho.txt* com esse editor, conforme mostrado abaixo: `vim basho.txt` Agora, começando com uma barra invertida (`\`), inicie uma busca usando esta linha: `^\u6c60`

seguida por Enter ou Return. O cursor se move para o início da correspondência, como você pode ver na figura 6.3.

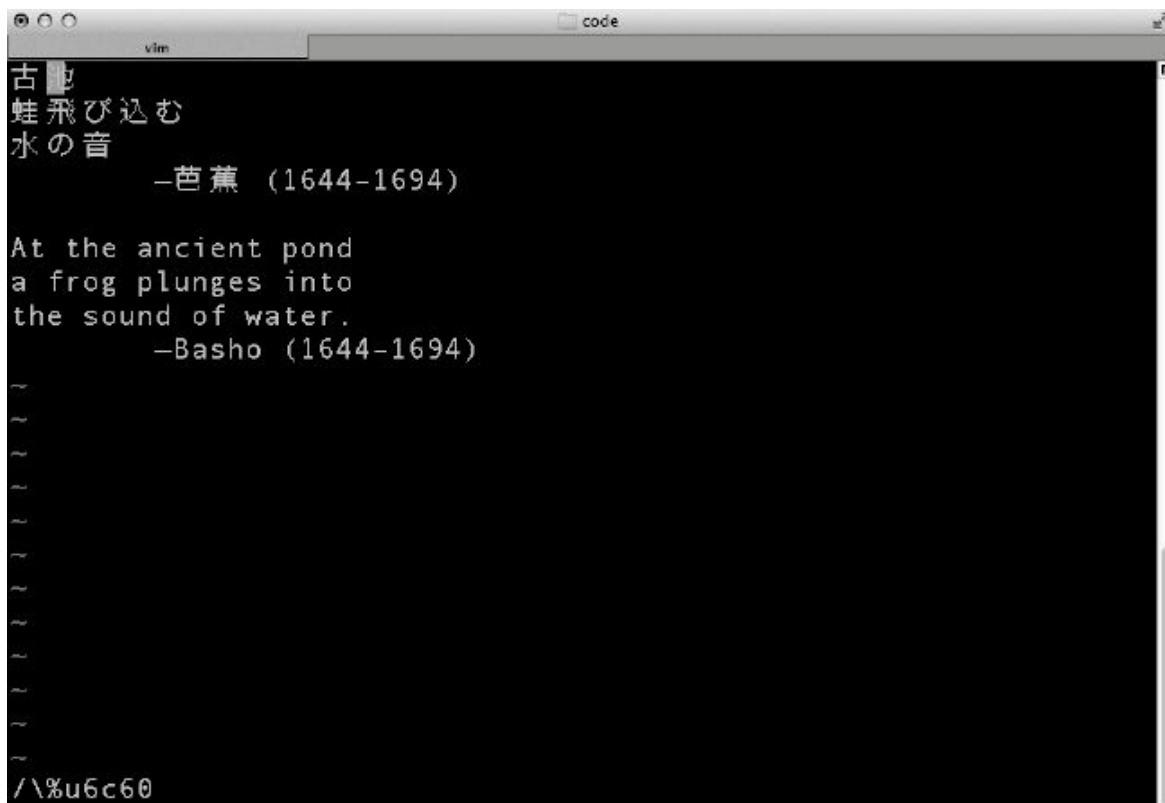


Figura 6.3 – Correspondendo a U+6c60 no Vim.

A tabela 6.1 mostra quais são as suas opções. Você pode usar `x` ou `X` depois de `\%` para corresponder a valores no intervalo de 0–255 (0–FF), `u` para corresponder a até quatro dígitos hexadecimais no intervalo de 256–65.535 (100–FFFF), ou `U` para corresponder a até oito dígitos no intervalo de 65.536–2.147.483.647 (10000–7FFFFFFF). Esse intervalo engloba um bocado de códigos – muito mais do que existe atualmente no Unicode.

Tabela 6.1 – Correspondendo a Unicode no Vim

Primeiro caractere	Máximo de caracteres	Valor máximo
x ou X	2	255 (FF)
u	4	65.535 (FFFF)
U	8	2.147.483.647 (7FFFFFFF)

Correspondendo a caracteres com números octais

Você também pode corresponder a caracteres usando um número octal (base 8), o qual usa os dígitos de 0 a 7. Em uma regex, isso é feito usando-se três dígitos, precedidos por uma barra invertida (\).

Por exemplo, o número octal a seguir: \351

é igual a: \u00e9

Experimente usar essas expressões no Regexpal com o texto de Voltaire. A expressão \351 corresponderá a é, exigindo um pouco menos de digitação.

Correspondendo a propriedades de caracteres Unicode

Em algumas implementações, como, por exemplo, em Perl, você pode fazer correspondências com base em propriedades de caracteres Unicode. As propriedades incluem características como, por exemplo, se o caractere é uma letra, um número ou um sinal de pontuação.

Vou apresentá-lo agora ao *ack*, uma ferramenta de linha de comando escrita em Perl que atua de modo muito semelhante ao *grep* (ver <http://betterthangrep.com>). Essa ferramenta não vem com seu sistema; é preciso baixá-la e instalá-la (dê uma olhada na seção de “Notas técnicas”).

Usaremos o *ack* em um excerto de “An die Freude”, de Friederich Schiller, escrito em 1785 (está em alemão, caso você não reconheça): An die Freude.

Freude, schöner Götterfunken, Tochter aus Elisium, Wir betreten feuertrunken Himmlische, dein Heiligthum.

Deine Zauber binden wieder, was der Mode Schwert getheilt; Bettler werden Fürstenbrüder, wo dein sanfter Flügel weilt.

Seid umschlungen, Millionen!

Diesen Kuß der ganzen Welt!

Brüder, überm Sternenzelt muß ein lieber Vater wohnen.

Há alguns caracteres interessantes nesse excerto que vão além do pequeno universo do ASCII. Olharemos o texto desse poema usando propriedades. (Se você quiser uma tradução desse

fragmento do poema, poderá inseri-lo no Google Translate [<http://translate.google.com>]).

Ao usar o *ack* na linha de comando, você pode especificar que deseja ver todos os caracteres cuja propriedade seja Letter, ou seja, Letra (L): `ack '\pL' schiller.txt` Esse comando mostra todas as letras em destaque. Para letras minúsculas, utilize `l` entre chaves: `ack '\p{l}' schiller.txt` É preciso adicionar as chaves. Para letras maiúsculas, utilize `Lu`: `ack '\p{Lu}' schiller.txt` Para especificar os caracteres que *não* possuem uma propriedade, usamos a letra maiúscula P: `ack '\PL' schiller.txt` Esse comando deixa em destaque os caracteres que não são letras.

O comando a seguir encontra os caracteres que não são letras minúsculas: `ack '\P{l}' schiller.txt` E este comando faz com que as letras que não são maiúsculas fiquem destacadas: `ack '\P{Lu}' schiller.txt` Você também pode executar essas operações em outro testador de regex baseado em web, <http://regex.larsolavtorvik.com>. A figura 6.4 mostra o texto de Schiller com as letras minúsculas em destaque, resultante do uso da propriedade de letra minúscula (`\p{l}`).

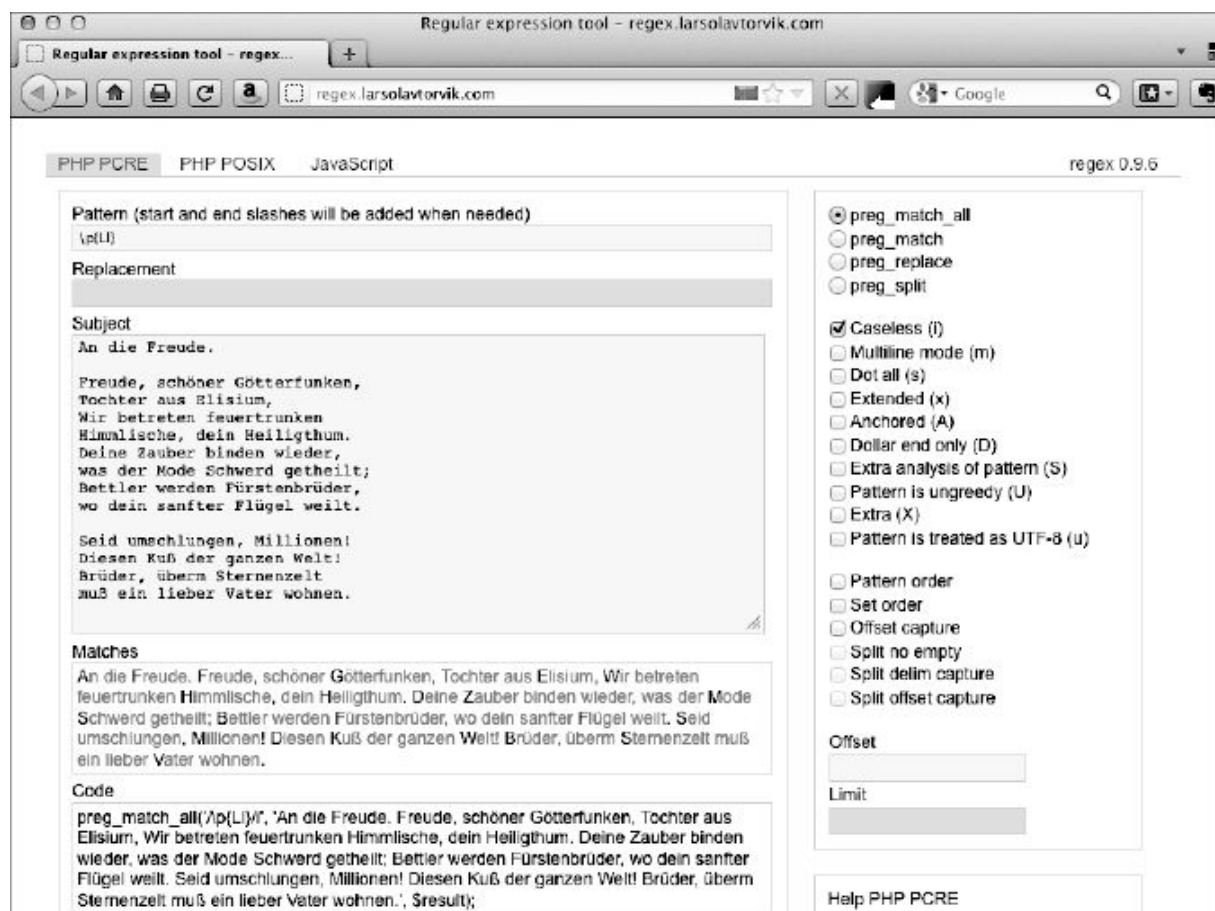


Figura 6.4 – Caracteres com a propriedade de letra minúscula.

A tabela 6.2 lista os nomes das propriedades dos caracteres para uso com `\p{propriedade}` ou com `\P{propriedade}` (ver a seção correspondente a `pcresyntax(3)` em <http://www.pcre.org/pcre.txt>). Você também pode efetuar correspondência de idiomas usando propriedades; ver tabela A.8.

Tabela 6.2 – Propriedades de caracteres

Propriedade	Descrição
C	Outro
Cc	Controle
Cf	Formatação
Cn	Não atribuído

Propriedade	Descrição
Co	Uso privado
Cs	Substituto
L	Letra
LI	Letra minúscula
Lm	Letra de modificação
Lo	Outra letra
Lt	Letra de título
Lu	Letra maiúscula
L&	LI, Lu ou Lt
M	Marca
Mc	Marca de espaçamento
Me	Marca de circunscrição
Mn	Não-marca de espaçamento
N	Número
Nd	Número decimal
NI	Dígito de letra
No	Outro número
P	Pontuação
Pc	Pontuação de conector
Pd	Pontuação de traço

Propriedade	Descrição
Pe	Pontuação de fechamento
Pf	Pontuação final
Pi	Pontuação inicial
Po	Outra pontuação
Ps	Pontuação de abertura
S	Símbolo
Sc	Símbolo de moeda
Sk	Símbolo de modificador
Sm	Símbolo matemático
So	Outro símbolo
Z	Separador
Zl	Separador de linha
Zp	Separador de parágrafo
Zs	Separador de espaço

Correspondendo a caracteres de controle

Como você faz para corresponder a caracteres de controle? Não é muito comum procurar por caracteres de controle em um texto, mas é algo bom para se conhecer. No repositório ou arquivo de exemplos, você encontrará o arquivo *ascii.txt*, que é um arquivo com 128 linhas contendo todos os caracteres ASCII, cada um deles em uma linha separada (por isso, as 128 linhas). Se você executar uma busca no arquivo, uma única linha será retornada caso uma correspondência ocorra. Esse arquivo é bom para efetuar testes e

para divertimento em geral.



Se você fizer buscas de strings ou de caracteres de controle no arquivo *ascii.txt* usando *grep* ou *ack*, essas ferramentas podem interpretar o arquivo como sendo binário. Se isso acontecer, ao executar um script no arquivo, as ferramentas podem simplesmente apresentar uma mensagem que diz “Binary file *ascii.txt* matches” (Houve correspondência no arquivo binário *ascii.txt*) quando uma correspondência for efetuada. É só isso que irá acontecer.

Em expressões regulares, você pode especificar um caractere de controle da seguinte maneira: `\cx` em que *x* é o caractere de controle com o qual você quer fazer a correspondência.

Suponhamos, por exemplo, que você queira encontrar um caractere nulo em um arquivo. Você pode usar Perl para fazer isso, usando o comando a seguir: `perl -n -e 'print if /\c@/' ascii.txt` Desde que o Perl esteja disponível em seu sistema e que esteja funcionando devidamente, você obterá o seguinte resultado: 0. Null O motivo pelo qual isso acontece é que há um caractere nulo nessa linha, apesar de não ser possível ver o caractere no resultado.



Se você abrir o arquivo *ascii.txt* com um editor que não seja o *vim*, é provável que ele vá remover os caracteres de controle do arquivo, portanto, sugiro que você não faça isso.

Você também pode usar `\0` para encontrar um caractere nulo. Experimente usar este comando também: `perl -n -e 'print if ' ascii.txt` Indo um pouco mais além, você pode encontrar o caractere de sino (BEL) usando: `perl -n -e 'print if /\cG/' ascii.txt` Este comando retornará a linha a seguir: 7. Bell Ou você pode usar o *shorthand*: `perl -n -e 'print if ' ascii.txt` Para encontrar o caractere de escape, use: `perl -n -e 'print if /\c[/ ' ascii.txt` que resultará em: 27. Escape Ou faça isso usando um *shorthand*: `perl -n -e 'print if /\e/ ' ascii.txt` E que tal um caractere de backspace? Experimente usar: `perl -n -e 'print if /\cH/ ' ascii.txt` o qual devolverá: 8. Backspace Você também pode encontrar um backspace usando uma expressão com colchetes: `perl -n -e 'print if [' ascii.txt` Sem os colchetes, como o `\b` seria interpretado? É isso mesmo, como uma borda de palavra, conforme você aprendeu no capítulo 2. Os colchetes mudam a maneira pela

qual o `\b` é interpretado pelo processador. Neste caso, o Perl o vê como um caractere de backspace.

A tabela 6.3 lista os modos pelos quais fizemos correspondências de caracteres neste capítulo.

Tabela 6.3 – Correspondendo a caracteres Unicode e a outros caracteres

Código	Descrição
<code>\uxxxx</code>	Unicode (quatro posições)
<code>\xxx</code>	Unicode (duas posições)
<code>\x{xxxx}</code>	Unicode (quatro posições)
<code>\x{xx}</code>	Unicode (duas posições)
<code>\000</code>	Octal (base 8)
<code>\cx</code>	Caractere de controle
<code>\0</code>	Nulo
<code>\a</code>	Sino
<code>\e</code>	Escape
<code>[b]</code>	Backspace

Com isso, encerramos este capítulo. No próximo capítulo, você aprenderá mais sobre quantificadores.

O que você aprendeu no capítulo 6

- Como corresponder a qualquer caractere Unicode usando `\uxxxx` ou `\xxx`.
- Como corresponder a qualquer caractere Unicode no *vim* usando `\%xxx`, `\%XXX`, `\%uxxxx` ou `\%Uxxxx`.
- Como corresponder a caracteres no intervalo de 0–255 usando o formato octal com `\000`.
- Como usar propriedades de caracteres Unicode usando `\p{x}`.

- Como corresponder a caracteres de controle usando `\e` ou `\cH`.
- Mais sobre como usar Perl na linha de comando (mais Perl em uma linha).

Notas técnicas

- Eu inseri os caracteres de controle no arquivo *ascii.txt* usando *vim* (<http://www.vim.org>). No *vim*, você pode usar Ctrl+V seguido pela sequência de controle adequada para o caractere, como, por exemplo, Ctrl+C para o caractere de fim de texto. Eu também usei Ctrl+V seguido de `x` e o código hexadecimal de dois dígitos para o caractere. Você também pode usar dígrafos para inserir os códigos de controle; no *vim*, digite `:digraph` para ver os códigos possíveis. Para inserir um dígrafo, use Ctrl+K no modo Insert, seguido por um dígrafo de dois caracteres (por exemplo, NU para null).
- O RegexHero (<http://regexhero.net/tester>) é uma implementação de regex .NET em um navegador escrito por Steve Wortham. Essa versão é paga, mas você pode testá-la gratuitamente, e se gostar dela, os preços são razoáveis (você pode comprá-la em versão de nível padrão ou profissional).
- O *vim* (<http://www.vim.org>) é uma evolução do editor *vi*, que foi criado por Bill Joy em 1976. O editor *vim* foi desenvolvido principalmente por Bram Moolenaar. Ele parece ser arcaico para os não iniciados, mas, como já mencionei antes, é incrivelmente poderoso.
- A ferramenta *ack* (<http://betterthangrep.com>) está escrita em Perl. Ela atua como o *grep* e possui muitas de suas opções de linha de comando, mas supera o *grep* em diversos aspectos. Por exemplo, o *ack* usa as expressões regulares do Perl, em vez das expressões regulares básicas usadas pelo *grep* (sem `-E`). Para instruções sobre instalação, acesse o site <http://betterthangrep.com/install/>. Eu usei as instruções específicas que se encontram em “Install the ack executable” (Instalar o executável do ack). Não usei o *curl*, mas simplesmente baixei o

ack usando o link disponibilizado e depois copieie o script para o diretório *usrbin*, tanto no meu Mac quanto em um microcomputador, executando Cygwin (<http://www.cygwin.com>) no Windows 7.

capítulo 7

Quantificadores

Você já viu alguns quantificadores em ação anteriormente neste livro, mas, neste capítulo, falarei mais detalhadamente sobre eles.

Desta vez, como exemplo, usaremos uma aplicação desktop para Mac que se chama Reggy (Figura 7.1), do mesmo modo que fizemos no capítulo 5. Desabilite a opção *Match All* na parte inferior para começar.

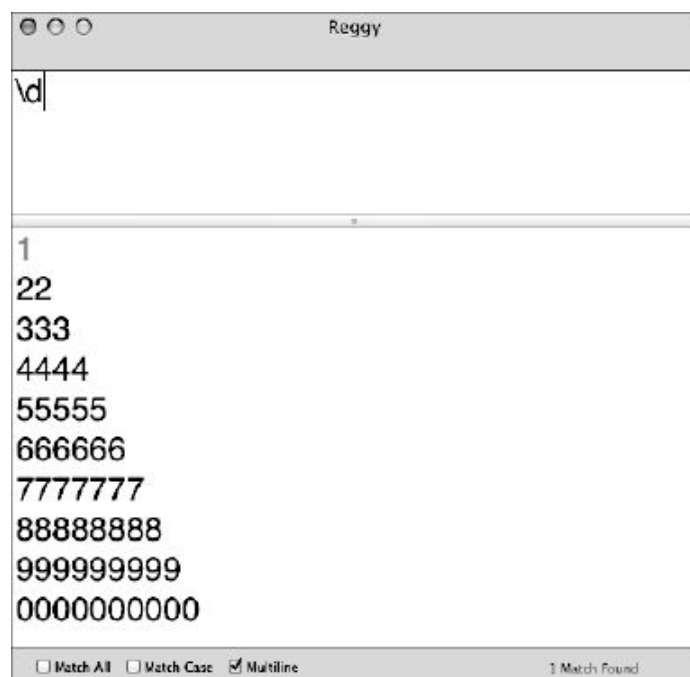


Figura 7.1 – Aplicativo Reggy.

Se você não estiver usando Mac, experimente executar esses exemplos em um dos aplicativos que você já conheceu antes neste livro. Copie e cole o triângulo retângulo de dígitos que está no arquivo *triangle.txt*. Esse arquivo está contido no arquivo de exemplos.

Guloso, preguiçoso e possessivo

Não estou falando de adolescentes aqui. Estou falando de quantificadores. Esses adjetivos podem não soar como boas

qualidades para personalidade, mas constituem recursos interessantes para os quantificadores e você precisa entendê-los se quiser usar expressões regulares de forma habilidosa.

Os quantificadores por si só são gulosos. Um quantificador guloso (*greedy*) tenta primeiro corresponder à string toda. O quantificador engole o máximo possível de caracteres, que é a entrada toda, na tentativa de efetuar uma correspondência. Se a primeira tentativa de corresponder à string toda não for bem-sucedida, volta-se um caractere para trás e a tentativa é efetuada novamente. Esse processo chama-se *backtracking*. Por meio dele, volta-se um caractere de cada vez até que uma correspondência ocorra ou até que não haja mais caracteres para comparar. Um registro do que está ocorrendo também é efetuado, de modo que esse método é o que mais consome recursos se comparado às duas abordagens seguintes. O *backtracking* enche a boca e depois fica cuspidando um pouquinho de cada vez, mastigando o que acabou de comer. Deu para ter uma ideia geral.

Um quantificador preguiçoso (*lazy*, ou às vezes chamado de *relutante*) assume uma postura diferente. Ele começa no início do texto-alvo, tentando encontrar uma correspondência. Ele olha para a string, caractere por caractere, tentando encontrar o que procura. Em último caso, ele tentará corresponder à string toda. Para fazer com que um quantificador seja preguiçoso, é necessário concatenar um ponto de interrogação (?) ao quantificador normal. Esse quantificador mastiga um bocadinho de cada vez.

Um quantificador possessivo (*possessive*) engole o texto-alvo todo e depois tenta encontrar uma correspondência, mas faz somente uma tentativa. Nenhum *backtracking* é efetuado. Um quantificador possessivo concatena um sinal de mais (+) a um quantificador normal. Ele não mastiga; somente engole e depois pensa no que acabou de comer. Vou demonstrar cada um desses quantificadores nas páginas que se seguem.

Correspondendo por meio de *, + e ?

Se você inseriu o triângulo de dígitos no Reggy, pode começar agora a efetuar o teste. Em primeiro lugar, usaremos a estrela de Kleene, que recebeu seu nome em homenagem ao homem a quem se atribui a invenção das expressões regulares, Stephen Kleene. Se você usar a estrela ou o asterisco depois de um ponto, desta maneira: .*

esse quantificador, sendo guloso, corresponderá a todos os caracteres (dígitos) do texto de assunto. Como você já sabe, em razão de leituras anteriores, .* corresponde a qualquer caractere, zero ou mais vezes. Todos os dígitos da caixa de texto inferior devem ficar em destaque, mudando de cor. A respeito da estrela de Kleene, um manual antigo afirmava: Uma expressão regular, seguida por “*” [estrela de Kleene], é uma expressão regular que corresponde a qualquer quantidade (inclusive zero) de ocorrências adjacentes do texto correspondido pela expressão regular.

Agora, experimente usar 9*

e a linha de nove mais para baixo deverá ficar em destaque. A expressão: 9.*

faz com que a linha de nove e a linha de zeros abaixo dela fiquem destacadas. Pelo fato de a opção *Multiline* estar selecionada (na parte inferior da janela do aplicativo), o ponto corresponderá ao caractere de mudança de linha entre as linhas; normalmente, isso não ocorreria.

Para corresponder a um ou mais 9s, experimente usar: 9+

De que modo essa expressão é diferente? Não é possível saber realmente, pois há nove 9s no texto de assunto. A principal diferença é que o + procura por no mínimo um 9, mas o * procura por zero ou mais.

Para fazer a correspondência zero ou uma vez (opcional), use 9?

Esta expressão corresponderá somente à primeira ocorrência de 9. Esse 9 é considerado opcional, mas, como ele está presente no texto de assunto, a correspondência é feita e o número fica em destaque. Se você usar: 99?

então a correspondência será efetuada tanto com o primeiro quanto com o segundo 9.

A tabela 7.1 lista os quantificadores básicos e algumas das possibilidades que esses quantificadores permitem. Por padrão, os quantificadores são *gulosos*, o que significa que eles correspondem ao máximo de caracteres possíveis na primeira tentativa.

Tabela 7.1 – Quantificadores básicos

Sintaxe	Descrição
?	Zero ou um (opcional)
+	Um ou mais
*	Zero ou mais

Efetuando a correspondência um número específico de vezes

Quando as chaves são usadas, você pode fazer a correspondência de um padrão um número específico de vezes em um intervalo. Se esses quantificadores não forem modificados, é sinal de que eles são *gulosos*. Por exemplo: 7{1}

corresponderá à primeira ocorrência de 7. Se você quiser corresponder a uma ou *mais* ocorrências do número 7, tudo o que você precisa fazer é adicionar uma vírgula: 7{1,}

Provavelmente, você percebeu que: 7+

e

7{1,}

são essencialmente a mesma coisa, e que: 7*

e

7{0,}

são também o mesmo. Além do mais, 7?

é o mesmo que $7\{0,1\}$

Para encontrar um intervalo de correspondências, ou seja, para corresponder de m a n vezes, use algo como: $7\{3,5\}$

Essa expressão corresponderá a três, quatro ou cinco ocorrências de 7.

Então, para recordar, as chaves ou a sintaxe para intervalo de correspondências é o quantificador mais flexível e mais preciso que existe. A tabela 7.2 resume essas características.

Tabela 7.2 – Resumo da sintaxe de intervalo

Sintaxe	Descrição
$\{n\}$	Corresponder exatamente n vezes
$\{n,\}$	Corresponder n ou mais vezes
$\{m,n\}$	Corresponder de m a n vezes
$\{0,1\}$	Mesmo que ? (zero ou um)
$\{1,0\}$	Mesmo que + (um ou mais)
$\{0,\}$	Mesmo que * (zero ou mais)

Quantificadores preguiçosos

Agora vamos deixar de lado a gulodice e vamos nos concentrar na preguiça. A maneira mais fácil de compreendê-la é vê-la em ação. No Reggy (certifique-se de que a opção *Match All* não esteja selecionada), tente corresponder a zero ou a um 5 usando um único ponto de interrogação (?): 5?

O primeiro 5 fica em destaque. Adicione um ? adicional para tornar o quantificador preguiçoso: 5??

Agora parece que essa expressão não corresponde a nada. O motivo pelo qual isso acontece é que o padrão está sendo preguiçoso, ou seja, ele não é forçado a corresponder nem ao

primeiro 5. Devido a sua natureza, uma correspondência *preguiçosa* corresponde ao mínimo de caracteres com os quais consegue se resolver. Ela é indolente.

Experimente usar zero ou mais vezes: 5^* ?

e nesse caso também não haverá correspondência porque você deu a opção de corresponder no mínimo zero vezes, e é isso o que foi feito.

Tente de novo, fazendo a correspondência uma ou mais vezes no modo preguiçoso: $5^+?$

E agora temos um resultado. O preguiçoso acaba de sair do sofá e correspondeu a um 5. É tudo o que ele teve de fazer para considerar seu trabalho realizado.

As coisas começam a ficar um pouco mais interessantes quando você aplica a correspondência m,n . Experimente usar: $5\{2,5\}^?$

Há correspondência somente de dois 5, e não de todas as cinco ocorrências, como seria feito com um quantificador guloso.

A tabela 7.3 lista os quantificadores preguiçosos. Em que ocasiões a correspondência preguiçosa seria útil? Você pode usar a correspondência preguiçosa quando quiser corresponder estritamente ao mínimo de caracteres necessários, e não ao máximo possível.

Tabela 7.3 – Quantificadores preguiçosos

Sintaxe	Descrição
??	Zero ou um (opcional) preguiçoso
+?	Um ou mais preguiçoso
*?	Zero ou mais preguiçoso
{n}?	n preguiçoso
{n,}?	n ou mais preguiçoso
{m,n}?	m,n preguiçoso

Quantificadores possessivos

Uma correspondência possessiva é como uma correspondência gulosa: ela engole o máximo de caracteres possíveis. Mas, diferentemente de uma correspondência gulosa, a correspondência possessiva não faz backtracking. Ela não abre mão de nada que encontra. Ela é egoísta. É por isso que ela se chama *possessiva*. Com braços firmemente cruzados, ela não cede nenhum terreno. Mas a boa notícia a respeito dos quantificadores possessivos é que eles são rápidos, pois não efetuam backtracking, e também falham rapidamente.



O fato é que você dificilmente conseguirá perceber qualquer diferença entre as correspondências gulosas, preguiçosas ou possessivas com os exemplos deste livro. Mas, à medida que se tornar mais experiente e que o ajuste de performance vier a ser mais importante, você vai querer ter ciência dessas diferenças.

Para esclarecer isso, primeiro, procuraremos corresponder aos zeros com um zero no início e, depois, com um zero no fim. No Reggy, certifique-se de que a opção *Match All* esteja selecionada e digite esta expressão com um zero no início: `0.*+`

O que aconteceu? Todos os zeros ficaram destacados. Houve uma correspondência. A correspondência possessiva parece fazer a mesma coisa que a correspondência gulosa, com uma diferença sutil: não há backtracking. Você agora pode fazer a prova. Digite isto, com um zero no final: `.*+0`

Não há correspondência. Isso acontece porque não houve backtracking. A expressão engoliu toda a entrada e nunca mais olhou para trás. Ela gastou sua herança com uma vida desregrada. Ela não consegue encontrar o zero no final. Ela não sabe para onde deve olhar. Se você remover o sinal de mais, ela encontrará todos os zeros, pois ela volta a ser uma correspondência gulosa.

`.*0`

Você pode querer usar um quantificador possessivo quando estiver ciente do que há em seu texto e sabe em quais lugares as correspondências serão encontradas. Você não se importa com o

fato de o quantificador engolir com gosto. Uma correspondência possessiva pode ajudá-lo a fazer correspondências com um desempenho melhor. A tabela 7.4 mostra os quantificadores possessivos.

Tabela 7.4 – Quantificadores possessivos

Sintaxe	Descrição
?+	Zero ou um (opcional) possessivo
++	Um ou mais possessivo
*+	Zero ou mais possessivo
{n}+	<i>n</i> possessivo
{n,}+	<i>n</i> ou mais possessivo
{m,n}+	<i>m,n</i> possessivo

No próximo capítulo, você vai conhecer os lookarounds.

O que você aprendeu no capítulo 7

- As diferenças entre correspondências gulosas (*greedy*), preguiçosas (*lazy*) e possessivas (*possessive*).
- Como corresponder uma ou mais vezes (+).
- Como corresponder opcionalmente (zero ou uma vez, ?).
- Como corresponder zero ou uma vez (*).
- Como usar quantificadores {*m,n*}.
- Como usar quantificadores gulosos, preguiçosos (relutantes) e possessivos.

Notas técnicas

A citação (original em inglês) é de Dennis Ritchie e Ken Thompson, *QED Text Editor* (Murray Hill, NJ, Bell Labs, 1970), p. 3 (ver <http://cm.bell-labs.com/cm/cs/who/dmr/qedman.pdf>).

capítulo 8

Lookarounds

Lookarounds são grupos de não captura que fazem correspondência de padrões com base no que encontram na frente ou atrás de um padrão. Os lookarounds também são considerados *asserções de largura zero*.

Os lookarounds incluem: • lookaheads positivos; • lookaheads negativos; • lookbehinds positivos; • lookbehinds negativos.

Neste capítulo, mostrarei como cada um deles funciona. Começaremos usando RegExr no desktop e depois continuaremos com Perl e *ack* (o *grep* não conhece lookarounds).

Nosso texto continuará sendo o velho poema de Coleridge.

Lookaheads positivos

Suponha que você queira encontrar todas as ocorrências da palavra *ancyent* que vem seguida pela palavra *marinere* (estou usando a ortografia arcaica porque é assim que a palavra se encontra no arquivo). Para fazer isso, podemos usar um lookahead positivo.

Em primeiro lugar, vamos experimentar isso no RegExr desktop. O padrão a seguir, que não diferencia maiúsculas de minúsculas, deve ser inserido na caixa de texto superior: `(?i)ancyent (?=marinere)`



Você também pode especificar se a correspondência será sensível ao caso no RegExr simplesmente habilitando a caixa de seleção ao lado de *ignoreCase*, mas ambos os métodos funcionam.

Como você está usando a opção `(?i)` para não diferenciar letras maiúsculas de minúsculas, não é preciso se preocupar com a forma usada (maiúsculas ou minúsculas) em seu padrão. Você está à procura de todas as linhas que contenham a palavra *ancyent* seguida imediatamente pela palavra *marinere*. Os resultados aparecerão em destaque na caixa de texto abaixo da caixa na qual está o padrão (ver figura 8.1); no entanto, somente a primeira parte do padrão

ficará em destaque (*ancyent*), e não o padrão de lookahead (*Marinere*).

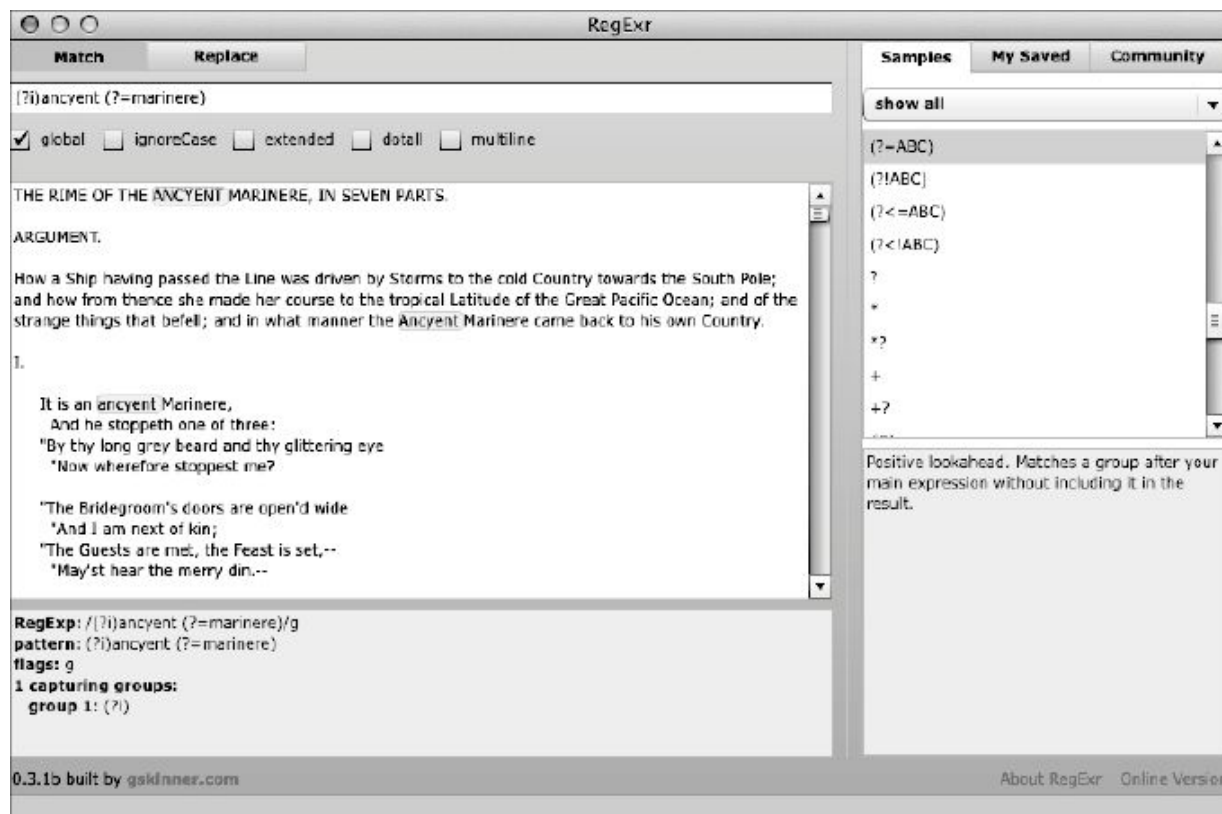


Figura 8.1 – Lookahead positivo no RegExr.

Vamos agora usar o Perl para fazer um lookahead positivo. Você pode construir o comando da seguinte maneira: perl -ne 'print if (?i)ancyent (?=marinere)' rime.txt e a saída deverá ser algo do tipo: THE RIME OF THE ANCYENT MARINERE, IN SEVEN PARTS.

How a Ship having passed the Line was driven by Storms to the cold Country towards the South Pole; and how from thence she made her course to the tropical Latitude of the Great Pacific Ocean; and of the strange things that befell; and in what manner the Ancyent Marinere came back to his own Country.

It is an ancyent Marinere, "God save thee, ancyent Marinere!

"I fear thee, ancyent Marinere!

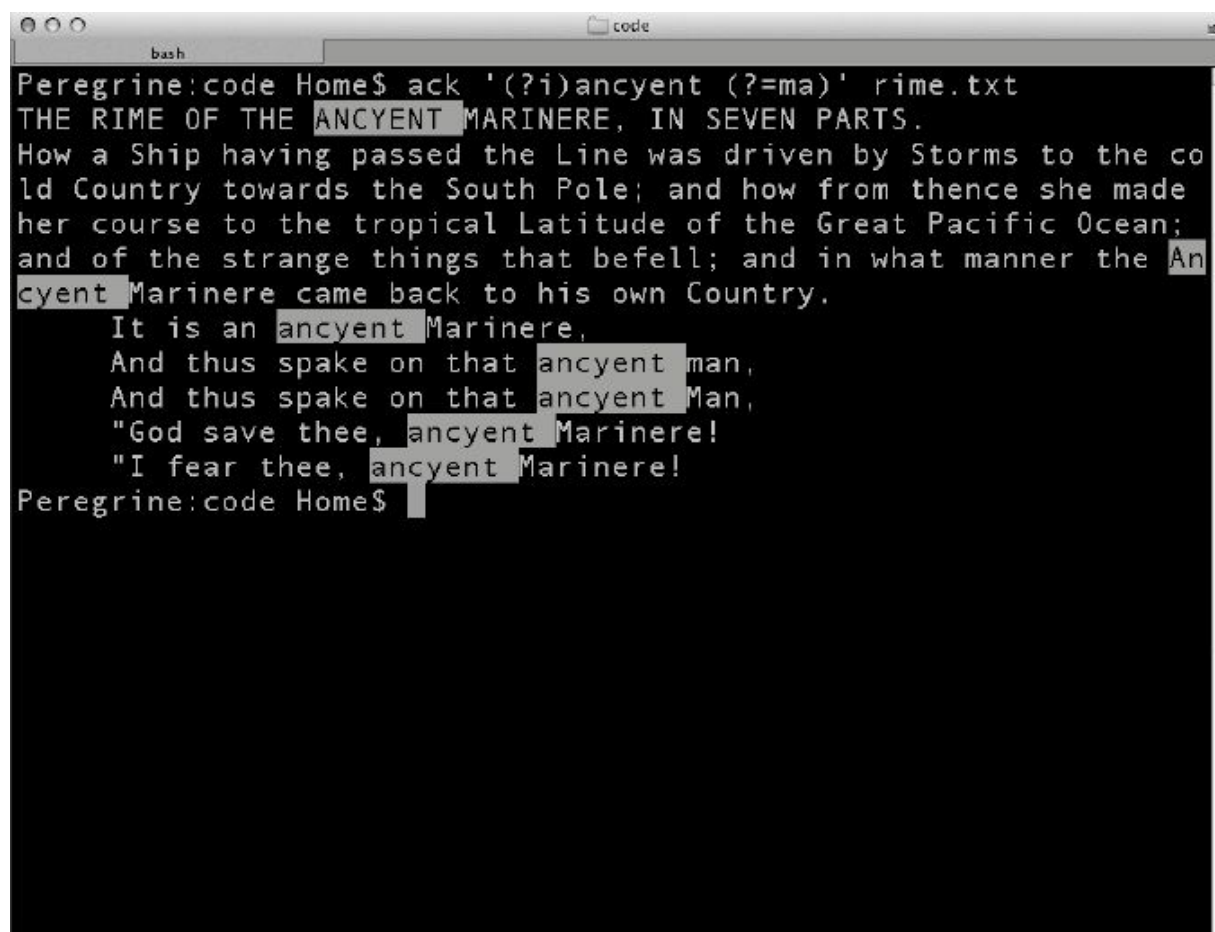
Existem cinco linhas no poema nas quais a palavra *ancyent* aparece imediatamente antes da palavra *marinere*. E se quiséssemos apenas verificar se a palavra que vem depois de *ancyent* começa com a letra *m*, independentemente de ser maiúsculo ou minúsculo? Poderíamos fazer isso desta maneira: perl -ne 'print if (?i)ancyent (?=m)' rime.txt Além de *Marinere*, teremos também *man* e *Man*: And thus spake on

that ancyent man, And thus spake on that ancyent Man, O *ack* também consegue fazer lookarounds, pois está escrito em Perl. A interface de linha de comando do *ack* é muito semelhante à do *grep*.

Experimente usar:

`ack '(?i)ancyent (?=ma)' rime.txt` e você verá resultados em destaque, como mostrado na figura 8.2.

Com o *ack*, você pode especificar se haverá diferenciação entre letras maiúsculas e minúsculas usando a opção de linha de comando `-i`, em vez de usar a opção embutida `(?i)`: `ack -i 'ancyent (?=ma)' rime.txt` Vou acrescentar algumas ideias aqui. Se você quiser adicionar números de linha na saída do *ack*, há várias maneiras de fazer isso. Você pode adicionar a opção `-H`: `ack -Hi 'ancyent (?=ma)' rime.txt` Ou você poderia adicionar este código com a opção `--output`: `ack -i --output '$.:$_' 'ancyent (?=ma)' rime.txt` Não é muito elegante, e o destaque é desabilitado, mas funciona.



```
Peregrine:code Home$ ack '(?i)ancyent (?=ma)' rime.txt
THE RIME OF THE ANCYENT MARINERE, IN SEVEN PARTS.
How a Ship having passed the Line was driven by Storms to the co
ld Country towards the South Pole; and how from thence she made
her course to the tropical Latitude of the Great Pacific Ocean;
and of the strange things that befell; and in what manner the An
cyent Marinere came back to his own Country.
    It is an ancyent Marinere,
    And thus spake on that ancyent man,
    And thus spake on that ancyent Man,
    "God save thee, ancyent Marinere!
    "I fear thee, ancyent Marinere!
Peregrine:code Home$
```

Figura 8.2 – Lookahead positivo usando ack no Terminal.

Lookaheads negativos

O inverso de um lookahead positivo é um lookahead negativo. Isso significa que, quando tentar corresponder a um padrão, você *não* encontrará um dado padrão de lookahead. Um lookahead negativo é constituído da seguinte maneira: `(?!i)ancyent (?!marinere)` Somente um caractere mudou: o sinal de igual (=) do lookahead positivo tornou-se um ponto de exclamação (!) no lookahead negativo. A figura 8.3 mostra esse lookahead negativo no navegador Opera.

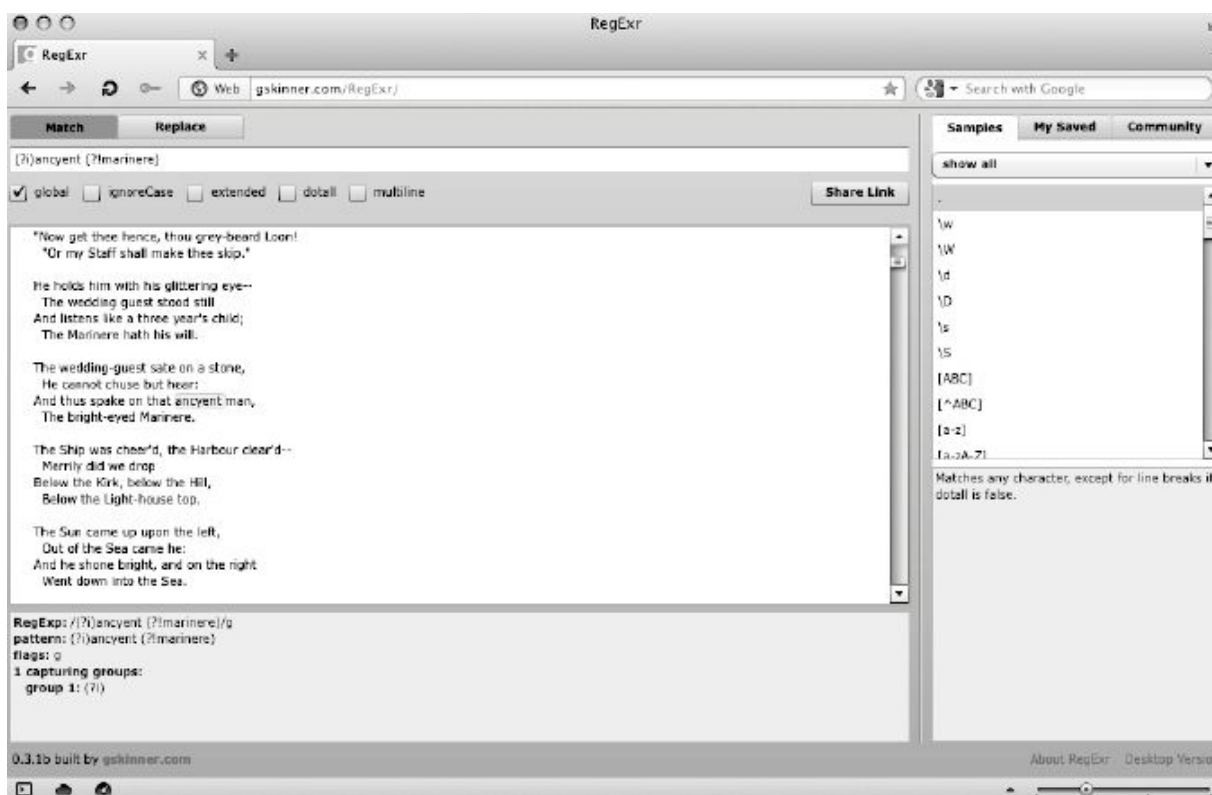


Figura 8.3 – Lookahead negativo no RegExr no navegador Opera.

Em Perl, podemos fazer um lookahead negativo desta maneira: `perl -ne 'print if (?!i)ancyent (?!marinere)'` rime.txt e a saída esperada é: `And thus spake on that ancyent man, And thus spake on that ancyent Man, No ack, os mesmos resultados podem ser produzidos usando: ack -i 'ancyent (?!marinere)' rime.txt`

Lookbehinds positivos

Um lookbehind positivo olha para a esquerda, na direção oposta a um lookahead. A sintaxe é: `(?i)(?<=ancyent) marinere` O lookbehind positivo utiliza um sinal de menor (`<`) para lembrá-lo da direção para a qual o lookbehind está olhando. Experimente usar essa expressão no RegExr e observe a diferença. Em vez de destacar a palavra *ancyent*, a palavra *marinere* é que será destacada. Por quê? Porque o lookbehind positivo é uma condição para a correspondência e não será incluída ou consumida nos resultados correspondidos.

Faça o seguinte com Perl: `perl -ne 'print if (?i)(?<=ancyent) marinere' rime.txt` E isto no `ack`: `ack -i '(?<=ancyent) marinere' rime.txt`

Lookbehinds negativos

Por último, temos o lookbehind negativo. E como você acha que isso funciona?

Ele olha para ver se um padrão *não* aparece antes, na cadeia de texto da esquerda para a direita. Novamente, esse lookbehind adiciona um sinal de menor (`<`) para lembrá-lo da direção do lookbehind.

Experimente usar isto no RegExr e observe os resultados: `(?1)(?<!ancyent) marinere` Faça a rolagem no texto para ver o resultado obtido.

Depois experimente usar esta linha no Perl: `perl -ne 'print if (?i)(?<!ancyent) marinere' rime.txt` Você deverá ver este resultado, sem sinal da palavra *ancyent* em lugar nenhum: The Marinere hath his will.

The bright-eyed Marinere.

The bright-eyed Marinere.

The Mariners gave it biscuit-worms, Came to the Marinere's hollo!

Came to the Marinere's hollo!

The Mariners all 'gan work the ropes, The Mariners all return'd to work The Mariners all 'gan pull the ropes, "When the Marinere's trance is abated."

He loves to talk with Mariners The Marinere, whose eye is bright, E, por último, dê este comando no `ack`: `ack -i '(?<!ancyent) marinere' rime.txt` Com isso, terminamos nossa breve introdução aos lookaheads e lookbehinds, um recurso poderoso das expressões regulares

modernas.

No próximo capítulo, você verá um exemplo completo de como inserir marcação HTML5 em um documento usando *sed* e Perl.

O que você aprendeu no capítulo 8

- Como usar lookaheads positivos e negativos.
- Como usar lookbehinds positivos e negativos

Notas técnicas

Consulte também as páginas 59 a 66 de *Mastering Regular Expressions*, 3ª edição (<http://shop.oreilly.com/product/9780596528126.do>).

capítulo 9

Inserindo marcação HTML em um documento

Este capítulo o conduzirá, passo a passo, pelo processo de inserir marcação HTML5 em um texto comum usando expressões regulares, concluindo assim o que havíamos começado anteriormente neste livro.

Agora, se dependesse de mim, eu usaria o AsciiDoc para fazer esse trabalho. Porém, em virtude dos nossos objetivos aqui, fingiremos que não existe algo como o AsciiDoc (que pena!). Trabalharemos com algumas ferramentas que temos à mão – ou seja, *sed* e Perl – e com nossa própria engenhosidade.

Como texto, ainda continuaremos a usar o poema de Coleridge que se encontra no arquivo *rime.txt*.



Os scripts usados neste capítulo funcionam bem com o arquivo *rime.txt* porque você compreende a estrutura desse arquivo. Esses scripts fornecerão resultados menos previsíveis quando usados com arquivos de texto arbitrários; no entanto, eles oferecem um ponto de partida para lidar com estruturas de texto em arquivos mais complexos.

Correspondendo a tags

Antes de começar a inserir marcações no poema, vamos conversar a respeito de como corresponder a tags HTML ou XML. Há inúmeras maneiras de fazer correspondências de tag, sejam tags iniciais (como `<html>`) ou tags finais (como `</html>`), mas eu acho que a expressão a seguir é confiável. Ela corresponde a tags iniciais, com ou sem atributos: `<[_a-zA-Z][^>]*>` Eis o que faz esta expressão: • O primeiro caractere é um sinal de menor (`<`).

- Os elementos podem começar com um underscore (`_`) em XML ou com uma letra pertencente ao conjunto ASCII, maiúscula ou minúscula (ver seção de “Notas técnicas”).
- Após o caractere de início, o nome pode ser seguido por zero ou por mais caracteres, qualquer caractere que não seja um sinal de

maior (>).

- A expressão termina com um sinal de maior.

Experimente usar o comando abaixo no *grep*. Execute-o usando um arquivo DITA de exemplo, que se encontra em *lorem.dita*: `grep -Eo '<[_a-zA-Z][^>]*>' lorem.dita` e o resultado deverá ser: `<topic id="lorem"> <title> <body> <p> <p> <p> <p>` Para corresponder tanto a tags iniciais quanto finais, basta adicionar uma barra para frente seguida por um ponto de interrogação. O ponto de interrogação faz com que a barra para frente seja opcional: `</?[_a-zA-Z][^>]*>` Estou me atendo somente a tags iniciais aqui. Para tornar a saída mais refinada, eu geralmente faço o pipe dela para algumas ferramentas adicionais, de modo a torná-la mais elegante: `grep -Eo '<[_a-zA-Z][^>]*>' lorem.dita | sort | uniq | sed 's/^<\/;/s/id=\".*\"\/;/s/> $/'`

Este comando resulta em uma lista de nomes de tags XML ordenadas: `body li`

`p`

`p`

`title`

`topic`

`ul`

Vou explicar melhor esse comando no próximo e último capítulo. As seções a seguir o conduzirão ao longo de alguns passos que você já aprendeu, mas que serão usados com pequenas variações.

Transformando texto normal usando sed

Vamos inserir algumas marcações na parte superior do texto que está no arquivo *rime.txt*. Podemos fazer isso usando o comando de inserção (`i\`). No diretório em que o arquivo *rime.txt* está localizado, digite o seguinte comando em um prompt de shell: `sed '1 i\ <!DOCTYPE html>\ <html lang="en">\ <head>\ <title>The Rime of the Ancyent Marinere (1798)</title>\ <meta charset="utf-8"/>\ </head>\ <body>\ q' rime.txt` Depois de teclar Enter ou Return, sua saída deverá ter a aparência a seguir, com as tags na parte superior: `<!DOCTYPE html> <html lang="en"> <head> <title>The`

```
Rime of the Ancyent Marinere (1798)</title> <meta charset="utf-8"/>
</head> <body> THE RIME OF THE ANCYENT MARINERE, IN
SEVEN PARTS.
```

O comando que você acabou de executar não muda efetivamente o arquivo – somente produz uma saída na tela. Mais tarde, mostrarei como fazer com que suas alterações sejam escritas em um arquivo.

Substituição usando sed

No próximo exemplo, o *sed* encontrará a primeira linha do arquivo e capturará a linha toda em um grupo de captura usando parênteses escapados, `\(` e `\)`. O *sed* precisa escapar os parênteses usados para capturar um grupo, a menos que você use a opção `-E` (você verá mais sobre isso em breve). O início da linha é demarcado com `^`, e o final da linha, com `$`. O retrovisor `\1` insere o texto capturado como conteúdo do elemento *title*, fazendo a endentação com um espaço.

Execute o comando que se segue: `sed '1s/^\(.*\)$/<title>\1</title>;q' rime.txt` A linha resultante será: `<title>THE RIME OF THE ANCYENT MARINERE, IN SEVEN PARTS.</title>` Agora, experimente executar desta maneira: `sed -E '1s/^\(.*\)$/<!DOCTYPE html>\ <html lang="en">\ <head>\ <title>\1</title>\ </head>\ <body>\ <h1>\1</h1>\ /;q' rime.txt` Vamos conversar sobre esse comando:

- A opção `-E` diz ao *sed* para usar as expressões regulares estendidas ou EREs (para que você não precise escapar os parênteses, etc.).

- Usando um comando de substituição (*s*), armazene a linha 1 em um grupo de captura (`^\(.*\)`) para poder reutilizar o texto usando `\1`.
- Crie tags HTML e escape as mudanças de linha com `\`.
- Insira o texto capturado nas tags *title* e *h1* usando `\1`.
- Saia neste momento (*q*) para interromper a apresentação do restante do poema na tela.

O resultado correto é: `<!DOCTYPE html> <html lang="en"> <head> <title>THE RIME OF THE ANCYENT MARINERE, IN SEVEN PARTS.</title> </head> <body> <h1>THE RIME OF THE ANCYENT MARINERE, IN SEVEN PARTS.</h1>`

Lidando com algarismos romanos no sed

O poema está dividido em sete seções, com cada seção sendo introduzida por um algarismo romano. Há também um cabeçalho denominado “ARGUMENT”. A linha a seguir usa o *sed* para capturar esse cabeçalho e os algarismos romanos e colocar tags *h2* ao redor: `sed -En 's/^(ARGUMENT\.|I{0,3}V?I{0,2}\.)$/<h2>\1</h2>/p' rime.txt` e aqui está o que você vai ver: `<h2>ARGUMENT\.</h2> <h2>I.</h2>`

`<h2>II.</h2>`

`<h2>III.</h2>`

`<h2>IV.</h2>`

`<h2>V.</h2>`

`<h2>VI.</h2>`

`<h2>VII.</h2>`

A seguir, uma descrição do comando *sed* apresentado acima:

- A opção `-E` disponibiliza as expressões regulares estendidas, e a opção `-n` suprime a apresentação de cada uma das linhas, que corresponderia ao comportamento padrão do *sed*.

- O comando de substituição (`s`) captura o cabeçalho e os sete algarismos romanos em letras maiúsculas, cada um deles em linhas separadas e seguidos por um ponto, no intervalo de I a VII.
- Em seguida, o comando `s` pega cada linha do texto capturado e a insere entre elementos *h2*.
- A flag `p` no final do comando de substituição mostra o resultado na tela.

Lidando com um parágrafo específico usando sed

A seguir, esta linha encontra um parágrafo na linha 5: `sed -En '5s/^([A-Z].*)$/<p>\1</p>/p' rime.txt` e coloca esse parágrafo em uma tag *p*: `<p>How a Ship having passed the Line was driven by Storms to the cold Country towards the South Pole; and how from thence she made her course to the tropical Latitude of the Great Pacific Ocean; and of the strange things that befell; and in what manner the Ancyent Marinere came back to his own Country.</p>`
Eu sei que, no momento, parece que estamos fazendo alterações muito pequenas a cada vez, mas tenha um pouco de paciência e

reuniremos tudo isso nas próximas páginas.

Lidando com as linhas do poema usando sed

A seguir, faremos a marcação das linhas do poema usando: sed -E '9s/^[\]*(.*)/ <p>\1<brV>/;10,832s/^[\]{5,7}.*)/\1<brV>/;833s/^(.*)/\1<Vp>/' rime.txt Essas substituições do *sed* dependem dos números das linhas para poder efetuar seu trabalhinho. Esse comando não funcionaria em um caso geral, mas funciona muito bem quando você sabe exatamente com o que está lidando.

- Na linha 9, a primeira linha contendo versos, o comando *s* pega a linha e, depois de colocar alguns espaços antes, insere uma tag inicial *p* e concatena uma tag *br* (quebra) no final da linha.
- Entre as linhas 10 e 832, toda linha que começa com 5 a 7 espaços em branco adquire um *br* concatenado.
- Na linha 833, a última linha do poema, em vez de um *br*, o *s* concatena uma tag *p* final.

Aqui está uma amostra da marcação resultante: <p>It is an ancyent Marinere,
 And he stoppeth one of three:
 "By thy long grey beard and thy glittering eye
 "Now wherefore stoppest me?
 "The Bridegroom's doors are open'd wide
 "And I am next of kin;
 "The Guests are met, the Feast is set,--
 "May'st hear the merry din.--
 Você também deveria substituir as linhas em branco com um *br*, para manter os versos separados: sed -E 's/^\$/<brV>/' rime.txt Observe o que você acabou de fazer: He prayeth best who loveth best, All things both great and small: For the dear God, who loveth us, He made and loveth all.

 The Marinere, whose eye is bright, Whose beard with age is hoar, Is gone; and now the wedding-guest Turn'd from the bridegroom's door.

 He went, like one that hath been stunn'd And is of sense forlorn: A sadder and a wiser man He rose the morrow morn.

Eu descobri que posso brincar infinitamente com esse tipo de coisa, inserindo tags e espaços exatamente nos lugares em que eu quero. Eu o incentivo a fazer isso sozinho.

Anexando tags

Agora faremos a anexação de algumas tags no final do poema. Usando o comando de anexação (`a\`), o `$` encontra o final (a última linha) do arquivo e anexa (`a\`) as tags `body` e `html` finais depois da última linha: `sed '$ a\ <\body>\ <\html>\ ' rime.txt` Aqui está como ficará o final do arquivo agora: He went, like one that hath been stunn'd And is of sense forlorn: A sadder and a wiser man He rose the morrow morn.

`</body> </html>` Chega de `sed`.

E se você quisesse fazer todas essas alterações ao mesmo tempo? Você sabe o que deve ser feito. Você já fez isso. Basta colocar todos esses comandos em um arquivo e usar a opção `-f` com o `sed`.

Usando um arquivo de comandos com o sed

Este exemplo apresenta o arquivo `html.sed`, que reúne todos os comandos `sed` anteriores em um arquivo, além de incluir um ou outro comando adicional. Usaremos esse arquivo de comandos para transformar o arquivo `rime.txt` em HTML usando `sed`. As numerações no exemplo irão orientá-lo em relação ao que acontece no script do `sed`.

`#!/usrbin/sed` ❶

`1s/^(.*)$/<!DOCTYPE html>\` ❷

`<html lang="en">\ <head>\ <title>\1<\title>\ <\head>\ <body>\ <h1>\1<\h1>\ /`

`s/^(ARGUMENT|{0,3}V?!{0,2})\.$/<h2>\1<\h2>/` ❸

`5s/^[A-Z].*)$/<p>\1<\p>/` ❹

`9s/^[]*(.*)/ <p>\1
/` ❺

`10,832s/^[]{5,7}.*)\1
/` ❻

`833s/^(.*)\1<\p>/` ❼

`13,$s/^$/
/` ❽

`$ a\` ❾

`<\body>\ <\html>\` ❶ A primeira linha chama-se linha *shebang*, uma dica para o shell saber em que local se encontra o executável (do `sed`).

- ❷ Na linha 1, substitua (`s`) a linha com as tags que se seguem. A barra invertida (`\`) indica que o texto que você quer adicionar continua na próxima linha, de modo que uma mudança de linha é inserida. Insira o título do poema que está na linha 1 usando `\1`

como sendo o conteúdo dos elementos *title* e *h1*.

- ③ Coloque tags *h2* ao redor de cabeçalhos e algarismos romanos.
- ④ Na linha 5, coloque um elemento *p* ao redor do parágrafo de introdução.
- ⑤ Na linha 9, insira uma tag *p* inicial na frente e adicione um *br* no final da linha.
- ⑥ Entre as linhas 9 e 832, adicione um *br* no final de cada linha que começa com um determinado número de espaços.
- ⑦ Ao final do poema, concatene uma tag *p* final.
- ⑧ Depois da linha 13, substitua cada uma das linhas em branco por uma quebra (*br*).
- ⑨ Concatene algumas tags no final (\$) do documento.

Para aplicar esse arquivo de comandos ao arquivo *rime.txt*, digite a linha abaixo, seguida de Enter ou Return: `sed -E -f html.sed rime.txt`
Para redirecionar a saída para um arquivo, execute: `sed -E -f html.sed rime.txt > rime.html` Abra o arquivo *rime.html* em um navegador para ver o que foi criado (ver figura 9.1).

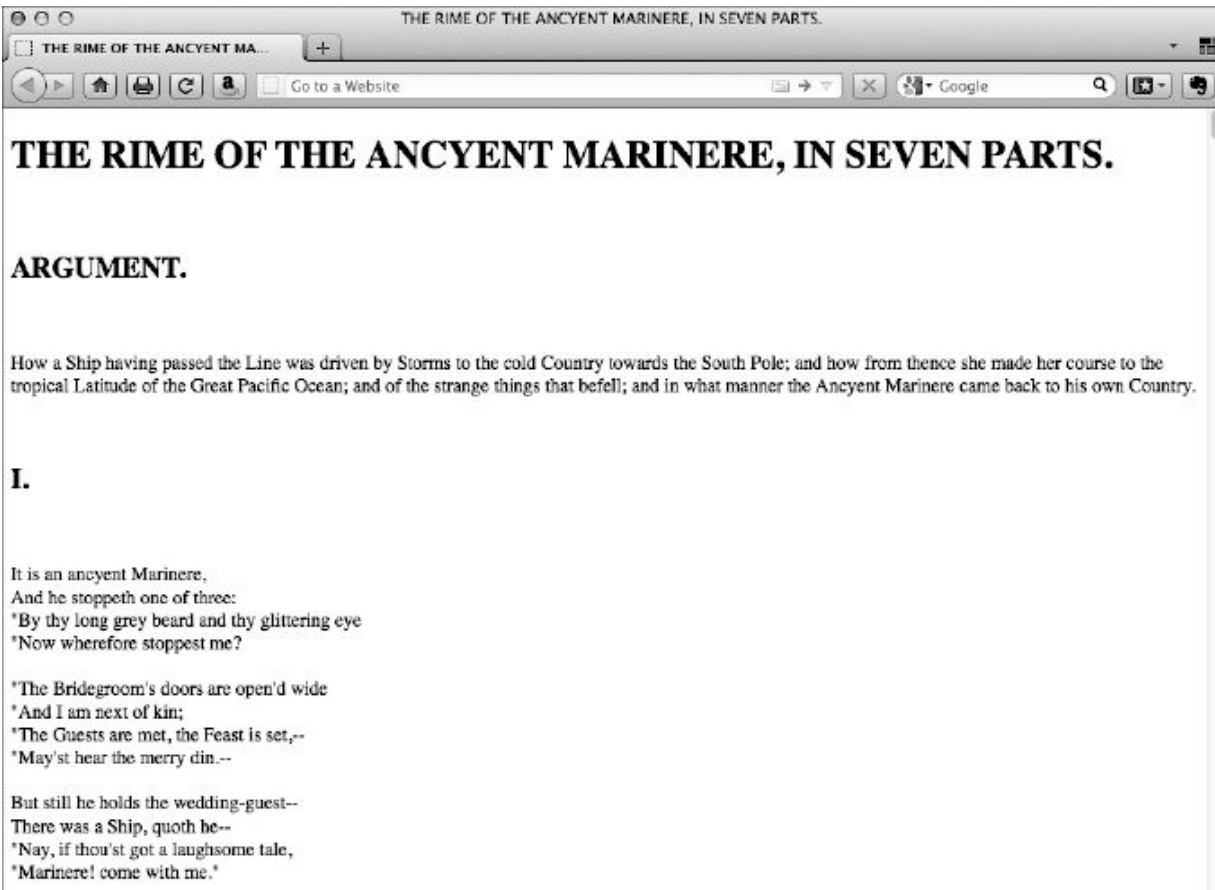


Figura 9.1 – O arquivo *rime.html* no Firefox.

Transformando texto normal usando Perl

Agora mostrarei como inserir marcação HTML em um arquivo usando Perl. Em primeiro lugar, do mesmo modo como fizemos com o *sed*, eu apresentarei uma série de comandos de uma linha; em seguida, mostrarei esses mesmos comandos em um arquivo.



Este livro apresenta somente os rudimentos da linguagem Perl e como começar a usá-los. Não é um tutorial ou manual de Perl, mas eu espero despertar seu interesse nessa linguagem e mostrar algumas de suas possibilidades. Um bom lugar para começar a conhecer Perl é no site *Learning Perl*, que pode ser acessado em <http://learn.perl.org/>, o qual inclui também instruções sobre como instalá-lo.

Se a linha atual (\$) for a linha 1, atribua a linha toda (\$) à variável \$title e mostre o conteúdo de \$title.

```
perl -ne 'if ($. == 1) {chomp($title = $_); print "<h1>" . $title . "</h1>" . "\n";};'
```

rime.txt Se tudo funcionar corretamente, o resultado deve ser: `<h1>THE RIME OF THE ANCYENT MARINERE, IN SEVEN PARTS.</h1>` Aqui está uma explicação para o comando Perl: • Teste se você está na linha 1 usando \$.

- Consoma a linha (\$_) e atribua a string à variável \$title. Quando você consome a linha usando a função `chomp`, ela remove o caractere de mudança de linha que está no final da string.
- Mostre \$title em um elemento `h1`, seguido por uma mudança de linha (`\n`).



Para mais informações sobre as variáveis built-in do Perl, como, por exemplo, o \$., digite o comando `perldoc -v $.` em um prompt (o *perldoc* normalmente é instalado quando você instala o Perl). Se isso não funcionar, consulte a seção de “Notas técnicas”.

Para inserir algumas marcações no início do arquivo, incluindo a tag `h1`, use: `perl -ne 'if ($. == 1) {chomp($title = $_); print "<!DOCTYPE html>\n <html xmlns=\"http://www.w3.org/1999/xhtml\">\n <head>\n <title>$title</title>\n <meta charset=\"utf-8\"/>\n </head>\n <body>\n <h1>$title</h1>\n" if $. == 1; exit}' rime.txt` e você obterá o seguinte resultado:

```
<!DOCTYPE html> <html
xmlns="http://www.w3.org/1999/xhtml"> <head> <title>THE RIME
OF THE ANCYENT MARINERE, IN SEVEN PARTS.</title> <meta
charset="utf-8"/> </head> <body> <h1>THE RIME OF THE
ANCYENT MARINERE, IN SEVEN PARTS.</h1>
```

A função *print* imprime as tags que vêm em seguida, e cada linha (exceto a última) é seguida por uma `\n`, a qual insere uma mudança de linha na saída. A variável \$title é expandida dentro dos elementos *title* e *h1*.

Lidando com algarismos romanos usando Perl

Para inserir tags no cabeçalho e nas divisões de seção que utilizam algarismos romanos, use: `perl -ne 'print if s/^(ARGUMENT\.[I]{0,3}V?I{0,2}\.)/<h2>\1</h2>/' rime.txt` Esta é a saída: `<h2>ARGUMENT.</h2> <h2>I.</h2> <h2>II.</h2> <h2>III.</h2> <h2>IV.</h2> <h2>V.</h2> <h2>VI.</h2> <h2>VII.</h2>` O comando de substituição (*s*) captura o cabeçalho *ARGUMENT* e os sete algarismos romanos em letras maiúsculas, cada um deles em linhas separadas e seguidos por um ponto, no intervalo de I a VII. Depois uma tag *h2* é colocada

ao redor do texto capturado.

Lidando com um parágrafo específico usando Perl

Use este código para colocar um elemento *p* ao redor do parágrafo de introdução, se o número da linha for igual a 5: `perl -ne 'if ($. == 5) {s/^([A-Z].*)$/<p>$1</p>/;print;}' rime.txt` Você deverá ver este resultado: `<p>How a Ship having passed the Line was driven by Storms to the cold Country towards the South Pole; and how from thence she made her course to the tropical Latitude of the Great Pacific Ocean; and of the strange things that befell; and in what manner the Ancyent Marinere came back to his own Country.</p>`

Lidando com as linhas do poema usando Perl

O comando a seguir insere uma tag *p* inicial no começo da primeira linha do poema e uma tag *br* no final dessa linha: `perl -ne 'if ($. == 9) {s/^[]*(.*)/ <p>$1
/;print;}' rime.txt` O resultado será: `<p>It is an ancyent Marinere,
` A seguir, entre as linhas 10 e 832, este pequeno código em Perl insere um *br* no final de cada linha do poema: `perl -ne 'if (10..832) { s/^([]{5,7}.*)/$1
/; print;}' rime.txt` Uma amostra do que você verá: `Farewell, farewell! but this I tell
To thee, thou wedding-guest!
He prayeth well who loveth well
Both man and bird and beast.
` Adicione uma tag *p* final no fim da última linha do poema.

`perl -ne 'if ($. == 833) {s/^(.*)/$1</p>/; print;}' rime.txt` O resultado será: `He rose the morrow morn.</p>` Substitua as linhas em branco por uma tag *br*: `perl -ne 'if (9..eof) {s/^$/
/; print;}' rime.txt` para obter: `
He prayeth best who loveth best, All things both great and small: For the dear God, who loveth us, He made and loveth all.`

`
The Marinere, whose eye is bright, Whose beard with age is hoar, Is gone; and now the wedding-guest Turn'd from the bridegroom's door.`

`
` E, por último, quando o final do arquivo for encontrado, imprima duas tags finais: `perl -ne 'if (eof) {print "</body>\n</html>\n";}' rime.txt` Todo esse código funciona junto de modo mais simples quando está em um arquivo. Você verá como fazer isso a seguir.

Usando um arquivo de comandos com Perl

A seguir, uma listagem do arquivo *html.pl*, o qual transforma o arquivo *rime.txt* em HTML usando Perl. A numeração no exemplo fornece orientações acerca do que acontece no script.

```

#!/usrbin/perl -p ❶
if ($. == 1) { ❷
chomp($title = $_); }
print "<!DOCTYPE html>\n" ❸
<html xmlns="http://www.w3.org/1999/xhtml">\ <head>\ <title>$title</title>\ <meta
charset="utf-8"/>\ </head>\ <body>\ <h1>$title</h1>\n" if $. == 1; s/^(ARGUMENT|l{0,3}V?
l{0,2})\.$/<h2>$1</h2>/; ❹
if ($. == 5) { ❺
s/^([A-Z].*)$/<p>$1</p>/; }
if ($. == 9) { ❻
s/^[ ]*(.*)/ <p>$1<br/>/; }
if (10..832) { ❼
s/^[ ]{5,7}.*)/$1<br/>/; }
if (9..eof) { ❽
s/^$/<br/>/; }
if ($. == 833) { ❾
s/^(.*)$/<p>$1</p>\n </body>\n</html>\n/; }

```

- ❶ É chamada de diretiva *shebang*, que dá uma dica ao shell sobre o local em que se encontra o programa que você está executando.
- ❷ Se a linha atual (\$) for a linha 1, então atribua a linha toda (\$) à variável \$title, removendo (com chomp) o último caractere da string (uma mudança de linha) no processo.
- ❸ Imprima um doctype e várias tags HTML no início do documento na linha 1 e reutilize o valor da variável \$title em diversos lugares.
- ❹ Insira tags *h2* no cabeçalho *ARGUMENT* e nos algarismos romanos.
- ❺ Coloque tags *p* ao redor do parágrafo de introdução.
- ❻ Coloque uma tag *p* inicial no começo da primeira linha de verso e concatene um *br* nessa linha.
- ❼ Anexe uma tag *br* no final de cada linha de verso, exceto na última linha.
- ❽ Substitua cada linha em branco, depois da linha 9, por uma tag *br*.
- ❾ Anexe as tags *p*, *body* e *html* finais na última linha.

Para executar tudo isso, basta fazer o seguinte: perl html.pl rime.txt
 Você também pode redirecionar a saída usando um > para salvar a

saída em um arquivo. No próximo e último capítulo, eu concluirei nosso tutorial sobre regex.

O que você aprendeu no capítulo 9

- Como usar o *sed* na linha de comando.
- Como inserir (na frente), substituir e concatenar texto (tags) usando *sed*.
- Como usar o Perl para fazer o mesmo.

Notas técnicas

- O AsciiDoc (<http://www.methods.co.nz/asciidoc/>) de Stuart Rackham é um formato de texto que pode ser convertido em HTML, PDF, ePUB, DocBook e páginas man, usando um processador Python. A sintaxe dos arquivos de texto é similar à do Wiki ou do Markdown e permite inserir marcações HTML ou XML de forma muito mais rápida do que se fossem inseridas manualmente.
- O caractere de underscore se aplica somente às tags XML, e não às tags HTML. Além do mais, as tags XML podem, obviamente, ter uma variedade muito maior de caracteres em seus nomes em comparação com aquilo que é representado pelo conjunto de caracteres ASCII. Para mais informações sobre os caracteres usados nos nomes XML, consulte o site <http://www.w3.org/TR/REC-xml/#sec-common-syn>.
- Se o comando `perldoc` não funcionar, algumas alternativas estão disponíveis. Em primeiro lugar, você pode facilmente obter informações online sobre Perl no site <http://perldoc.perl.org>. (Para saber mais sobre o \$., por exemplo, consulte o site <http://perldoc.perl.org/perlvar.html#Variables-related-to-filehandles>.) Se você está usando Mac, experimente o `perldoc5.12`. Se o Perl foi instalado a partir do ActiveState, você o encontrará em `usrlocal/bin` quando compilado e gerado a partir do código-fonte. Você pode adicionar o diretório `usrlocal/bin` ao seu path, para que o `perl` e o `perldoc` possam ser executados. Para informações sobre configuração de sua variável de path, consulte o site

<http://java.com/en/download/help/path.xml>.

capítulo 10

O fim do começo

“O Unix não foi projetado para que você parasse de fazer coisas estúpidas porque isso faria com que você também parasse de fazer coisas inteligentes.” – Doug Gwyn Parabéns por ter conseguido chegar tão longe. Você não é mais um usuário inexperiente de expressões regulares. Você foi apresentado às sintaxes mais comumente usadas em expressões regulares. E isso abrirá uma porção de possibilidades para você em seu trabalho como programador.

Aprender as expressões regulares fez com que eu economizasse um bocado de tempo. Deixe-me dar um exemplo.

Eu uso muito XSLT no trabalho e com frequência preciso analisar as tags existentes em um grupo de arquivos XML.

Mostrei uma parte do que vem a seguir no capítulo anterior, mas aqui está um comando longo de uma linha que extrai uma lista de nomes de tags do arquivo *lorem.dita* e a converte em uma folha de estilo XSLT simples: `grep -Eo '<[_a-zA-Z][^>]*>' lorem.dita | sort | uniq | sed '1 i\ <xml:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">\ ; s/^<\<xsl:template match=";s id="\."/;s>$/">\ <xsl:apply-templatesV>\<\xsl:template>/;$ a\ \</xsl:stylesheet>\ '`

Eu sei que esse script pode parecer um pouco acrobático, mas depois de trabalhar durante muito tempo com esse tipo de coisa, você começa a pensar dessa maneira. Eu nem vou explicar o que eu fiz aqui porque tenho certeza de que, a essa altura, você é capaz de descobrir sozinho.

Eis o que você terá como resultado: `<xml:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"> <xsl:template match="body"> <xsl:apply-templates/> </xsl:template> <xsl:template match="li"> <xsl:apply-templates/> </xsl:template> <xsl:template match="p"> <xsl:apply-templates/> </xsl:template> <xsl:template match="title"> <xsl:apply-templates/> </xsl:template> <xsl:template`

```
match="topic"> <xsl:apply-templates/> </xsl:template> <xsl:template
match="ul"> <xsl:apply-templates/> </xsl:template> </xsl:stylesheet>
```

Isso é apenas o começo. É claro que essa folha de estilo simples precisará de um bocado de edição antes que possa servir para fazer qualquer coisa útil, mas esse é o tipo de coisa que pode fazer você economizar bastante digitação.

Admito que teria sido mais fácil se eu tivesse colocado esses comandos `sed` em um arquivo. Para dizer a verdade, eu os coloquei. Você encontrará o arquivo `xslt.sed` no arquivo de exemplos. Aqui está o arquivo: `#!/usr/bin/sed 1 i\ <xml:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">\ s/^</> <xsl:template match="";s id="\.*"/>s$/>\ <xsl:apply-templates/>\ </xsl:template>/;$ a\ \`

`</xsl:stylesheet>\` E aqui está o modo de executá-lo: `grep -Eo '<[_a-zA-Z][^>]*>' lorem.dita | sort | uniq | sed -f xslt.sed`

Aprendendo mais¹

Apesar de ter um bom domínio sobre as regex agora, ainda há muito que aprender. Eu tenho algumas sugestões sobre a direção a ser tomada a seguir.

Eu ofereço essas recomendações como resultado de experiência e de observação, e não por causa de qualquer senso de obrigação ou para “vender” a ideia. Não terei nenhuma vantagem pelo fato de mencioná-las. Falarei sobre elas porque esses recursos realmente irão beneficiá-lo.

O livro *Dominando Expressões Regulares*, 3ª edição (<http://www.altabooks.com.br/dominando-expressoos-regulares.html>), de Jeffrey E. F. Friedl, é a fonte que muitos programadores buscam para um entendimento definitivo das expressões regulares. Expansivo e bem escrito. Se você pretende fazer qualquer trabalho significativo com regex, é preciso ter esse livro em sua estante ou em seu e-reader. Ponto final.

O livro *Expressões Regulares Cookbook* (<http://novatec.com.br/livros/regexpcookbook/>), de Jan Goyvaerts e

Steven Levithan, é outra obra excelente, especialmente se você vai comparar diferentes implementações. Eu compraria esse livro também.

O livro *Expressões Regulares – Guia de Bolso: Expressões Regulares para Perl, Ruby, PHP, Python, C, Java e .NET* (<http://www.altabooks.com.br/guia-de-bolso-expressoes-regulares.html>), de Tony Stubblebine, é um pequeno guia que, apesar de já ter sido publicado há vários anos, continua sendo popular.

O livro *Beginning Regular Expressions* (Wrox, 2005), de Andrew Watt, tem ótima reputação. Eu achei particularmente útil o tutorial de Bruce Barnett a respeito do *sed* (ver <http://www.grymoire.com/Unix/Sed.html>). Ele demonstra uma série de comandos menos compreendidos do *sed*, recursos que eu não expliquei neste livro.

Ferramentas, implementações e bibliotecas interessantes

Eu mencionei diversas ferramentas, implementações e bibliotecas neste livro. Vou dar uma recapitulada nelas e mencionarei outras aqui.

Perl

O Perl é uma linguagem de programação popular, de uso geral. Muitas pessoas preferem Perl para processamento de textos usando expressões regulares a outras linguagens. Provavelmente, você já o tem, mas para informações sobre como instalar Perl em seu sistema, acesse o site <http://www.perl.org/get.html>. Leia a respeito de expressões regulares em Perl no site <http://perldoc.perl.org/perlre.html>. Não me compreenda mal. Há muitas outras linguagens que fazem um excelente trabalho com regex, mas vale a pena ter Perl na sua caixa de ferramentas. Para aprender mais, eu obteria uma cópia da última edição de *Learning Perl* (<http://shop.oreilly.com/product/0636920018452.do>), de Randal Schwartz, brian d foy e Tom Phoenix, publicado pela O'Reilly.

PCRE

O Perl Compatible Regular Expressions ou PCRE (ver <http://www.pcre.org>) é uma biblioteca de expressões regulares escrita em C (para 8 e 16 bits). Essa biblioteca consiste principalmente em funções que podem ser chamadas a partir de qualquer framework em C ou a partir de qualquer outra linguagem que possa usar bibliotecas C. Ela é compatível com as expressões regulares do Perl 5, como seu nome sugere, e inclui alguns recursos de outras implementações de regex. O editor Notepad++ usa a biblioteca PCRE.

O *pcregrep* é uma ferramenta de 8 bits no estilo do *grep* que permite que você use recursos da biblioteca PCRE na linha de comando. Você usou essa ferramenta no capítulo 3. Acesse o site <http://www.pcre.org> para informações sobre download (a partir de <ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/>).

Você pode obter o *pcregrep* para o Mac acessando o Macports (<http://www.macports.org>) e executando o comando `sudo port install pcre` (Xcode é um pré-requisito; ver <https://developer.apple.com/technologies/tools/>; um login será necessário). Para instalá-lo na plataforma Windows (binários), acesse o site <http://gnuwin32.sourceforge.net/packages/pcre.htm>.

Ruby (Oniguruma)

O Oniguruma é uma biblioteca de expressões regulares que é compatível com Ruby 1.9; consulte o site <http://oniguruma.rubyforge.org/>. Ela está codificada em C e foi escrita especificamente para suportar Ruby. Você pode experimentar as expressões regulares do Ruby usando o Rubular, uma aplicação online que suporta tanto a versão 1.8.7 quanto a 1.9.2 (ver <http://www.rubular.com> e a figura 10.1). O TextMate, a propósito, usa a biblioteca Oniguruma.



Figura 10.1 – Regex do número de telefone no Rubular.

Python

O Python é uma linguagem de programação de uso geral que suporta expressões regulares (ver <http://www.python.org>). O Python foi inicialmente criado por Guido van Rossum em 1991. Você pode ler a respeito da sintaxe de expressões regulares em Python 3 neste site: <http://docs.python.org/py3k/library/re.html?highlight=regular%20expressions>.

RE2

O RE2 é uma biblioteca de expressões regulares em C++ que não faz backtracking (ver <http://code.google.com/p/re2>). Embora o RE2 seja bem rápido, ele não faz backtracking ou referências para trás. O RE2 está disponível na forma de um pacote CPAN para Perl e pode contar com a biblioteca nativa do Perl, caso seja necessário o uso de retrovisores. Para instruções sobre chamadas de API, consulte o site <http://code.google.com/p/re2/wiki/CplusplusAPI>. Para uma discussão interessante sobre RE2, ver “Regular Expression Matching in the Wild” no site <http://swtch.com/~rsc/regexp/regexp3.html>.

Correspondendo a um número de telefone no padrão norte-americano

Você se lembra do exemplo com o número de telefone no padrão norte-americano, usado no primeiro capítulo? Você percorreu um longo caminho desde então.

Aqui está uma expressão regular mais robusta para corresponder a números de telefone quando comparada com aquela que usamos anteriormente. Ela foi adaptada do exemplo de Goyvaerts e Levithan que está no livro *Expressões Regulares Cookbook* (1a edição).

```
^\((?:\d{3})\)?[-.]?(?:\d{3})[-.]?(?:\d{4})$
```

Brinque com essa expressão usando a ferramenta que preferir (veja-a no Reggy na figura 10.2). A esta altura, você deverá ser capaz de entender essa regex praticamente sem a necessidade de ser conduzido pela mão. Isso me deixa orgulhoso de você. Mas eu vou explicá-la, de qualquer modo.

Caracteres	Descrição
<code>^</code>	É a asserção de largura zero para o início de uma linha ou de um assunto
<code>\(?</code>	É um abre parênteses literal, mas ele é opcional (?)
<code>(?:\d{3})</code>	É um grupo de não-captura que corresponde a três dígitos consecutivos
<code>\)?</code>	É um fecha parênteses opcional
<code>[-.]?</code>	Permite um hífen ou ponto opcional
<code>(?:\d{3})</code>	É outro grupo de não-captura que corresponde a mais três dígitos consecutivos
<code>[-.]?</code>	Permite um hífen ou um ponto opcional, novamente
<code>(?:\d{4})</code>	É outro grupo de não-captura que corresponde exatamente a quatro dígitos consecutivos

Caracteres	Descrição
\$	Corresponde ao final de uma linha ou um assunto

Essa expressão poderia ser mais sofisticada ainda, mas deixo isso a seu encargo porque você agora é capaz de fazê-lo sozinho.



Figura 10.2 – Regex do número de telefone no Reggy.

Correspondendo a um endereço de email

Por último, vou apresentar mais uma expressão regular que corresponde a um endereço de email: `^([\\w-!#$%&'*+./=?^_`{|}~]+)@((?:\\w+\\.)+)(?:[a-zA-Z]{2,4})$`

Esta é uma adaptação de uma expressão disponibilizada por Grant Skinner no RegExr. Gostaria de desafiá-lo a dar o melhor de si para explicar o que cada um dos caracteres significa no contexto de uma expressão regular e ver se você consegue melhorar essa expressão. Estou certo de que você é capaz disso.

Obrigado pelo seu tempo. Gostei muito de passá-lo com você. A partir de agora, você deverá ter um bom domínio dos conceitos fundamentais relacionados às expressões regulares. Você não faz mais parte do clube dos iniciantes. Espero que você tenha feito amizade com as expressões regulares e que tenha aprendido alguma coisa que valha a pena ao longo do caminho.

O que você aprendeu no capítulo 10

- Como extrair uma lista de elementos XML de um documento e converter a lista em uma folha de estilo XSLT.
- Onde encontrar recursos adicionais para aprender a respeito de expressões regulares.
- Quais são algumas ferramentas, implementações e bibliotecas interessantes.
- Um padrão um pouco mais robusto para corresponder a um número de telefone no padrão norte-americano.

¹ Conheça também o excelente livro *Expressões Regulares – Uma Abordagem Divertida*, de autoria de Aurelio Jargas, publicado pela Novatec Editora.

apêndice

Referência para Expressões Regulares

Este apêndice é uma referência para as expressões regulares.

Expressões regulares no QED

O QED (abreviação de *Quick Editor*) foi escrito originalmente para o *Berkeley Time-Sharing System*, que era executado no *Scientific Data Systems SDS 940*. Uma versão reescrita do editor QED original, feita por Ken Thompson para o *Compatible Time-Sharing System* do MIT, resultou em uma das implementações práticas mais antigas (se não a primeira) das expressões regulares em computação. A tabela A.1, extraída das páginas 3 e 4 de um memorando do Bell Labs, escrito em 1970 (original em inglês), apresenta os recursos de regex do QED. O que me deixa impressionado é que a maioria dessa sintaxe permanece em uso até os dias de hoje, mais de quarenta anos depois.

Tabela A.1 – Expressões regulares no QED

Recurso	Descrição
<i>literal</i>	“a) um caractere comum [literal] é uma expressão regular que corresponde a esse caractere.”
<code>^</code>	“b) <code>^</code> é uma expressão regular que corresponde ao caractere nulo no início de uma linha.”
<code>\$</code>	“c) <code>\$</code> é uma expressão regular que corresponde ao caractere nulo antes do caractere <code><nl></code> [mudança de linha] (geralmente no final de uma linha).”
<code>.</code>	“d) <code>.</code> é uma expressão regular que corresponde a qualquer caractere, exceto a <code><nl></code> [mudança de linha].”
<code>[<string>]</code>	“e) <code>[<string>]</code> é uma expressão regular que corresponde a qualquer um dos caracteres em <code><string></code> e a nenhum outro.”

Recurso	Descrição
[[^] <string>]	“f) “[[^] <string>] é uma expressão regular que corresponde a qualquer caractere, exceto a <nl> [mudança de linha] e aos caracteres de <string>.”
*	<p>“g) Uma expressão regular seguida de “*” é uma expressão regular que corresponde a qualquer número (inclusive zero) de ocorrências adjacentes do texto correspondido pela expressão regular .”</p> <p>“h) Duas expressões regulares adjacentes formam uma expressão regular que corresponde a ocorrências adjacentes de texto correspondido pelas expressões regulares.”</p>
	“i) Duas expressões regulares separadas por “ ” formam uma expressão regular que corresponde ao texto correspondido por qualquer uma das expressões regulares.”
()	“j) Uma expressão regular entre parênteses é uma expressão regular que corresponde ao mesmo texto que a expressão regular original. Os parênteses são usados para alterar a ordem de avaliação implicada por g), h) e i): a(b c)d corresponde a abd ou a acd, enquanto ab cd corresponde a ab ou cd.”
{ }	“k) Se “<regexp>” for uma expressão regular, “{<regexp>}x” é uma expressão regular na qual x é qualquer caractere. Essa expressão regular corresponde ao mesmo que <regexp> corresponde; ela tem certos efeitos colaterais conforme explicado no comando Substitute.” [O comando Substitute tinha o formato (..)S/<regexp>/<string>/ (consulte a página 13 do memorando), similar ao modo como ainda é usado em programas, como sed e Perl.]

Recurso	Descrição
\E	<p>“l) Se <rexname> for o nome de uma expressão regular nomeada com o comando E (abaixo), então “\E<rexname>” é uma expressão regular que corresponde ao mesmo que a expressão regular especificada no comando E. Discussões adicionais serão apresentadas na seção sobre o comando E.” [O comando \E permitia dar nome a uma expressão regular e usá-la novamente acessando seu nome.]</p> <p>“m) A expressão regular nula sozinha é equivalente à última expressão regular encontrada. Inicialmente, a expressão regular nula é indefinida; ela também se torna indefinida depois de uma expressão regular incorreta e após o uso do comando E.”</p> <p>“n) Nada mais é uma expressão regular.”</p> <p>“o) Nenhuma expressão regular corresponderá a textos espalhados por mais de uma linha.”</p>

Metacaracteres

Existem catorze metacaracteres que são usados em expressões regulares, cada um dos quais possuindo um significado especial, conforme descrito na tabela A.2. Se você quiser usar um desses caracteres como um literal, é preciso inserir uma barra invertida antes para escapá-lo. Por exemplo, você escaparia o sinal de cifrão deste modo: \\$; ou uma barra invertida deste modo: \.

Tabela A.2 – Metacaracteres em expressões regulares

Metacaractere	Nome	Code Point	Função
.	Ponto	U+002E	Corresponde a qualquer caractere
\	Barra invertida	U+005C	Escapa um caractere

Metacaractere	Nome	Code Point	Função
	Barra Vertical	U+007C	Alternância (ou)
^	Circunflexo	U+005E	Âncora para início de linha
\$	Cifrão	U+0024	Âncora para fim de linha
?	Ponto de interrogação	U+003F	Quantificador zero ou um
*	Asterisco	U+002A	Quantificador zero ou mais
+	Sinal de mais	U+002B	Quantificador um ou mais
[Abertura de colchete	U+005B	Abre classe de caracteres
]	Fechamento de colchete	U+005D	Fecha classe de caracteres
{	Abertura de chaves	U+007B	Abre quantificador ou bloco
}	Fechamento de chaves	U+007D	Fecha quantificador ou bloco
(Abre parênteses	U+0028	Abre grupo
)	Fecha parênteses	U+0029	Fecha grupo

Shorthand de caracteres

A tabela A.3 lista os *shorthands* de caracteres usados em expressões regulares.

Tabela A.3 – Shorthand de caracteres

Shorthand de caracteres	Descrição	Shorthand de caracteres	Descrição
\a	Alerta	\xxxx	Valor de um caractere em octal
\b	Borda de palavra	\s	Branco
[\b]	Backspace	\S	Não-branco
\B	Não-borda de palavra	\t	Tabulação horizontal
\cx	Caractere Control	\v	Tabulação vertical
\d	Dígito	\w	Caractere de palavra
\D	Não-dígito	\W	Não-caractere de palavra
\dxxx	Valor de um caractere decimal em	\0	Caractere nulo
\f	Alimentação de formulário	\xxx	Valor de um caractere em hexadecimal
\r	Retorno de carro	\uxxxx	Valor de um caractere em Unicode
\n	Mudança de linha		

Espaços em branco

A tabela A-4 contém uma lista de *shorthands* de caracteres para

espaços em branco.

Tabela A.4 – Caracteres brancos

Shorthand de caracteres	Descrição
\f	Alimentação de formulário
\h	Branco horizontal
\H	Não-branco horizontal
\n	Mudança de linha
\r	Retorno de carro
\t	Tabulação horizontal
\v	Branco vertical
\V	Não-branco vertical

Caracteres Unicode brancos

Os caracteres Unicode brancos estão listados na tabela A.5.

Tabela A.5 – Caracteres brancos em Unicode

Abreviação ou apelido	Nome	Code Point Unicode	Regex
HT	Tabulação horizontal	U+0009	\u0009 ou \t
LF	Mudança de linha	U+000A	\u000A ou \n
VT	Tabulação vertical	U+000B	\u000B ou \v
FF	Alimentação de formulário	U+000C	\u000C ou \f

Abreviação ou apelido	Nome	Code Point Unicode	Regex
CR	Retorno de carro	U+000D	\u000d ou \r
SP	Espaço	U+0020	\u0020 ou \s*
NEL	Próxima linha	U+0085	\u0085
NBSP	Espaço sem quebra	U+00A0	\u00A0
—	Marca de espaço de Ogham	U+1680	\u1680
MVS	Separador de vogal mongol	U+180E	\u180E
BOM	Marca de ordem de byte	U+FEFF	\ufeff
NQSP	Meio quadratim	U+2000	\u2000
MQSP, Mutton Quad	Quadratim	U+2001	\u2001
ENSP, Nut	Espaço de meio quadratim	U+2002	\u2002
EMSP, Mutton	Espaço de quadratim	U+2003	\u2003
3MSP, Thick space	Terço de quadratim	U+2004	\u2004
4MSP, Mid space	Quarto de quadratim	U+2005	\u2005
6/MSP	Sexto de quadratim	U+2006	\u2006
FSP	Espaço tabular	U+2007	\u2007
PSP	Espaço de pontuação	U+2008	\u2008
THSP	Espaço fino	U+2009	\u2009
HSP	Espaço ultrafino	U+200A	\u200A

Abreviação ou apelido	Nome	Code Point Unicode	Regex
ZWSP	Espaço de largura zero	U+200B	\u200B
LSEP	Separador de linha	U+2028	\u2028
PSEP	Separador de parágrafo	U+2029	\u2029
NNBSP	Espaço sem quebra estreito	U+202F	\u202F
MMSP	Espaço matemático médio	U+205F	\u205f
IDSP	Espaço para caractere ideográfico	U+3000	\u3000

* Também corresponde a outros espaços em branco.

Caracteres de controle

A tabela A.6 mostra uma maneira de corresponder a caracteres de controle em expressões regulares.

Tabela A.6 – Correspondendo a caracteres de controle

Caractere de controle	Valor em Unicode	Abreviação	Nome
c@*	U+0000	NUL	Nulo
\cA	U+0001	SOH	Início de cabeçalho
\cB	U+0002	STX	Início de texto
\cC	U+0003	ETX	Fim de texto
\cD	U+0004	EOT	Fim de transmissão
\cE	U+0005	ENQ	Enquiry (Interroga)

Caractere de controle	Valor em Unicode	Abreviação	Nome
\cF	U+0006	ACK	Acknowledge (Reconhecimento)
\cG	U+0007	BEL	Sino
\cH	U+0008	BS	Backspace
\cI	U+0009	HT	Tabulação de caractere ou tabulação horizontal
\cJ	U+000A	LF	Mudança de linha (linha nova, fim de linha)
\cK	U+000B	VT	Tabulação de linha ou tabulação vertical
\cL	U+000C	FF	Alimentação de formulário
\cM	U+000D	CR	Retorno de carro
\cN	U+000E	SO	Shift out
\cO	U+000F	SI	Shift in
\cP	U+0010	DLE	Data link escape
\cQ	U+0011	DC1	Controle de dispositivo um
\cR	U+0012	DC2	Controle de dispositivo dois
\cS	U+0013	DC3	Controle de dispositivo três
\cT	U+0014	DC4	Controle de dispositivo quatro
\cU	U+0015	NAK	Reconhecimento negativo
\cV	U+0016	SYN	Ocioso síncrono
\cW	U+0017	ETB	Fim de bloco de transmissão

Caractere de controle	Valor em Unicode	Abreviação	Nome
\cX	U+0018	CAN	Cancelar
\cY	U+0019	EM	Fim de mídia
\cZ	U+001A	SUB	Substituir
\c[U+001B	ESC	Escape
\c\	U+001C	FS	Separador de informação quatro
\c]	U+001D	GS	Separador de informação três
\c^	U+001E	RS	Separador de informação dois
\c_	U+001F	US	Separador de informação um

* Pode usar letras maiúsculas ou minúsculas. Por exemplo, \cA ou \ca são equivalentes; porém, as implementações Java exigem letras maiúsculas.

Propriedades de caracteres

A tabela A.7 lista os nomes das propriedades de caracteres para uso com \p{propriedade} ou \P{propriedade}.

*Tabela A.7 – Propriedades de caracteres**

Propriedade	Descrição	Propriedade	Descrição
C	Outro	NI	Dígito de letra
Cc	Controle	No	Outro número
Cf	Formatação	P	Pontuação
Cn	Não atribuído	Pc	Pontuação de conector

Propriedade	Descrição	Propriedade	Descrição
Co	Uso privado	Pd	Pontuação de traço
Cs	Substituto	Pe	Pontuação de fechamento
L	Letra	Pf	Pontuação final
LI	Letra minúscula	Pi	Pontuação inicial
Lm	Letra de modificação	Po	Outra pontuação
Lo	Outra letra	Ps	Pontuação de abertura
Lt	Letra de título	S	Símbolo
Lu	Letra maiúscula	Sc	Símbolo de moeda
L&	LI, Lu ou Lt	Sk	Símbolo modificador
M	Marca	Sm	Símbolo matemático
Mc	Marca de espaçamento	So	Outro símbolo
Me	Marca de circunscrição	Z	Separador
Mn	Não-marca de espaçamento	Zl	Separador de linha
N	Número	Zp	Separador de parágrafo
Nd	Número decimal	Zs	Separador de espaço

* Ver a seção pcreyntax(3) em <http://www.pcre.org/pcre.txt>.

Nomes de script para propriedades de caracteres

A tabela A.8 mostra os nomes de script de idiomas para usar com `/p{propriedade}` ou `/P{propriedade}`.

*Tabela A.8 – Nomes de script**

Arabic (Arab)	Glagolitic (Glag)	Lepcha (Lepc)	Samaritan (Samr)
Armenian (Armn)	Gothic (Goth)	Limbu (Limb)	Saurashtra (Saur)
Avestan (Avst)	Greek (Grek)	Linear B (Linb)	Shavian (Shaw)
Balinese (Bali)	Gujarati (Gujr)	Lisu (Lisu)	Sinhala (Sinh)
Bamum (Bamu)	Gurmukhi (Guru)	Lycian (Lyci)	Sundanese (Sund)
Bengali (Beng)	Han (Hani)	Lydian (Lydi)	Syloti Nagri (Sylo)
Bopomofo (Bopo)	Hangul (Hang)	Malayalam (Mlym)	Syriac (Syrç)
Braille (Brai)	Hanunoo (Hano)	Meetei Mayek (Mtei)	Tagalog (Tglg)
Buginese (Bugi)	Hebrew (Hebr)	Mongolian (Mong)	Tagbanwa (Tagb)
Buhid (Buhd)	Hiragana (Hira)	Myanmar (Mymr)	Tai Le (Tale)
Canadian Aboriginal (Cans)	Hrkt: Katakana ou Hiragana)	New Tai Lue (Talu)	Tai Tham (Lana)
Carian (Cari)	Imperial Aramaic (Armi)	Nko (Nkoo)	Tai Viet (Tavt)
Cham (None)	Inherited (Zinh/Qaai)	Ogham (Ogam)	Tamil (Taml)
Cherokee (Cher)	Inscriptional Pahlavi (Phli)	Ol Chiki (Olck)	Telugu (Telu)
Common (Zyyy)	Inscriptional Parthian	Old Italic (Ital)	Thaana

	(Prti)		(Thaa)
Coptic (Copt/Qaac)	Javanese (Java)	Old Persian (Xpeo)	Thai (None)
Cuneiform (Xsux)	Kaithi (Kthi)	Old South Arabian (Sarab)	Tibetan (Tibt)
Cypriot (Cprt)	Kannada (Knda)	Old Turkic (Orkh)	Tifinagh (Tfng)
Cyrillic (Cyr)	Katakana (Kana)	Oriya (Orya)	Ugaritic (Ugar)
Deseret (Dsrt)	Kayah Li (Kali)	Osmanya (Osma)	Unknown (Zzzz)
Devanagari (Deva)	Kharoshthi (Khar)	Phags Pa (Phag)	Vai (Vaii)
Egyptian Hieroglyphs (Egyp)	Khmer (Khmr)	Phoenician (Phnx)	Yi (Yiii)
Ethiopic (Ethi)	Lao (Laoo)	Rejang (Rjng)	
Georgian (Geor)	Latin (Latn)	Runic (Runr)	

* Ver a seção `pcresyntax(3)` em <http://www.pcre.org/pcre.txt> ou em <http://ruby.runpaint.org/regexps#properties>.

Classes de Caracteres POSIX

A tabela A.9 mostra uma lista de classes de caracteres POSIX.

Tabela A.9 – Classes de caracteres POSIX

Classe de caracteres	Descrição
<code>[:alnum:]</code>	Caracteres alfanuméricos (letras e dígitos)
<code>[:alpha:]</code>	Caracteres alfabéticos (letras)
<code>[:ascii:]</code>	Caracteres ASCII (todos os 128)

<code>[:blank:]</code>	Caracteres brancos
<code>[:ctrl:]</code>	Caracteres de controle
<code>[:digit:]</code>	Dígitos
<code>[:graph:]</code>	Caracteres gráficos
<code>[:lower:]</code>	Letras minúsculas
<code>[:print:]</code>	Caracteres imprimíveis
<code>[:punct:]</code>	Caracteres de pontuação
<code>[:space:]</code>	Espaços em branco
<code>[:upper:]</code>	Letras maiúsculas
<code>[:word:]</code>	Caracteres de palavra
<code>[:xdigit:]</code>	Dígitos hexadecimais

Opções/modificadores

As tabelas A.10 e A.11 listam opções e modificadores.

Tabela A.10 – Opções em expressões regulares

Opção	Descrição	Suportado por
(?d)	Linhas Unix	Java
(?i)	Não sensível ao caso	PCRE, Perl, Java
(?J)	Permitir nomes duplicados	PCRE*
(?m)	Múltiplas linhas	PCRE, Perl, Java
(?s)	Linha única (dotall)	PCRE, Perl, Java
(?u)	Diferenciar caso em Unicode	Java
(?U)	Correspondência preguiçosa (<i>lazy</i>) por padrão	PCRE

(?x)	Ignorar brancos, comentários	PCRE, Perl, Java
(?-...)	Desabilitar ou desligar opções	PCRE

* Consulte a seção “Named Subpatterns” em <http://www.pcre.org/pcre.txt>.

*Tabela A.11 – Modificadores do Perl (flags)**

Modificador	Descrição
a	Corresponder \d, \s, \w e POSIX somente no intervalo ASCII
c	Manter a posição atual depois de falha na correspondência
d	Usar regras padrão, nativas da plataforma
g	Correspondência global
i	Correspondência não sensível ao caso
l	Usar regras da localidade atual
m	Strings de múltiplas linhas
p	Conservar a string correspondida
s	Tratar strings como única linha
u	Usar regras Unicode ao corresponder
x	Ignorar brancos e comentários

* Ver <http://perldoc.perl.org/perlre.html#Modifiers>.

Tabela de códigos ASCII com Regex

A tabela A.12 é uma tabela de códigos ASCII com referências cruzadas para regex.

Tabela A.12 – Tabela de códigos ASCII

Binário	Oct	Dec	Hex	Carac	Tecl	Regex	Nome
---------	-----	-----	-----	-------	------	-------	------

Binário	Oct	Dec	Hex	Carac	Tecl	Regex	Nome
00000000	0	0	0	NUL	^@	\c@	Caractere nulo
00000001	1	1	1	SOH	^A	\cA	Início de cabeçalho
00000010	2	2	2	STX	^B	\cB	Início de texto
00000011	3	3	3	ETX	^C	\cC	Fim de texto
00000100	4	4	4	EOT	^D	\cD	Fim de transmissão
00000101	5	5	5	ENQ	^E	\cE	Enquiry (Interroga)
00000110	6	6	6	ACK	^F	\cF	Acknowledge (Reconhecimento)
00000111	7	7	7	BEL	^G	\a, \cG	Sino
00001000	10	8	8	BS	^H	[\b], \cH	Backspace
00001001	11	9	9	HT	^I	\t, \cI	Tabulação horizontal
00001010	12	10	0A	LF	^J	\n, \cJ	Mudança de linha
00001011	13	11	0B	VT	^K	\v, \cK	Tabulação vertical
00001100	14	12	0C	FF	^L	\f, \cL	Alimentação de formulário
00001101	15	13	0D	CR	^M	\r, \cM	Retorno de carro
00001110	16	14	0E	SO	^N	\cN	Shift out
00001111	17	15	0F	SI	^O	\cO	Shift in
00010000	20	16	10	DLE	^P	\cP	Data link escape
00010001	21	17	11	DC1	^Q	\cQ	Controle de dispositivo um (XON)

Binário	Oct	Dec	Hex	Carac	Tecl	Regex	Nome
00010010	22	18	12	DC2	^R	\cR	Controle de dispositivo dois
00010011	23	19	13	DC3	^S	\cS	Controle de dispositivo três (XOFF)
00010100	24	20	14	DC4	^T	\cT	Controle de dispositivo quatro
00010101	25	21	15	NAK	^U	\cU	Reconhecimento negativo
00010110	26	22	16	SYN	^V	\cV	Ocioso síncrono
00010111	27	23	17	ETB	^W	\cW	Fim de bloco de transmissão
00011000	30	24	18	CAN	^X	\cX	Cancelar
00011001	31	25	19	EM	^Y	\cY	Fim de mídia
00011010	32	26	1A	SUB	^Z	\cZ	Substituir
00011011	33	27	1B	ESC	^[\e, \c[Escape
00011100	34	28	1C	FS	^	\c	Separador de arquivo
00011101	35	29	1D	GS	^]	\c]	Separador de grupo
00011110	36	30	1E	RS	^^	\c^	Separador de registro
00011111	37	31	1F	US	^_	\c_	Separador de unidade
00100000	40	32	20	SP	SP	\s, []	Espaço
00100001	41	33	21	!	!	!	Ponto de exclamação
00100010	42	34	22	“	“	“	Aspas

Binário	Oct	Dec	Hex	Carac	Tecl	Regex	Nome
00100011	43	35	23	#	#	#	Sustenido
00100100	44	36	24	\$	\$	\\$	Cifrão
00100101	45	37	25	%	%	%	Sinal de porcentagem
00100110	46	38	26	&	&	&	E comercial (ampersand)
00100111	47	39	27	'	'	'	Apóstrofo
00101000	50	40	28	(((, \((Abre parênteses
00101001	51	41	29))), \)	Fecha parênteses
00101010	52	42	2A	*	*	*	Asterisco
00101011	53	43	2B	+	+	+	Sinal de mais
00101100	54	44	2C	,	,	,	Vírgula
00101101	55	45	2D	-	-	-	Hífen
00101110	56	46	2E	.	.	\., [.]	Ponto
00101111	57	47	2F	/	/	/	Barra

Binário	Oct	Dec	Hex	Carac	Tecl	Regex	Nome
00110000	60	48	30	0	0	\d, [0]	Dígito zero
00110001	61	49	31	1	1	\d, [1]	Dígito um
00110010	62	50	32	2	2	\d, [2]	Dígito dois
00110011	63	51	33	3	3	\d, [3]	Dígito três
00110100	64	52	34	4	4	\d, [4]	Dígito quatro
00110101	65	53	35	5	5	\d, [5]	Dígito cinco
00110110	66	54	36	6	6	\d, [6]	Dígito seis
00110111	67	55	37	7	7	\d, [7]	Dígito sete
00111000	70	56	38	8	8	\d, [8]	Dígito oito
00111001	71	57	39	9	9	\d, [9]	Dígito nove
00111010	72	58	3A	:	:	:	Dois-pontos
00111011	73	59	3B	;	;	;	Ponto e vírgula
00111100	74	60	3C	<	<	<	Menor
00111101	75	61	3D	=	=	=	Igual
00111110	76	62	3E	>	>	>	Maior
00111111	77	63	3F	?	?	?	Ponto de interrogação
01000000	100	64	40	@	@	@	Arroba

Binário	Oct	Dec	Hex	Carac	Tecl	Regex	Nome
01000001	101	65	41	A	A	\w, [A]	Letra A maiúscula
01000010	102	66	42	B	B	\w, [B]	Letra B maiúscula
01000011	103	67	43	C	C	\w, [C]	Letra C maiúscula
01000100	104	68	44	D	D	\w, [D]	Letra D maiúscula
01000101	105	69	45	E	E	\w, [E]	Letra E maiúscula
01000110	106	70	46	F	F	\w, [F]	Letra F maiúscula
01000111	107	71	47	G	G	\w, [G]	Letra G maiúscula
01001000	110	72	48	H	H	\w, [H]	Letra H maiúscula
01001001	111	73	49	I	I	\w, [I]	Letra I maiúscula
01001010	112	74	4A	J	J	\w, [J]	Letra J maiúscula
01001011	113	75	4B	K	K	\w, [K]	Letra K maiúscula
01001100	114	76	4C	L	L	\w, [L]	Letra L maiúscula
01001101	115	77	4D	M	M	\w, [M]	Letra M maiúscula
01001110	116	78	4E	N	N	\w, [N]	Letra N maiúscula
01001111	117	79	4F	O	O	\w, [O]	Letra O maiúscula
01010000	120	80	50	P	P	\w, [P]	Letra P maiúscula
01010001	121	81	51	Q	Q	\w, [Q]	Letra Q maiúscula
01010010	122	82	52	R	R	\w, [R]	Letra R maiúscula
01010011	123	83	53	S	S	\w, [S]	Letra S maiúscula
01010100	124	84	54	T	T	\w, [T]	Letra T maiúscula

Binário	Oct	Dec	Hex	Carac	Tecl	Regex	Nome
01010101	125	85	55	U	U	\w, [U]	Letra U maiúscula
01010110	126	86	56	V	v	\w, [V]	Letra V maiúscula
01010111	127	87	57	W	w	\w, [W]	Letra W maiúscula
01011000	130	88	58	X	x	\w, [X]	Letra X maiúscula
01011001	131	89	59	Y	y	\w, [Y]	Letra Y maiúscula
01011010	132	90	5A	Z	z	\w, [Z]	Letra Z maiúscula
01011011	133	91	5B	[[\[Abre colchetes
01011100	134	92	5C	\	\	\	Barra invertida
01011101	135	93	5D]]	\]	Fecha colchetes
01011110	136	94	5E	^	^	^, [^]	Acento circunflexo
01011111	137	95	5F	_	_	, []	Underscore
00100000	140	96	60	`	`	\`	Acento grave
01100001	141	97	61	a	a	\w, [a]	Letra a minúscula
01100010	142	98	62	b	b	\w, [b]	Letra b minúscula
01100011	143	99	63	c	c	\w, [c]	Letra c minúscula
01100100	144	100	64	d	d	\w, [d]	Letra d minúscula

Binário	Oct	Dec	Hex	Carac	Tecl	Regex	Nome
01100101	145	101	65	e	e	\w, [e]	Letra e minúscula
01100110	146	102	66	f	f	\w, [f]	Letra f minúscula
01100111	147	103	67	g	g	\w, [g]	Letra g minúscula
01101000	150	104	68	h	h	\w, [h]	Letra h minúscula
01101001	151	105	69	i	i	\w, [i]	Letra i minúscula
01101010	152	106	6A	j	j	\w, [j]	Letra j minúscula
01101011	153	107	6B	k	k	\w, [k]	Letra k minúscula
01101100	154	108	6C	l	l	\w, [l]	Letra l minúscula
01101101	155	109	6D	m	m	\w, [m]	Letra m minúscula
01101110	156	110	6E	n	n	\w, [n]	Letra n minúscula
01101111	157	111	6F	o	o	\w, [o]	Letra o minúscula
01110000	160	112	70	p	p	\w, [p]	Letra p minúscula
01110001	161	113	71	q	q	\w, [q]	Letra q minúscula
01110010	162	114	72	r	r	\w, [r]	Letra r minúscula
01110011	163	115	73	s	s	\w, [s]	Letra s minúscula
01110100	164	116	74	t	t	\w, [t]	Letra t minúscula
01110101	165	117	75	u	u	\w, [u]	Letra u minúscula
01110110	166	118	76	v	v	\w, [v]	Letra v minúscula
01110111	167	119	77	w	w	\w, [w]	Letra w minúscula
01111000	170	120	78	x	x	\w, [x]	Letra x minúscula

Binário	Oct	Dec	Hex	Carac	Tecl	Regex	Nome
01111001	171	121	79	y	y	\w, [y]	Letra y minúscula
01111010	172	122	7A	z	z	\w, [z]	Letra z minúscula
01111011	173	123	7B	{	{	{	Abre chaves
01111100	174	124	7C				Barra vertical
01111101	175	125	7D	}	}	}	Fecha chaves
01111110	176	126	7E	~	~	\~	Til
01111111	177	127	7F	DEL	^?	\c?	Apagar

Notas técnicas

Você pode acessar o memorando e o manual do QED, de Ken Thompson e Dennis Ritchie, no endereço <http://cm.bell-labs.com/cm/cs/who/dmr/qedman.pdf>.

Glossário de expressões regulares

alternância

Separação de uma lista de expressões regulares usando uma barra vertical (`|`), indicando *ou*. Em outras palavras, corresponda a qualquer uma das expressões regulares separadas por um ou mais caracteres `|`. Em algumas aplicações, como, por exemplo, no *grep* ou no *sed*, que usam expressões regulares básicas (BREs), o caractere `|` vem precedido por uma barra invertida, como em `\|`. *Ver também* expressões regulares básicas.

âncora

Especifica um local em uma linha ou string. Por exemplo, o circunflexo (`^`) significa o início de uma linha ou de uma string de caracteres, e o cifrão (`$`) indica o final de uma linha ou de uma string.

ASCII

American Standard Code for Information Interchange. Um esquema de codificação que usa 128 caracteres para caracteres do inglês (latinos), desenvolvido na década de 1960. *Ver também* Unicode.

asserções

Ver asserções de largura zero.

asserções de largura zero

Bordas que não consomem nenhum caractere em uma correspondência. São exemplos desse tipo de asserção `^` e `$`, que correspondem ao início e ao fim de uma linha, respectivamente.

átomo

Ver metacaractere.

backtracking

Dar um passo para trás, caractere por caractere, no decorrer de uma tentativa de encontrar uma correspondência bem-sucedida. Usado em uma correspondência gulosa, mas não em uma

correspondência preguiçosa ou possessiva. O backtracking catastrófico ocorre quando um processador de regex faz talvez milhares de tentativas para efetuar uma correspondência e consome uma enorme quantidade (leia-se *a maior parte*) dos recursos de processamento disponíveis. Uma maneira de evitar o backtracking catastrófico é usando agrupamento atômico. *Ver também* grupo atômico, correspondência gulosa, correspondência preguiçosa e correspondência possessiva.

backtracking catastrófico

Ver backtracking.

branch (galho)

Uma concatenação de partes em uma expressão regular na terminologia POSIX.1. *Ver também* POSIX.

BREs

Ver expressões regulares básicas.

buffer de armazenamento

Ver espaço de armazenamento.

buffer de trabalho

Ver espaço de padrão.

caracteres octais

Um caractere pode ser representado por meio de uma notação octal em expressões regulares. Em expressões regulares, um dado caractere no formato octal é especificado como `[\oxx]\o_xx_`, no qual `x` representa um número no intervalo de 1–9, usando uma ou duas posições. Por exemplo, `\o` representa o caractere `é`, a letra *e* latina minúscula com acento agudo.

classe de caracteres

Geralmente, um conjunto de caracteres entre colchetes; por exemplo, `[a-zA-B0-9]` é uma classe de caracteres para todas as letras, maiúsculas e minúsculas, mais os dígitos no conjunto de caracteres ASCII ou no Latim Básico.

code point

Ver Unicode.

composicionalidade

“Uma linguagem de esquema (ou uma linguagem de programação) disponibiliza diversos objetos atômicos e diversos métodos de composição. Os métodos de composição podem ser usados para combinar objetos atômicos, formando objetos compostos, os quais, por sua vez, podem ser combinados para formar outros objetos compostos. A composicionalidade da linguagem é o grau segundo o qual os vários métodos de composição podem ser aplicados uniformemente para todos os diversos objetos da linguagem, tanto atômicos quanto compostos. A composicionalidade facilita o aprendizado e o uso. A composicionalidade também faz com que haja tendência de melhoria na razão entre complexidade e poder: para um dado nível de complexidade, uma linguagem que permite mais composição será mais poderosa do que uma linguagem que permite menos composição.” De James Clark, “The Design of RELAX NG”, <http://www.thaiopensource.com/relaxng/design.html#section:5>.

conjunto de caracteres

Ver classe de caracteres.

correspondência

Uma expressão regular pode corresponder a um dado padrão no texto e, então, dependendo da aplicação, disparar um resultado.

correspondência gulosa (greedy) Uma correspondência gulosa consome o máximo possível de uma string-alvo e depois faz **backtracking** na string na tentativa de encontrar uma **correspondência**. Ver **backtracking**, correspondência preguiçosa, correspondência possessiva.

correspondência possessiva (possessive) Uma correspondência possessiva consome toda uma string de assunto de uma só vez na tentativa de encontrar uma **correspondência**. Não faz **backtracking**. Ver *também* **backtracking**, correspondência gulosa, correspondência preguiçosa.

correspondência preguiçosa (lazy) Uma correspondência preguiçosa consome uma string de assunto, um caractere de cada vez, na tentativa de efetuar uma correspondência. O **backtracking** não é feito. *Ver também* backtracking, correspondência gulosa, correspondência possessiva.

ed

O editor de linha do Unix criado por Ken Thompson em 1971 que implementava expressões regulares. Foi um precursor do *sed* e do *vi*.

EREs

Ver expressões regulares estendidas.

escape de caractere

Um caractere precedido por uma barra invertida. Por exemplo: `\t` (tabulação horizontal), `\v` (tabulação vertical) e `\f` (alimentação de formulário).

espaço de armazenamento

Usado pelo *sed* para armazenar uma ou mais linhas para processamento posterior. Também chamado de *buffer de armazenamento*. *Ver também* espaço de padrão, *sed*.

espaço de padrão

O programa *sed* normalmente processa uma linha de cada vez como entrada. À medida que cada linha é processada, ela é colocada em uma área que se chama *espaço de padrão*, na qual os padrões serão aplicados. Também é conhecido como *buffer de trabalho*. *Ver também* espaço de armazenamento, *sed*.

expressão entre colchetes

Uma expressão regular especificada entre colchetes; por exemplo, `[a-f]`, ou seja, o intervalo de letras minúsculas que vai de *a* até *f*. *Ver também* classe de caracteres.

expressão regular

Uma string de caracteres especialmente codificada que, quando

usada em uma aplicação ou em um utilitário, pode corresponder a outras strings ou a outros conjuntos de strings. Descrita inicialmente no início da década de 1950 pelo matemático Stephen Kleene (1909–1994) em seu trabalho com teoria de linguagens formais em seu livro *Introduction to Metamathematics*, publicado em 1952. Começou a ganhar destaque nas ciências da computação com o trabalho de Ken Thompson, *et al.* no editor QED (no *General Electric Time Sharing System* [GE-TSS] em um computador GE-635) e, posteriormente, em outras ferramentas no sistema operacional Unix da AT&T Bell Labs no início da década de 1970.

expressões regulares básicas Uma antiga implementação das expressões regulares, que é menos avançada e considerada obsoleta pela maioria. Também chamada de *BREs*. As *BREs* exigiam que determinados caracteres fossem escapados para que funcionassem como metacaracteres, como, por exemplo, as chaves (`\` e `\\`). *Ver também* expressões regulares estendidas.

expressões regulares estendidas As expressões regulares estendidas ou *EREs* adicionaram funcionalidades extras às expressões regulares básicas ou *BREs*, tais como alternância (`|`) e quantificadores como `?` e `+`, os quais funcionam com *egrep* (extended *grep*). Esses recursos novos foram definidos no padrão POSIX 1003.2-1992 do IEEE. Você pode usar a opção `-E` com o *grep* (é o mesmo que usar *egrep*), o que significa que você quer usar as expressões regulares estendidas, e não as expressões regulares básicas. *Ver também* alternância, expressões regulares básicas, *grep*.

flag

Ver modificador.

grep

Um utilitário de linha de comando do Unix para busca de strings usando expressões regulares. Criado por Ken Thompson em 1973, diz-se que o *grep* nasceu a partir do comando `g/re/p` (global/regular expression/print) do editor *ed*. Superado, mas não eliminado pelo

egrep (ou grep -E – que possui metacaracteres adicionais, como |, +, ?, (e) –, o grep usa expressões regulares básicas, enquanto o grep -E ou egrep usa expressões regulares estendidas). O fgrep (grep -F) faz busca em arquivos usando strings literais; metacaracteres como \$, * e | não possuem significados especiais. *Ver também* expressões regulares básicas, expressões regulares estendidas.

grupo atômico

Um grupamento que desabilita o backtracking quando uma expressão regular dentro de (?>...) falha ao efetuar a correspondência. *Ver também* backtracking, grupos.

grupo de captura

Ver grupos.

grupo de não-captura

Um grupo dentro de parênteses que não é capturado (ou seja, não é armazenado em memória para uso futuro). A sintaxe para um grupo de não-captura é (?:padrão). *Ver também* grupos.

grupos

Os grupos combinam átomos de expressões regulares dentro de parênteses, (). Em algumas aplicações, como *grep* ou *sed* (sem -E), é preciso colocar uma barra invertida antes dos parênteses, como em \) ou \(. Há grupos de captura e grupos de não-captura. Um grupo de captura armazena o grupo capturado na memória, de modo que esse pode ser reutilizado, enquanto um grupo de não-captura não faz isso. Grupos atômicos não fazem backtracking. *Ver também* grupo atômico.

hexadecimal

Um sistema de numeração de base 16 representado pelos dígitos de 0 a 9 e pelas letras de A até F ou a até f. Por exemplo, o número decimal 15 é representado como F no sistema hexadecimal, e o decimal 16 corresponde a 10 em hexadecimal.

limitante (bound)

Ver quantificador.

literal

Ver string literal.

lookahead

Uma expressão regular que efetua correspondência somente se outra expressão regular especificada seguir-se à primeira. Um lookahead positivo usa a sintaxe `regex(?=regex)`. Um lookahead negativo significa que a expressão regular *não* é seguida pela expressão regular que se segue à primeira. A sintaxe `regex(?!regex)` é utilizada.

lookbehind negativo

Ver lookbehind.

lookahead positivo

Ver lookahead.

lookaround

Ver lookahead, lookbehind.

lookbehind

Uma expressão regular que efetua correspondência somente se outra expressão regular especificada vier antes da primeira. Um lookbehind positivo usa a sintaxe `regex(?<=regex)`. Um lookbehind negativo significa que a expressão regular *não* vem seguida pela expressão regular que antecede a primeira. A sintaxe `regex(?<!regex)` é utilizada.

lookahead negativo

Ver lookahead.

lookbehind positivo

Ver lookbehind.

metacaractere

Um caractere que tem um significado especial em expressões regulares. Esses caracteres são (as vírgulas desta lista são separadores) `.`, `\`, `|`, `*`, `+`, `?`, `~`, `$`, `[`, `]`, `(`, `)`, `{`, `}`. Metacaracteres também são chamados de *átomos*.

modificador

Um caractere colocado depois de um padrão de correspondência ou de substituição que modifica o processo de correspondência. Por exemplo, o modificador *i* faz com que não haja diferenciação entre letras maiúsculas e minúsculas na correspondência. Também chamado de *flag*.

negação

Indica que uma expressão regular não corresponde a um dado padrão. Especificado no interior de classes de caracteres com um circunflexo no início, como em `[^2-7]`, que faz a correspondência com outros dígitos que não 2, 3, 4, 5, 6 ou 7 – ou seja, corresponde a 0, 1, 8, 9.

opções

Permite habilitar e desabilitar opções que modificam a correspondência. Por exemplo, a opção `(?i)` indica que a correspondência não levará em conta a diferença entre letras maiúsculas e minúsculas. Semelhantes aos modificadores, mas utilizam uma sintaxe diferente. *Ver também* modificador.

Perl

Uma linguagem de programação de uso geral, criada por Larry Wall em 1987, o Perl é conhecido por seu suporte poderoso a expressões regulares e por suas capacidades de processamento de textos. *Ver* <http://www.perl.org>.

piece (pedaço)

Uma porção de uma expressão regular, geralmente concatenada, na terminologia do POSIX.1. *Ver também* POSIX.

POSIX

Portable Operating System Interface for Unix. Uma família de padrões relacionados ao Unix, estabelecida pelo Institute of Electrical and Electronics Engineers (IEEE). O padrão POSIX mais recente para expressões regulares é o POSIX.1-2008 (ver <http://standards.ieee.org/findstds/standard/1003.1-2008.html>).

quantificador

Define o número de vezes que uma expressão regular pode ocorrer em uma tentativa de correspondência. Um número inteiro ou um par de números inteiros separados por vírgula, cercados por chaves, é uma das maneiras de especificá-lo; por exemplo, {3} indica que a expressão pode ocorrer exatamente três vezes (com ferramentas mais antigas que usam expressões regulares básicas, será necessário escapar as chaves, desta maneira: `\{3\}`).

Outros quantificadores incluem ? (zero ou uma vez), + (uma ou mais vezes) e * (zero ou mais vezes). Um quantificador também é chamado de *limitante* ou *modificador*. Por si só, os quantificadores são gulosos. Há também quantificadores preguiçosos (por exemplo, {3}?) e quantificadores possessivos (por exemplo, {3}+). *Ver também* expressões regulares básicas, correspondência gulosa, correspondência preguiçosa, correspondência possessiva.

restrição de ocorrências

Ver quantificador.

retrovisor (backreference) Refere-se a uma expressão regular anterior capturada com parênteses que usa uma referência no formato `\1`, `\2` e assim por diante.

sed

Um editor de streaming do Unix que aceita expressões regulares e transforma textos. Foi desenvolvido no início da década de 1970 por Lee McMahon no Bell Labs. Um exemplo de uso do `sed`: `sed -n 's/this/that/g' file.ext > new.ext`. Use `sed -E` para indicar que você deseja usar expressões regulares estendidas. *Ver também* expressões regulares estendidas.

string literal

Uma cadeia de caracteres interpretada literalmente – por exemplo, a string literal “It is an ancyent Marinere” em oposição a algo do tipo “[li]t[]is[].*nere.”

Unicode

O Unicode é um sistema para codificação de caracteres para os sistemas de escrita existentes no mundo. A cada caractere em Unicode atribui-se um *code point* numérico. Há mais de 100 mil caracteres representados em Unicode. Em expressões regulares, um caractere Unicode pode ser especificado na forma `\uxxxx` ou `\x{xxxx}`, em que *x* representa um número hexadecimal no intervalo 0–9, A–F (ou a–f), usando de uma a quatro posições. Por exemplo, `\u00E9` representa o caractere *é*, que é a letra latina e minúscula com acento agudo. *Ver também* o site <http://www.unicode.org>.

vi

Um editor do Unix, inicialmente desenvolvido em 1976 por Bill Joy, que usa expressões regulares. O editor *vim* é um substituto melhorado do *vi*, desenvolvido principalmente por Bram Moolenaar (ver <http://www.vim.org>). Atualmente, eu uso seis ou sete editores diferentes durante um dia normal de trabalho, mas aquele que eu uso com mais frequência é o *vim*. De fato, se eu naufragasse e ficasse perdido em uma ilha deserta e pudesse ter somente um editor de textos, eu escolheria o *vim*. Sem dúvida.

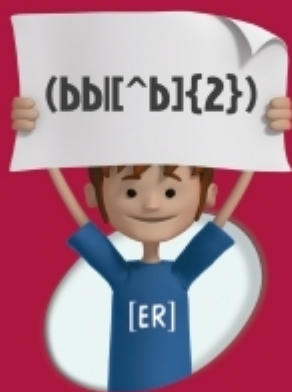
vim

Ver *vi*.

5ª EDIÇÃO
Revisada e ampliada

[EXPRESSÕES REGULARES]

UMA ABORDAGEM DIVERTIDA



novatec

Aurelio Marinho Jargas
www.aurelio.net

Expressões Regulares - 5ª edição

Jargas, Aurelio Marinho

9788575224755

248 páginas

[Compre agora e leia](#)

Você procura uma sigla em um texto longo, mas não lembra direito quais eram as letras. Só lembra que era uma sigla de quatro letras. Simples, procure por `[A-Z]{4}`. Revisando aquela tese de mestrado, você percebe que digitou errado o nome daquele pesquisador alemão famoso. E foram várias vezes. Escreveu Miller, Mueller e Müller, quando na verdade era Müller. Que tal corrigir todos de uma vez? Fácil, use a expressão `M(i|ue|ü)ll?er`. Que tal encontrar todas as palavras repetidas repetidas em seu texto? Ou garantir que há um espaço em branco após todas as vírgulas e os pontos finais? Se você é programador, seria bom validar dados em um único passo, não? Endereço de e-mail, número IP, telefone, data, CEP, CPF... Chega de percorrer vetores e fazer checagens "na mão". Estes são exemplos de uso das Expressões Regulares, que servem para encontrar rapidamente trechos de texto informando apenas o seu formato. Ou ainda pesquisar textos com variações, erros ortográficos e muito mais! Visite o site do livro: www.piazinho.com.br

[Compre agora e leia](#)

O'REILLY®

Padrões para Kubernetes

Elementos reutilizáveis no design de aplicações
nativas de nuvem



novatec

Bilgin Ibryam
Roland Huß

Padrões para Kubernetes

Ibryam, Bilgin

9788575228159

272 páginas

[Compre agora e leia](#)

O modo como os desenvolvedores projetam, desenvolvem e executam software mudou significativamente com a evolução dos microsserviços e dos contêineres. Essas arquiteturas modernas oferecem novas primitivas distribuídas que exigem um conjunto diferente de práticas, distinto daquele com o qual muitos desenvolvedores, líderes técnicos e arquitetos estão acostumados. Este guia apresenta padrões comuns e reutilizáveis, além de princípios para o design e a implementação de aplicações nativas de nuvem no Kubernetes. Cada padrão inclui uma descrição do problema e uma solução específica no Kubernetes. Todos os padrões acompanham e são demonstrados por exemplos concretos de código. Este livro é ideal para desenvolvedores e arquitetos que já tenham familiaridade com os conceitos básicos do Kubernetes, e que queiram aprender a solucionar desafios comuns no ambiente nativo de nuvem, usando padrões de projeto de uso comprovado. Você conhecerá as seguintes classes de padrões:

- Padrões básicos, que incluem princípios e práticas essenciais para desenvolver aplicações nativas de nuvem com base em contêineres.
- Padrões comportamentais, que exploram conceitos mais específicos para administrar contêineres e interações com a plataforma.
- Padrões estruturais, que ajudam você a organizar contêineres em um Pod para tratar casos de uso específicos.
- Padrões de configuração, que oferecem insights sobre como tratar as configurações das aplicações no Kubernetes.
- Padrões

avançados, que incluem assuntos mais complexos, como operadores e escalabilidade automática (autoscaling).

[Compre agora e leia](#)

CANDLESTICK

Um método para ampliar lucros na Bolsa de Valores



novatec

Carlos Alberto Debastiani

Candlestick

Debastiani, Carlos Alberto

9788575225943

200 páginas

[Compre agora e leia](#)

A análise dos gráficos de Candlestick é uma técnica amplamente utilizada pelos operadores de bolsas de valores no mundo inteiro. De origem japonesa, este refinado método avalia o comportamento do mercado, sendo muito eficaz na previsão de mudanças em tendências, o que permite desvendar fatores psicológicos por trás dos gráficos, incrementando a lucratividade dos investimentos. Candlestick – Um método para ampliar lucros na Bolsa de Valores é uma obra bem estruturada e totalmente ilustrada. A preocupação do autor em utilizar uma linguagem clara e acessível a torna leve e de fácil assimilação, mesmo para leigos. Cada padrão de análise abordado possui um modelo com sua figura clássica, facilitando a identificação. Depois das características, das peculiaridades e dos fatores psicológicos do padrão, é apresentado o gráfico de um caso real aplicado a uma ação negociada na Bovespa. Este livro possui, ainda, um índice resumido dos padrões para pesquisa rápida na utilização cotidiana.

[Compre agora e leia](#)



AVALIANDO
EMPRESAS

INVESTINDO EM AÇÕES

A APLICAÇÃO PRÁTICA DA
ANÁLISE FUNDAMENTALISTA NA
AVALIAÇÃO DE EMPRESAS

novatec

CARLOS ALBERTO DEBASTIANI
FELIPE AUGUSTO RUSSO

Avaliando Empresas, Investindo em Ações

Debastiani, Carlos Alberto

9788575225974

224 páginas

[Compre agora e leia](#)

Avaliando Empresas, Investindo em Ações é um livro destinado a investidores que desejam conhecer, em detalhes, os métodos de análise que integram a linha de trabalho da escola fundamentalista, trazendo ao leitor, em linguagem clara e acessível, o conhecimento profundo dos elementos necessários a uma análise criteriosa da saúde financeira das empresas, envolvendo indicadores de balanço e de mercado, análise de liquidez e dos riscos pertinentes a fatores setoriais e conjunturas econômicas nacional e internacional. Por meio de exemplos práticos e ilustrações, os autores exercitam os conceitos teóricos abordados, desde os fundamentos básicos da economia até a formulação de estratégias para investimentos de longo prazo.

[Compre agora e leia](#)

Marcos Abe

MANUAL DE ANÁLISE TÉCNICA

ESSÊNCIA E ESTRATÉGIAS AVANÇADAS

TUDO O QUE UM INVESTIDOR PRECISA SABER PARA
PROSPERAR NA BOLSA DE VALORES ATÉ EM TEMPOS DE CRISE

novatec

Manual de Análise Técnica

Abe, Marcos

9788575227022

256 páginas

[Compre agora e leia](#)

Este livro aborda o tema Investimento em Ações de maneira inédita e tem o objetivo de ensinar os investidores a lucrarem nas mais diversas condições do mercado, inclusive em tempos de crise. Ensinará ao leitor que, para ganhar dinheiro, não importa se o mercado está em alta ou em baixa, mas sim saber como operar em cada situação. Com o Manual de Análise Técnica o leitor aprenderá:

- os conceitos clássicos da Análise Técnica de forma diferenciada, de maneira que assimile não só os princípios, mas que desenvolva o raciocínio necessário para utilizar os gráficos como meio de interpretar os movimentos da massa de investidores do mercado;
- identificar oportunidades para lucrar na bolsa de valores, a longo e curto prazo, até mesmo em mercados baixistas;
- um sistema de investimentos completo com estratégias para abrir, conduzir e fechar operações, de forma que seja possível maximizar lucros e minimizar prejuízos;
- estruturar e proteger operações por meio do gerenciamento de capital.

Destina-se a iniciantes na bolsa de valores e investidores que ainda não desenvolveram uma metodologia própria para operar lucrativamente.

[Compre agora e leia](#)