

5ª EDIÇÃO  
Revisada e ampliada

# [EXPRESSÕES REGULARES]

UMA ABORDAGEM DIVERTIDA



novatec

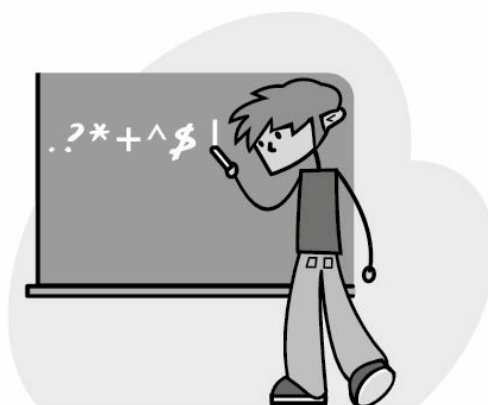
Aurelio Marinho Jargas  
[www.aurelio.net](http://www.aurelio.net)

# [EXPRESSÕES REGULARES]

UMA ABORDAGEM DIVERTIDA

**5ª Edição – Revisada e Ampliada**

**Aurelio Marinho Jargas**



**novatec**

Copyright © 2006, 2008, 2009, 2012, 2016 da Novatec Editora Ltda.  
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

Assistente editorial: Priscila A. Yoshimatsu

Revisão de textos: Marta Almeida de Sá

Capa, projeto gráfico e editoração eletrônica: Karine Hermes

Adaptação do projeto gráfico: Carolina Kuwabata

Ilustração da capa: Vinicius Vogel

ISBN: 978-85-7522-475-5

Histórico de edições impressas:

Janeiro/2016 Quinta edição (ISBN: 978-85-7522-474-8)

Outubro/2012 Quarta edição (ISBN: 978-85-7522-337-6)

Setembro/2009 Terceira edição (ISBN: 978-85-7522-212-6)

Julho/2008 Segunda edição (ISBN: 978-85-7522-173-0)

Novembro/2006 Primeira edição (ISBN: 85-7522-100-0)

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](mailto:novatec@novatec.com.br)

Site: [www.novatec.com.br](http://www.novatec.com.br)

Twitter: [twitter.com/novateceditora](https://twitter.com/novateceditora)

Facebook: [facebook.com/novatec](https://facebook.com/novatec)

LinkedIn: [linkedin.com/in/novatec](https://linkedin.com/in/novatec)

*Para Wanderlei Antonio Cavassin,  
Que um dia virou-se e perguntou:  
Por que você não escreve um livro?*

# Sumário

## [Agradecimentos](#)

## [Sobre o autor](#)

## [Prefácio](#)

## [Capítulo 1 • Introdução](#)

[Objetivo](#)

[Sobre o livro](#)

[Apresentando as Expressões Regulares](#)

[História](#)

[Terminologia](#)

[Para que servem?](#)

## [Capítulo 2 • Os metacaracteres](#)

[Metacaracteres tipo Representante](#)

[Ponto: o necessitado .](#)

[Lista: a exigente \[ . . . \]](#)

[Lista negada: a experiente \[ ^ . . . \]](#)

[Metacaracteres tipo Quantificador](#)

[Opcional: o opcional ?](#)

[Asterisco: o tanto-faz \\*](#)

[Mais: o tem-que-ter +](#)

[Chaves: o controle { \*n\* , \*m\* }](#)

[Metacaracteres tipo Âncora](#)

[Circunflexo: o início ^](#)

[Cifrão: o fim \\$](#)

[Borda: a limítrofe \b](#)

[Outros metacaracteres](#)

[Escape: a criptonita \](#)

[Ou: o alternativo |](#)

[Grupo: o pop \( . . . \)](#)

[Retrovisor: o saudosista \1 . . . \9](#)

## **Capítulo 3 • Mais sobre metacaracteres**

[Épocas e aplicativos diversos, metacaracteres distorcidos](#)

[Quantificadores gulosos](#)

[Quantificadores não-gulosos](#)

[Metacaracteres tipo barra-letra](#)

[Metacaracteres modernos](#)

[Precedência entre metacaracteres](#)

## **Capítulo 4 • Os 6 mandamentos do Criador**

[Não complique](#)

[Use o circunflexo](#)

[Evite a lista negada](#)

[Evite o curinga](#)

[Seja específico](#)

[Não seja afobado, seja ninja](#)

## **Capítulo 5 • Como lidar com...**

[Problemas com maiúsculas e minúsculas](#)

[ERs pré-processadas e cruas](#)

[Multilinha](#)

[Acentuação](#)

## **Capítulo 6 • Editores de texto e planilhas**

[Emacs](#)

[Google Docs: Planilhas](#)

[Microsoft Word](#)

[Notepad++](#)

[\(Libre|Br|Open\)Office Writer](#)

[\(Libre|Br|Open\)Office Calc](#)

[Vim](#)

## **Capítulo 7 • Aplicativos**

[Apache HTTPd](#)

[Find](#)

[Grep](#)

[Nginx](#)

## **Capítulo 8 • Linguagens de programação**

[Awk](#)

[C](#)

[HTML5](#)

[Java](#)

[JavaScript / ActionScript](#)

[Lua](#)

[.NET](#)

[Perl](#)

[PHP \(PCRE\)](#)

[PHP \(POSIX\)](#)

[PowerShell](#)

[Python](#)

[Ruby](#)

[Sed](#)

[Shell Script \(Bash\)](#)

[Tcl](#)

[VBscript](#)

## **Capítulo 9 • Bancos de dados**

[MySQL](#)

[Oracle](#)

[PostgreSQL](#)

[SQLite](#)

## **Capítulo 10 • Bibliotecas e programas relacionados**

[Bibliotecas](#)

[Programas](#)

## **Apêndice A • Onde obter mais informações**

[Livros](#)

[Fóruns](#)

[Internet](#)

## **Apêndice B • Tabelas**

[Diferenças de metacaracteres entre aplicativos](#)

[Resumão dos metacaracteres e seus detalhes](#)

[POSIX, barra-letras e outros aliens](#)

[Modernosos, remendos e precedência](#)

[Caracteres ASCII imprimíveis \(ISO-8859-1\)](#)

# Agradecimentos

À Mog, pelo amor, inspiração, incentivo e dedicação.

À minha família: Geny, Karla, Gabriel e Felipe, pelo apoio incondicional.

Ao Julio Cezar Neves, pelo bom-humor que inspira.

Ao Wanderlei Antonio Cavassin, pela pergunta que deu início ao livro.

Ao Osvaldo Santana Neto, pela ajuda e pelo incentivo quando este livro ainda era somente uma possibilidade.

À Conectiva, pelo conhecimento e pela dispensa no trabalho durante os dias de arremate da primeira versão do livro.

Ao Rubens Prates (Novatec), pelo apoio e entusiasmo desde o primeiro e-mail trocado, pelos conselhos durante a escrita e pela total transparência em todo o processo de publicação do livro.

Ao Rubens Queiroz (Dicas-L) e ao Augusto Campos (BR-Linux), pela divulgação.

Ao Guaracy Monteiro, pelas informações sobre a linguagem Ruby e ao Thobias Salazar Trevisan, pelas informações sobre a linguagem C.

Ao Cristóferon Bueno e ao Flavio Gabriel Duarte, pelas informações sobre o HTML 5, e ao Denis Brandl e José Victor Macedo pelos testes com o Internet Explorer.

A Felipe Rossi, Gustavo Figueira e ao Lindolfo Rodrigues pelos testes no Oracle.

Ao Ricardo Felipe Klein por ceder seu servidor para meus testes com o Apache.



A Celso Fernandes, Felipe Bonifácio, Osvaldo Santana Neto e Vinicius Moreira Mello, pelas informações sobre o Nginx.

A todos que acompanharam “ao vivo” no Twitter a escrita desta edição, em especial àqueles que participaram com ideias e sugestões, foi muito empolgante a experiência.

Aos que contribuíram para a melhoria do livro: Alexandre Frey, Allan Koch Veiga, Ana Claudia Nogueira, Arnaldo Ribeiro, Bruno Gastaldi, Caio Felipe Correa, Caio Graco Pereira Santos, Carlos Alberto Teixeira, Cassyus Pereira Lobo, Danilo Marques, Denilson Figueiredo de Sá, Denis de Azevedo, Elias Júnior, Fábio Emilio Costa, Fabricio Beltram, Francival Lima, Guilherme Fonseca, José Ricardo Almeida de Britto Filho, João Alberto de Oliveira Lima, Julio Cezar Neves, Karlisson de Macêdo Bezerra, Klayson Bonatto, Leslie Harlley Watter, Luciano Silveira do Espírito Santo, Marcelo Assis, Marcio Marchini, Max Fernandes Moura, Meleu, Rafael Colatusso, Renato Abel Abrahão, Richard Sachsse, Rodolfo Pilas, Rodrigo Stulzer Lopes, Rogério Brito, Sérgio Fischer e Tales Pinheiro de Andrade.

A Deus, por tudo.



# Capítulo 1

## Introdução

Olá. Que tal esquecer um pouco a rotina e a realidade e fazer uma viagem ao interior de sua mente? Descobrir conceitos novos, diferentes. Ao voltar, as coisas não serão mais tão normais quanto antes, pois símbolos estranhos farão parte de seu dia a dia.

Inspirado pelo funcionamento de seus próprios neurônios, descubra o fascinante mundo abstrato das expressões regulares.

### Objetivo

Neste nosso mundo tecnoinformatizado, onde o acesso rápido à informação desejada é algo crucial, temos nas expressões regulares uma mão amiga, que quanto mais refinada for sua construção, mais preciso e rápido será o resultado, diferenciando aqueles que as dominam daqueles que perdem horas procurando por dados que estão ao alcance da mão.

O assunto é algo bem peculiar, pois apesar de a maioria das linguagens de programação, programas e editores de texto mais utilizados possuírem esse recurso, poucos o dominam, principalmente pelo fato de a documentação sobre o assunto, quando existente, ser enigmática e pouco didática, ou simplesmente se resumir a listagens, sem explicar os conceitos. Esta obra nasceu dessa necessidade e tem como objetivo preencher essa lacuna, sendo

uma documentação completa e didática para iniciantes, tipo tutorial, e um guia de referência para os já iniciados.

Este livro é a primeira publicação em português totalmente dedicada ao assunto, e espero que esse pioneirismo traga muitos frutos, inclusive outras publicações sobre o tema, para difundir e desmistificar o uso das expressões regulares.

## Sobre o livro

A primeira parte é o “feijão com arroz”, indicada àqueles que desconhecem ou ainda não se sentem à vontade para criar suas próprias expressões regulares. Faremos um *tour* por todas as pecinhas que compõem esse mundo fantástico, explicando didaticamente, do zero, o que são, de onde vieram, para que servem e como utilizá-las (Exemplos! Exemplos!).

Após ler e entender essa primeira parte, algo como

```
^[A-Za-z0-9_]+:(.*)$
```

vai fazer parte de sua realidade, sem lhe causar pânico.

A segunda parte é a “feijoada”, para aqueles que querem uma experiência mais intensa. Mergulharemos de cabeça e entenderemos de vez essa maquininha esquisita. São as explicações dos conceitos envolvidos, bem como táticas e dicas para você realmente entender e otimizar seu uso das expressões regulares. Ao final da leitura, você entenderá por que

```
^[^:]+:([A-Za-z]+):
```

é melhor que

```
.*: (.*):
```

Mas note que, tudo isso, sem viajar muito nos detalhes intrínsecos e sem conhecer os becos escuros que você talvez nunca precisará saber que existem. Acima de tudo este é um livro prático. É para ler e fazer suas expressões. Isso não o torna superficial, apenas direto.

Com tudo isso, temos diversas tabelas e listagens que servem para ser consultadas rapidamente em caso de dúvida ou esquecimento. Relaxe, não é

um bicho de  $[0-9]^+$  cabeças... Vamos bater um papo descontraído sobre o assunto. Então respire fundo, desligue a TV, olhe fixamente para estas letras e vamos!

## **Apresentando as Expressões Regulares**

Então, para podermos começar nossa viagem, nada como uma apresentação de nosso assunto, pois, afinal de contas, que raios são estas expressões?

Bem resumido, uma expressão regular é um método formal de se especificar um padrão de texto.

Mais detalhadamente, é uma composição de símbolos, caracteres com funções especiais, que, agrupados entre si e com caracteres literais, formam uma sequência, uma expressão. Essa expressão é interpretada como uma regra que indicará sucesso se uma entrada de dados qualquer casar com essa regra, ou seja, obedecer exatamente a todas as suas condições.

Ou como variações aceitas também pode-se afirmar que é:

- uma maneira de procurar um texto que você não lembra exatamente como é, mas tem ideia das variações possíveis;
- uma maneira de procurar um trecho em posições específicas como no começo ou no fim de uma linha, ou palavra;
- uma maneira de um programador especificar padrões complexos que podem ser procurados e casados em uma cadeia de caracteres;
- uma construção que utiliza pequenas ferramentas, feita para obter determinada sequência de caracteres de um texto.

Ou ainda, didaticamente falando, é:

- Como o brinquedo LEGO, várias pecinhas diferentes, cada uma com sua característica, que juntas compõem estruturas completas e podem ser arranjadas com infinitas combinações diferentes.
- Como um jogo de truco, com as cartas normais e as quentes: gato, copas, espadilha e mole, que são especiais e têm uma ordem de grandeza.

- Como um quebra-cabeça, sempre tem solução, às vezes óbvia, às vezes difícil, mas, decifrando as partes, junta-se tudo e chega-se ao todo.
- Como um jogo, no começo é difícil, mas após conhecer todas as regras, basta jogar e curtir.
- Como uma receita culinária, com seus ingredientes e uma ordem correta para adicioná-los à mistura.
- Como consertar carros. Você tem várias peças e várias ferramentas. Dependendo do tipo de peça, há uma ferramenta certa para você lidar com ela. E dependendo da sua localização, você tem de incrementar a ferramenta com mais barras e cotovelos para alcançá-la.
- Como o alfabeto. Você aprende primeiro as letras individualmente. Depois as sílabas, as palavras, as frases e finalmente os textos. Mas no fundo são apenas letras.

Acima de tudo, assim como um sorvete no domingo ensolarado, uma expressão regular é:

Divertida!

**Divertida? Tá louco?**  
**Todos aqueles símbolos**  
**estranhos...**



Calma... É normal estranharmos ou até repudiarmos aquilo que ainda não conhecemos ou não dominamos bem. Como diria o vovô Simpson no meio da multidão: “Vamos destruir aquilo que não entendemos!”.

Ao final da leitura, ficará claro que as expressões são apenas pequenos pedacinhos simples que agrupados formam algo maior. O importante é você compreender bem cada um individualmente, e depois apenas lê-los em sequência. Lembre-se do alfabeto: são apenas letras...

# História



Sim! Vou te contar uma história. A fecundação dessas expressões aconteceu no ano de 1943, quando os “pais”, dois neurologistas, publicaram um estudo que teorizava o funcionamento dos nossos neurônios. Sentiu o drama? Nosso assunto é nobre desde a sua origem.

Anos depois, o “parteiro”, um matemático, descreveu algebricamente os modelos desse estudo utilizando símbolos para representar seus recém-criados grupos regulares (do inglês “regular sets”). Com a criação dessa notação simbólica, nasceram as expressões regulares, que durante toda a sua infância e juventude (cerca de vinte anos) foram muito estudadas pelos matemáticos da época.

Mas o encontro com o computador só aconteceu mesmo em 1968, em um algoritmo de busca utilizado no editor de textos `qed`, que depois virou o `ed`, editor padrão dos primeiros sistemas Unix. Este `ed` tinha o comando de contexto `g`, que aceitava expressões regulares e um comando `p`. Sua sintaxe ficava `g/RE/p` (“Global Regular Expression Print”), que deu origem ao aplicativo `grep`, que por sua vez originou o `egrep`.

Outros filhos como o `sed` e o `awk` também apareceram, cada um implementando as expressões do seu próprio jeito; e finalmente em 1986 foi criado o divisor de águas, um pacote pioneiro em C chamado `regex` que tratava das expressões regulares e qualquer um poderia incluí-lo em seu próprio programa, de graça. Opa! Falaram as palavras mágicas: de graça. Aí não teve mais volta, as expressões caíram no gosto popular e cada vez mais e mais programas e linguagens as utilizam.

**Curiosidade:** apesar de esse assunto ser antigo, o que vamos ver aqui basicamente é o mesmo que um estudante veria 25 anos atrás. É um conceito consistente, que não sofre alterações com o passar do tempo.

## Terminologia

E se eu te disser que “ERs são metacaracteres que casam um padrão”? Não entendeu?

Bem, como expressões regulares é um termo muito extenso, daqui em diante, chamarei apenas de ER (ê-érre) para simplificar a leitura. Outras nomenclaturas que podem ser encontradas em outras fontes são expreg, regexp, regex e RE. Particularmente, regex é uma boa escolha para usar em ferramentas de busca na Internet.

E como estamos falando de termos, tem mais alguns novos que farão parte de nossa conversa. Lembra que as expressões são formadas por símbolos e caracteres literais? Esses símbolos são chamados de metacaracteres, pois possuem funções especiais, que veremos detalhadamente adiante.

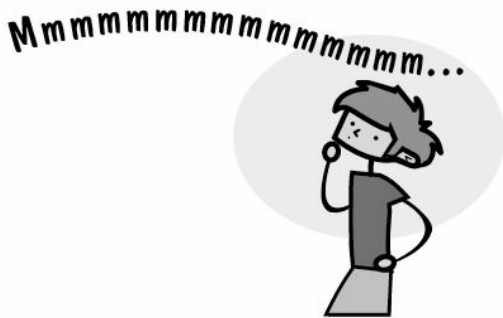
Outro termo que é interessante e às vezes pode assustar um leitor meio distraído é o casar (“match”). Casamento aqui não é juntar os trapos, mas, sim, o ato de bater, conferir, combinar, igualar, encontrar, encaixar, equiparar. É como em um caixa 24 horas, em que você só retirará o dinheiro se sua senha digitada casar com aquela já cadastrada no banco.

Também temos o padrão (“pattern”), que é nosso objetivo quando fazemos uma ER: casar um padrão. Esse padrão pode ser uma palavra, várias, uma linha vazia, um número, ou seja, o que quer que precise ser encontrado pela nossa ER.

E ainda tem o robozinho, que é uma referência ao compilador e interpretador das expressões regulares, o código que vai ler, checar, entender e aplicar sua ER no texto desejado. Como exemplo, para programas em C o robozinho é a biblioteca regex, que faz todo o serviço.

## Para que servem?

Basicamente servem para você dizer algo abrangente de forma específica. Definido seu padrão de busca, você tem uma lista (finita ou não) de possibilidades de casamento. Em um exemplo rápido, [rgp]ato pode casar rato, gato e pato. Ou seja, sua lista “abrange especificamente” essas três palavras, nada mais.



Na prática, as expressões regulares servem para uma infinidade de tarefas, é difícil fazer uma lista, pois elas são úteis sempre que você precisar buscar ou validar um padrão de texto que pode ser variável, como:

- data
- horário
- número IP
- nome de pessoa
- endereço de e-mail
- endereço de Internet
- nome de usuário e senha
- declaração de uma função()
- dados na coluna N de um texto
- dados que estão entre <tags></tags>
- campos específicos de um texto tabulado
- número de telefone, RG, CPF, cartão de crédito



- dados que estão apenas no começo ou no fim da linha

E mais uma infinidade de outros padrões que não podem ser especificados com caracteres literais.

Um exemplo prático: você tem uma lista diária de acesso de usuários que entraram em seu sistema, onde consta, em cada linha, o horário do acesso e o login do usuário, algo como:

```
05:15  ernesto  
08:39  ricardo  
10:32  patricia  
14:59  gabriel  
16:27  carla  
22:23  marcelo
```

Como fazer para buscar automaticamente apenas os usuários que acessaram o sistema no período da tarde (meio-dia às seis)? Você tem várias opções, desde procurar uma a uma manualmente até fazer um programa que compare os primeiros caracteres de cada linha, mas, falando de algo prático e rápido, que não exija conhecimentos de programação, a ER é simplesmente  $^1[2-8]$ .



Caaaaaaalma. Acompanhe o próximo tópico e vamos conhecer todos os metacaracteres, essas coisinhas úteis que facilitam nossa vida.



## Capítulo 2

# Os metacaracteres

Então, para já matar sua curiosidade, aqui estão os tão falados metacaracteres-padrão que serão nossos personagens das próximas páginas:

. ? \* + ^ \$ | [ ] { } ( ) \

E aí, sentiu um frio na barriga? Cada simbolozinho desses tem sua função específica, que pode mudar dependendo do contexto no qual está inserido, e podemos agregá-los uns com os outros, combinando suas funções e fazendo construções mais complexas. Olha, ainda dá tempo de fechar o livro e voltar a assistir à novela...

Então deixe eu te assustar mais um pouquinho. Além destes, temos outros metacaracteres estendidos que foram criados posteriormente, pois tarefas mais complexas requisitavam funções mais específicas ainda. E para terminar de complicar, sua sintaxe de utilização não é a mesma para todos os programas que suportam expressões regulares.

Bem, já que você não desistiu (eu tentei), vamos logo ao que interessa, e para começar vamos dar nomes aos bois. Leia, releia e treleia esta lista, repetindo para si mesmo e associando o nome ao símbolo, pois estas palavras farão parte de sua vida, de sua rotina. Acostume-se com os nomes e não os mude.

Metacaractere	Nome	Metacaractere	Nome
---------------	------	---------------	------

.	Ponto	^	Circunflexo
[ ]	Lista	\$	Cifrão
[^]	Lista negada	\b	Borda
?	Opcional	\	Escape
*	Asterisco		Ou
+	Mais	( )	Grupo
{ }	Chaves	\1	Retrovisor

Agora que sabemos como chamar nossos amigos novos, veremos uma prévia, um apanhado geral de todos os metacaracteres e suas funções. Eles estão divididos em quatro grupos distintos, de acordo com características comuns entre eles.

## Representantes

Metacaractere	Nome	Função
.	Ponto	Um caractere qualquer
[ . . . ]	Lista	Lista de caracteres permitidos
[ ^ . . . ]	Lista negada	Lista de caracteres proibidos

## Quantificadores

Metacaractere	Nome	Função
?	Opcional	Zero ou um
*	Asterisco	Zero, um ou mais
+	Mais	Um ou mais
{ <i>n</i> , <i>m</i> }	Chaves	De <i>n</i> até <i>m</i>

## Âncoras

Metacaractere	Nome	Função

^	Circunflexo	Início da linha
\$	Cifrão	Fim da linha
\b	Borda	Início ou fim de palavra

## Outros

Metacaractere	Nome	Função
\c	Escape	Torna literal o caractere c
	Ou	Ou um ou outro
(...)	Grupo	Delimita um grupo
\1...\9	Retrovisor	Texto casado nos grupos 1...9



Opa, não confunda! Os curingas usados na linha de comando para especificar nomes de arquivos, como \*.txt, relatorio.{txt,doc} e foto-???.jpg não são expressões regulares. São curingas específicos de nomes de arquivo, e, apesar de parecidos, são outra coisa e os significados de seus símbolos são diferentes dos das expressões. Então o melhor que você faz agora é esquecer esses curingas, senão eles podem confundi-lo e atrapalhar seu aprendizado.

Ah, antes que eu me esqueça: para testar os exemplos que veremos a seguir, acesse o site do livro: [www.piazinho.com.br](http://www.piazinho.com.br). Há uma ferramenta especial para você testar todos os exemplos, além de poder fazer suas próprias expressões. Experimente!

## Metacaracteres tipo Representante

O primeiro grupo de metacaracteres que veremos são os do tipo

representante, ou seja, são metacaracteres cuja função é representar um ou mais caracteres.

Também podem ser encarados como apelidos, links ou qualquer outra coisa que lhe lembre essa associação entre elementos. Todos os metacaracteres desse tipo casam a posição de um único caractere, e não mais que um.

## Ponto: o necessitado .

O ponto é nosso curinga solitário, que está sempre à procura de um casamento, não importa com quem seja. Pode ser um número, uma letra, um Tab, um @, o que vier ele traça, pois o ponto casa qualquer coisa.

Suponhamos uma ER que contenha os caracteres “fala” e o metacaractere ponto, assim: “fala.”. No texto a seguir, essa ER casaria tudo o que está sublinhado:

“Olha, com vocês me pressionando, a fala não vai sair natural. Eu não consigo me concentrar na minha fala. Aliás, isso é um falatório, pois nunca vi um comercial com tantas falas assim. Vou me queixar com o problemasnafala@medicos.com.br.”

Nesse pequeno trecho de texto, nossa ER casou cinco vezes, tendo o ponto casado com os seguintes caracteres: “.ts@”.



**ATENÇÃO:** O metacaractere ponto casa, entre outros, o caractere ponto.


Como exemplos de uso do ponto, em um texto normal em português, você pode procurar palavras que você não se lembra se acentuou ou não, que podem começar com maiúsculas ou não ou que foram escritas errado:

Expressão	Casa com
n.o	não, nao, ...
.ec lado	teclado, Teclado, ...
e.tendido	estendido, extendido, eztendido, ...

Ou, para tarefas mais específicas, procurar horário com qualquer separador

ou com marcações (“tags”) HTML:

Expressão	Casa com
12 . 45	12:45, 12 45, 12.45, ...
< . >	<B>, <i>, <p>, ...



- O ponto casa com qualquer coisa.
- O ponto casa com o ponto.
- O ponto é um curinga para casar um caractere.

## Lista: a exigente [ . . . ]

Bem mais exigente que o ponto, a lista não casa com qualquer um. Ela sabe exatamente o que quer, e não aceita nada diferente daquilo, a lista casa com quem ela conhece.

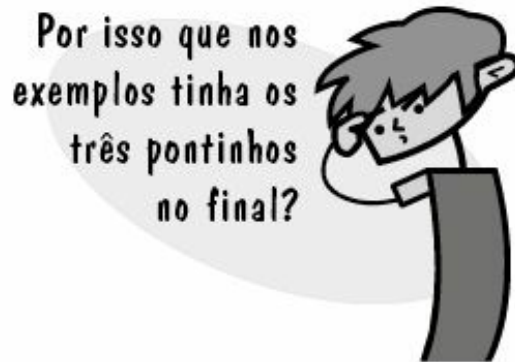
Ela guarda dentro de si os caracteres permitidos para casar, então algo como [aeiou] limita nosso casamento a letras vogais.

No exemplo anterior do ponto, sobre acentuação, tínhamos a ER n.o. Além dos casamentos desejados, ela é muito abrangente, e também casa coisas indesejáveis como neo, n-o, n5o e n o.

Para que nossa ER fique mais específica, trocamos o ponto pela lista, para casar apenas “não” e “nao”, veja:

n[ãa]o

E, assim como o n.o, todos os outros exemplos anteriores do ponto casam muito mais que o desejado, justo pela sua natureza promíscua.



Exatamente, eles indicam que havia mais possibilidades de casamento. Como o ponto casa com qualquer coisa, ele é nada específico. Então vamos impor limites às ERs:

Expressão	Casa com
<code>n[ãa]o</code>	não, nao
<code>[Tt]eclado</code>	Teclado, teclado
<code>e[sx]tendido</code>	estendido, extendido
<code>12[:. ]45</code>	12:45, 12.45, 12 45
<code>&lt;[BIP]&gt;</code>	<B>, <I>, <P>



Pegadinha! Não. Registre em algum canto de seu cérebro: dentro da lista, todo mundo é normal. Repetindo: dentro da lista, todo mundo é normal. Então aquele ponto é simplesmente um ponto normal, e não um metacaractere.

No exemplo de marcação `<[BIP]>`, vemos que as ERs são sensíveis a maiúsculas e minúsculas, então, se quisermos mais possibilidades, basta

incluí-las:

Expressão	Casa com
<[BIPbip]>	<B>, <I>, <P>, <b>, <i>, <p>

## Intervalos em listas

Por enquanto, vimos que a lista abriga todos os caracteres permitidos em uma posição. Como seria uma lista que dissesse que em uma determinada posição poderia haver apenas números?

**Peraí que essa eu sei... deixa ver...  
[0123456789]. Acertei?**



Sim! Então, para casar uma hora, qualquer que ela seja, fica como? Lembre que o formato é hh:mm.

**Tá. [0123456789][0123456789]:[0123 – Argh! QUE SACO!**



Pois é! Assim também pensaram nossos ilustres criadores das ERs, e, para evitar esse tipo de listagem extensa, dentro da lista temos o conceito de intervalo.

Lembra que eu disse para você memorizar que dentro da lista, todo mundo é normal? Pois é, aqui temos a primeira exceção à regra. Todo mundo, fora o traço. Se tivermos um traço (-) entre dois caracteres, isso representa todo o intervalo entre eles.



Não entendeu? É assim, olhe:

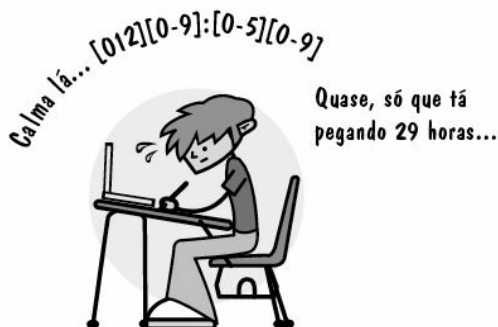
[0123456789] é igual a [0-9]

É simples assim. Aquele tracinho indica um intervalo, então 0-9 se lê: de zero a nove.

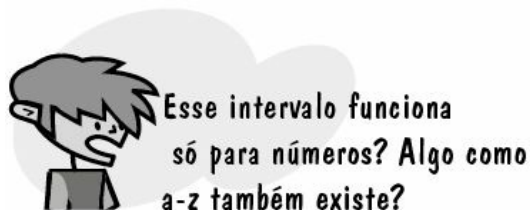
Voltando a nossa ER da hora, poderíamos fazer

[0-9][0-9]:[0-9][0-9]

mas veja que não é específico o bastante, pois permite uma hora como 99:99, que não existe. Como poderíamos fazer uma ER que casasse no máximo 23 horas e 59 minutos?



Excelente! Com o que aprendemos até agora, esse é o máximo de precisão que conseguimos. Mais adiante, quem poderá nos ajudar será o ou. Depois voltamos a esse problema.



Era isso que eu ia falar agora. Sim, qualquer intervalo é válido, como a-z, A-Z, 5-9, a-f, :-@ etc.



Sim. Por exemplo, se eu quiser uma lista que case apenas letras maiúsculas, minúsculas e números: [A-Za-z0-9].



Sim. Ah! E tem uma pegadinha. Como o traço é especial dentro da lista, como fazer quando você quiser colocar na lista um traço literal?

Sei lá, eu queria saber sobre o intervalo do arroba...



Espere um pouco. Basta colocar o traço no final da lista, assim [0-9-] casa números ou um traço. E tem os colchetes, que são os delimitadores da lista. Como incluí-los dentro dela?

O colchete que abre não tem problema, pode colocá-lo em qualquer lugar na lista, pois ela já está aberta mesmo e não se pode ter uma lista dentro da outra.

O colchete que fecha deve ser colocado no começo da lista, ser o primeiro item dela, para não confundir com o colchete que termina a lista. Então []-] casa um ] ou um -.

Vamos juntar tudo e fazer uma lista que case ambos os colchetes e o traço: [][]-. Calma. Pare, pense, respire fundo, encare esta ER. Vamos lê-la um por um: o primeiro [ significa que é o começo de uma lista, já dentro da lista, temos um ] literal, seguido de um [ literal, seguido de um - literal, e por último o ] que termina a lista. Intuitivo, não? ;)

Tá, confundi tudo,  
mas que diabos tem  
entre o : e o @???



Tudo bem, você venceu. Nesse intervalo tem : ; < = > ? @. Como saber isso? Os intervalos respeitam a ordem numérica da tabela ASCII, então basta tê-la em mãos para ver que um intervalo como A-z não pega somente as maiúsculas e minúsculas, como era de se esperar.

Para sua comodidade, a tabela está no fim do livro, e nela podemos ver que A-z pega também “[\]^\_`” e não pega os caracteres acentuados como “áéóôç”. Infelizmente, não há um intervalo válido para pegarmos todos os caracteres acentuados de uma vez. Mas já já veremos a solução...



**ATENÇÃO:** Não use o intervalo A-z,  
prefira A-Za-z.

## Dominando caracteres acentuados (POSIX)

Como para nós brasileiros se a-z não casar letras acentuadas não serve para muita coisa, temos uns curingas para usar dentro de listas que são uma mão na roda. Duas até.

Eles são chamados de classes de caracteres POSIX. São grupos definidos por tipo, e POSIX é um padrão internacional que define esse tipo de regra, como será sua sintaxe etc. Falando em sintaxe, aqui estão as classes:

Classe POSIX	Similar	Significa
[ :upper: ]	[A-Z]	Letras maiúsculas
[ :lower: ]	[a-z]	Letras minúsculas
[ :alpha: ]	[A-Za-z]	Maiúsculas/minúsculas
[ :alnum: ]	[A-Za-z0-9]	Letras e números

<code>[:digit:]</code>	<code>[0-9]</code>	Números
<code>[:xdigit:]</code>	<code>[0-9A-Fa-f]</code>	Números hexadecimais
<code>[:punct:]</code>	<code>[.,!?:...]</code>	Sinais de pontuação
<code>[:blank:]</code>	<code>[ \t]</code>	Espaço e Tab
<code>[:space:]</code>	<code>[ \t\n\r\f\v]</code>	Caracteres brancos
<code>[:cntrl:]</code>		Caracteres de controle
<code>[:graph:]</code>	<code>[^ \t\n\r\f\v]</code>	Caracteres imprimíveis
<code>[:print:]</code>	<code>[^ \t\n\r\f\v]</code>	Imprimíveis e o espaço

Note que os colchetes fazem parte da classe e não são os mesmos colchetes da lista. Para dizer maiúsculas, fica `[:upper:]`, ou seja, um `[:upper:]` dentro de uma lista `[]`.



**ATENÇÃO:** O `[:upper:]` é uma classe POSIX dentro de uma lista.

Então, em uma primeira olhada, `[:upper:]` é o mesmo que A-Z, letras maiúsculas. Mas a diferença é que essas classes POSIX levam em conta a localidade do sistema.

Atenção para essa diferença, pois a literatura na língua inglesa sempre fala sobre esse assunto muito superficialmente, pois eles não utilizam acentuação e deve ser às vezes até difícil para quem está escrevendo o documento entender isso.


Como nossa situação é inversa, e nossa língua é rica em caracteres acentuados, entender essa diferença é de suma importância.

Como estamos no Brasil, geralmente nossas máquinas estão configuradas para usar os números no formato `nnn.nnn,nn`, a data é no formato `dd/mm/aaaa`, medidas de distância são em centímetros e há outras coisinhas que são diferentes nos demais países.

Entre outros, também está definido que áéíóú são caracteres válidos em

nosso alfabeto, bem como ÁÉÍÓÚ.

Então, toda essa volta foi para dizer que o `[ :upper: ]` leva isso em conta e inclui as letras acentuadas também na lista. O mesmo para o `[ :lower: ]`, o `[ :alpha: ]` e o `[ :alnum: ]`.

 **ATENÇÃO:** Nos Estados Unidos,  
`[ :upper: ]` é igual a `[ A-Z ]`. No Brasil,  
`[ :upper: ]` é igual a  
`[ A-ZÁÃÂÄÊËÍÓÕÔÚÇ... ]`

Por isso, para nós, essas classes POSIX são importantíssimas, e sempre que você tiver de fazer ERs que procurarão em textos em português, prefira `[ :alpha: ]` em vez de `A-Za-z`, sempre.

Então, refazendo a ER que casava maiúsculas, minúsculas e números, temos:

```
[ :upper: ][ :lower: ][ :digit: ]
```

ou melhor:

```
[ :alpha: ][ :digit: ]
```

ou melhor ainda:

```
[ :alnum: ]
```

Todas são equivalentes.



Tudo bem, acabou (será?). Mas não se assuste, a lista é o único metacaractere que tem suas próprias regras, funcionando como uma minilinguagem dentro das expressões regulares.

---



- A lista casa com quem ela conhece e tem suas próprias regras.
- Dentro da lista, todo mundo é normal.
- Dentro da lista, traço indica intervalo.
- Um - literal deve ser o último item da lista.
- Um ] literal deve ser o primeiro item da lista.
- Os intervalos respeitam a tabela ASCII (não use A-Z).
- [:classes POSIX:] incluem acentuação, A-Z não.

## Lista negada: a experiente [^...]

Nem tão exigente quanto a lista nem tão necessitada quanto o ponto, temos a lista negada, que pelas suas más experiências passadas, sabe o que não serve para ela casar.




É rapidinho. A lista negada é exatamente igual à lista, podendo ter caracteres literais, intervalos e classes POSIX. Tudo o que se aplica a lista normal se aplica à negada também.

A única diferença é que ela possui lógica inversa, ou seja, ela casará com qualquer coisa, exceto com os caracteres listados.

Observe que a diferença em sua notação é que o primeiro caractere da lista é um circunflexo, ele indica que essa é uma lista negada. Então, se [0-9] são números, [^0-9] é qualquer coisa fora números. Pode ser letras, símbolos, espaço em branco, qualquer coisa, menos números.

Mas tem de ser alguma coisa. Só porque ela é uma lista negada isso não significa que ela pode casar “nada”.

Explicando em outras palavras, se você diz “qualquer coisa fora números”, deve haver outra coisa no lugar dos números e não simplesmente “se não houver números”. Então essa ER não casaria uma linha vazia, por exemplo.

 **ATENÇÃO:** “qualquer coisa fora alguns caracteres” não inclui “nenhum caractere”.

Como o traço e o colchete que fecha, o circunflexo é especial, então, para colocarmos um ^ literal em uma lista, precisamos pô-lo em qualquer posição que não seja a primeira. Assim [A-Z^] casa maiúsculas e o circunflexo e [^A-Z^] é o inverso: qualquer coisa fora maiúsculas e o circunflexo.

Ah! As classes POSIX também podem ser negadas, então

[^[:digit:]]

casa “qualquer coisa fora números”.

A lista negada é muito útil quando você sabe exatamente o que não pode ter em uma posição, como um erro ortográfico ou de escrita. Por exemplo, como mandam as regras da boa escrita, sempre após caracteres de pontuação, como a vírgula ou o ponto, devemos ter um espaço em branco os separando do resto do texto. Então vamos procurar por qualquer coisa que não seja o espaço após a pontuação:

[;,:.!?][^ ]

Ou, ainda, explicitando melhor nosso objetivo:

[[:punct:]][^ ]



- Uma lista negada segue todas as regras de uma lista normal.
- Um ^ literal não deve ser o primeiro item da lista.
- [:classes POSIX:] podem ser negadas.

- A lista negada sempre deve casar algo.

## Metacaracteres tipo Quantificador

Aqui chegamos ao segundo tipo de metacaracteres, os quantificadores, que servem para indicar o número de repetições permitidas para a entidade imediatamente anterior. Essa entidade pode ser um caractere ou metacaractere.

Em outras palavras, eles dizem a quantidade de repetições que o átomo anterior pode ter, ou seja, quantas vezes ele pode aparecer.

Os quantificadores não são quantificáveis, então dois deles seguidos em uma ER é um erro, salvo quantificadores não-gulosos, que veremos depois.

E memorize, por enquanto sem entender o porquê: todos os quantificadores são gulosos.

## Opcional: o opcional ?

O opcional é um quantificador que não esquentar a cabeça, para ele pode ter ou não a ocorrência da entidade anterior, pois ele a repete 0 ou 1 vez. Por exemplo, a ER 7? significa zero ou uma ocorrência do número 7. Se tiver um 7, beleza, casamento efetuado. Se não tiver, beleza também. Isso torna o 7 opcional (daí o nome), ou seja, tendo ou não, a ER casa. Veja mais um exemplo, o plural. A ER ondas? tem a letra s marcada como opcional, então ela casa onda e ondas.



**ATENÇÃO:** Cada letra normal é um átomo da ER, então o opcional é aplicado somente ao s e não à palavra toda.





Claro! Agora vamos começar a ver o poder de uma expressão regular. Já vimos o metacaractere lista e agora vimos o opcional, então, que tal fazermos uma lista opcional? Voltando um pouco àquele exemplo da palavra fala, vamos fazer a ER `fala[r!]`?. Mmmmmm... As ERs estão começando a ficar interessantes, não? Mas, antes de analisar essa ER, uma dica que vale ouro, memorize já: leia a ER átomo por átomo, da esquerda para a direita. Repetindo: leia a ER átomo por átomo, da esquerda para a direita.

## Como ler uma ER

É bem simples, uma ER se lê exatamente como o robozinho (lembra quem ele é?) leria. Primeiro lê-se átomo por átomo, depois entende-se o todo e então se analisa as possibilidades. Na nossa ER `fala[r!]` em questão, sua leitura fica: um f seguido de um a, seguido de um l, seguido de um a, seguido de: ou r, ou !, ambos opcionais.

Essa leitura é essencial para o entendimento da ER. Ela pode em um primeiro momento ser feita em voz alta, de maneira repetitiva, até esse processo se tornar natural. Depois ela pode ser feita mentalmente mesmo, e de maneira automática. É como você está fazendo agora, repetindo mentalmente estas palavras escritas aqui enquanto as lê. Você não percebe, faz normalmente.

Feita a leitura, agora temos de entender o todo, ou seja, temos um trecho literal `fala`, seguido de uma lista opcional de caracteres. Para descobrirmos as possibilidades, é o `fala` seguido de cada um dos itens da lista e por fim

seguido por nenhum deles, pois a lista é opcional. Então fica:


Expressão	Casa com
<code>fa la[r!]?</code>	falar, fala!, fala

Pronto! Desvendamos os segredos da ER. É claro, esta é pequena e fácil, mas o que são ER grandes senão várias partes pequenas agrupadas? O principal é dominar essa leitura por átomos. O resto é ler devagar até chegar ao final. Não há mistério.

Então voltemos ao nosso exemplo de marcações HTML, podemos facilmente incluir agora as marcações que fecham o trecho, em que a única diferença é que vem uma barra / antes da letra:

Expressão	Casa com
<code>&lt;/?[BIPbip]&gt;</code>	<code>&lt;/B&gt;</code> , <code>&lt;/I&gt;</code> , <code>&lt;/P&gt;</code> , <code>&lt;/b&gt;</code> , <code>&lt;/i&gt;</code> , <code>&lt;/p&gt;</code> , <code>&lt;B&gt;</code> , <code>&lt;I&gt;</code> , <code>&lt;P&gt;</code> , <code>&lt;b&gt;</code> , <code>&lt;i&gt;</code> , <code>&lt;p&gt;</code>

Por alguns segundos, contemple a ER anterior. Estamos começando a dizer muito com poucos caracteres, sendo específicos. Vamos continuar que vai ficar cada vez mais interessante.



- O opcional é opcional.
- O opcional é útil para procurar palavras no singular e plural.
- Podemos tornar opcionais caracteres e metacaracteres.
- Leia a ER átomo por átomo, da esquerda para a direita.
- Leia a ER, entenda o todo e analise as possibilidades.

## Asterisco: o tanto-faz \*

Se o opcional já não esquentar a cabeça, podendo ter ou não a entidade anterior, o asterisco é mais tranquilo ainda, pois para ele pode ter, não ter ou ter vários, infinitos. Em outras palavras, a entidade anterior pode aparecer em

qualquer quantidade.

Expressão	Casa com
$7^*0$	0, 70, 770, 7770, ..., 777777777770, ...
$bi^*p$	bp, bip, biip, biiip, biiiip...
$b[ip]^*$	b, bi, bip, biipp, bpipipi, biiiip, bppp, ...

Como HTML é sempre um ótimo exemplo, voltamos ao nosso caso das marcações, que podem ter vários espaços em branco após o identificador, então `<b >` e `</b >` são válidos. Vamos colocar essa condição na ER:

Expressão	Casa com
$</?[BIPbip]^*>$	<code>&lt;/B&gt;</code> , <code>&lt;/B &gt;</code> , <code>&lt;/B &gt;</code> , ..., <code>&lt;p &gt;</code> , ...

Note que agora, com o asterisco, nossa ER já não tem mais um número finito de possibilidades. Vejamos como fica a leitura dessa ER: um `<`, seguido ou não de uma `/`, seguido de: ou B, ou I, ou P, ou b, ou i, ou p, seguido ou não de vários espaços, seguido de `>`.

## Apresentando a gulodice

Pergunta: o que casará  $[ar]^*a$  na palavra arara? Alternativas:

- 1) a  $[ar]$  zero vezes, seguido de a
- 2) ara  $[ar]$  duas vezes (a,r), seguido de a
- 3) arara  $[ar]$  quatro vezes (a,r,a,r), seguido de a
- 4) n.d.a.

Acertou se você escolheu a número 3. O asterisco repete em qualquer quantidade, mas ele sempre tentará repetir o máximo que conseguir. As três alternativas são válidas, mas entre casar a lista  $[ar]$  zero, duas ou quatro vezes, ele escolherá o maior número possível. Por isso se diz que o asterisco é guloso.

Essa gulodice às vezes é boa, às vezes é ruim. Os próximos quantificadores, mais e chaves, bem como o opcional já visto, são igualmente gulosos. Mais detalhes sobre o assunto, confira mais adiante.

## Apresentando o curinga .\*

Vimos até agora que temos dois metacaracteres extremamente abrangentes, como o ponto (qualquer caractere) e o asterisco (em qualquer quantidade). E se juntarmos os dois? Teremos qualquer caractere, em qualquer quantidade. Pare um instante para pensar nisso. O que isso significa? Tudo? Nada? A resposta é: ambos.

O nada, pois “qualquer quantidade” também é igual a “nenhuma quantidade”. Então é opcional termos qualquer caractere, não importa. Assim, uma ER que seja simplesmente .\* sempre será válida e casará mesmo uma linha vazia.

O tudo, pois “qualquer quantidade” também é igual a “tudo o que tiver”. E é exatamente isso o que o asterisco faz, ele é guloso, ganancioso, e sempre tentará casar o máximo que conseguir. Repita comigo: o MÁXIMO que conseguir.



**ATENÇÃO:** O curinga .\* é qualquer coisa!

Assim, temos aqui o curinga das ERs, uma carta para se usar em qualquer situação. É muito comum, ao escrever uma expressão regular, você definir alguns padrões que procura, e lá no meio, em uma parte que não importa, pode ser qualquer coisa, você coloca um .\* e depois continua a expressão normalmente.

Por exemplo, para procurar ocorrência de duas palavras na mesma linha, relatório.\*amanhã serve para achar aquela linha maldita em que lhe pediram um trabalho “para ontem”. Ou, ainda, procurar acessos de usuários em uma data qualquer: 22/11/2001.\*login.



■ O asterisco repete em qualquer quantidade.

- Quantificadores são gulosos.
- O curinga . \* é o tudo e o nada, qualquer coisa.

## Mais: o tem-que-ter +

O mais tem funcionamento idêntico ao do asterisco, tudo o que vale para um se aplica ao outro.

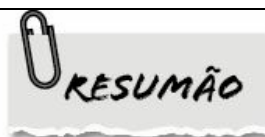
A única diferença é que o mais não é opcional, então a entidade anterior deve casar pelo menos uma vez, e pode haver várias.

Sua utilidade é quando queremos no mínimo uma repetição. Não há muito que acrescentar, é um asterisco mais exigente...



Pois é...

Expressão	Casa com
$7+\emptyset$	70, 770, 7770, ..., 777777770, ...
$bi+p$	bip, biip, biiip, biiiip...
$b[ip]^+$	bi, bip, biipp, bpipipi, biiiip, bppp, ...



- O mais repete em qualquer quantidade, pelo menos uma vez.
- O mais é igual ao asterisco, só mais exigente.

## Chaves: o controle $\{n, m\}$

Aqui Chaves não é o autor mexicano preferido de dez entre dez brasileiros. As chaves são a solução para uma quantificação mais controlada, onde se pode especificar exatamente quantas repetições se quer da entidade anterior.

Basicamente,  $\{n, m\}$  significa de  $n$  até  $m$  vezes, assim algo como  $7\{1, 4\}$  casa 7, 77, 777 e 7777. Só, nada mais que isso.

Temos também a sintaxe relaxada das chaves, em que podemos omitir a quantidade final ou ainda, especificar exatamente um número:

Metacaractere	Repetições
$\{1, 3\}$	De 1 a 3
$\{3, \}$	Pelo menos 3 (3 ou mais)
$\{0, 3\}$	Até 3
$\{3\}$	Exatamente 3
$\{1\}$	Exatamente 1
$\{0, 1\}$	Zero ou 1 (igual ao opcional)
$\{0, \}$	Zero ou mais (igual ao asterisco)
$\{1, \}$	Um ou mais (igual ao mais)

Note que o  $\{1\}$  tem efeito nulo, pois  $7\{1\}$  é igual a 7. Pode ser útil caso você queira impressionar alguém com sua ER, pode encher de  $\{1\}$  que não mudará sua lógica. Mas observe os três últimos exemplos.

Com as chaves, conseguimos simular o funcionamento de outros três metacaracteres, o opcional, o asterisco e o mais.

Se temos as chaves que já fazem o serviço, então pra que ter os outros três? Você pode escolher a resposta que achar melhor. Eu tenho algumas:

- $*$  é menor e mais fácil que  $\{0, \}$ .
- As chaves foram criadas só depois dos outros.
- Precisavam de mais metacaracteres para complicar o assunto.

- \*, + e ? são links para as chaves.
- Alguns teclados antigos vinham sem a tecla {.
- O asterisco é tão bonitinho...

Como você pode perceber, não há uma resposta certa. Então todas as especulações citadas podem ser corretas. Invente uma e me mande, vamos fazer uma coleção!

Ah, e sendo {0,} algo mais feio que um simples \*, isso também pode ser usado para tornar sua ER grande e intimidadora. Só cuidado para não atirar no próprio pé e depois não conseguir entender sua própria criação...



- Chaves são precisas.
- Você pode especificar um número exato, um mínimo, um máximo, ou uma faixa numérica.
- As chaves simulam os seguintes metacaracteres: \* + ?.
- As chaves não são o Chaves.

## Metacaracteres tipo Âncora

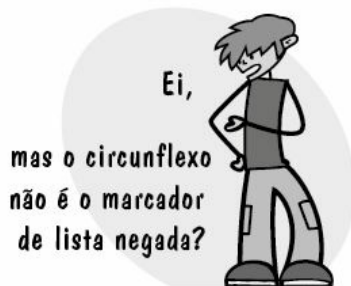
Bem, deixando os quantificadores de lado, vamos agora falar sobre os metacaracteres do tipo âncora.

Por que âncora? Porque eles não casam caracteres ou definem quantidades, em vez disso, eles marcam uma posição específica na linha.

Assim, eles não podem ser quantificados, então o mais, o asterisco e as chaves não têm influência sobre âncoras.

## Circunflexo: o início ^

O nosso amigo circunflexo (êta nome comprido e chato) marca o começo de uma linha. Nada mais.



Também, mas apenas dentro da lista (e no começo), fora dela, ele é a âncora que marca o começo de uma linha, veja:

```
^[0-9]
```

Isso quer dizer: a partir do começo da linha, case um número, ou seja, procuramos linhas que começam com números. O contrário seria:

```
^[^0-9]
```

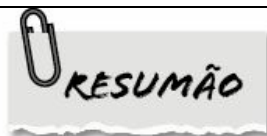
Ou seja, procuramos linhas que não começam com números. O primeiro circunflexo é a âncora e o segundo é o “negador” da lista. E como não poderia deixar de ser, é claro que o circunflexo como marcador de começo de linha só é especial se estiver no começo da ER. Não faz sentido procurarmos uma palavra seguida de um começo de linha, pois se tiver uma palavra antes do começo de uma linha, ali não é o começo da linha! Desse modo, a ER:

```
[0-9]^
```

Casa um número seguido de um circunflexo literal, em qualquer posição da linha. Com isso em mente, você pode me dizer o que casa a ER:

```
^^
```

Pois é, uma ER tão singela e harmônica como essa procura por linhas que começam com um circunflexo. Legal né? E para fechar, uma ER que, em um e-mail, casa as conversas anteriores, aquelas linhas que começam com os sinais de maior >, abominados por muitos. Ei! Essa você mesmo pode fazer, não?



▪ Circunflexo é um nome chato, porém chapeuzinho é legal.



- Serve para procurar palavras no começo da linha.
- Só é especial no começo da ER (e de uma lista).

## Cifrão: o fim \$

Similar e complementar ao circunflexo, o cifrão marca o fim de uma linha e só é válido no final de uma ER. Como o exemplo anterior, `[0-9]$` casa linhas que terminam com um número. E o que você me diz da ER a seguir?

`^$`



Sim, e o que isso significa?



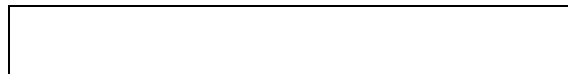
Isso! É sempre bom ter essa ER na manga, pois procurar por linhas em branco é uma tarefa comum nas mais diversas situações. Podemos também casar apenas os cinco últimos caracteres de uma linha.

`.....$`

Ou, ainda, que tal casarmos linhas que tenham entre 20 e 60 caracteres?

`^.{20,60}$`

É comum pessoas (inclusive eu) chamarem o cifrão de dólar. Vamos abolir essa prática. Chame até de “ésse riscado”, mas dólar é feio. É como diria meu amigo Julio Neves, lugar de dólar é no bolso.





- Serve para procurar palavras no fim da linha.
- Só é especial no final da ER.
- É cifrão, e não dólar.

## Borda: a limítrofe \b

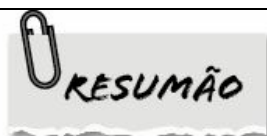
A outra âncora que temos é a borda, que como o próprio nome já diz, marca uma borda, mais especificamente, uma borda de palavra.

Ela marca os limites de uma palavra, ou seja, onde ela começa e/ou termina. Muito útil para casar palavras exatas, e não partes de palavras. Veja como se comportam as ERs nas palavras dia, diafragma, melodia, radial e bom-dia!:

Expressão	Casa com
dia	dia, diafragma, melodia, radial, bom-dia!
\bdia	dia, diafragma, bom-dia!
dia\b	dia, melodia, bom-dia!
\bdia\b	dia, bom-dia!

Assim vimos que a borda força um começo ou a terminação de uma palavra. Entenda que “palavra” aqui é um conceito que engloba [A-Za-z0-9\_] apenas, ou seja, letras, números e o sublinhado. Por isso \bdia\b também casa bom-dia!, pois o traço e a exclamação não são parte de uma palavra.

Ah! Dependendo do aplicativo, o sublinhado não faz parte de uma palavra.



- A borda marca os limites de uma palavra.
- O conceito “palavra” engloba letras, números e o sublinhado.
- A borda é útil para casar palavras exatas e não parciais.

## Outros metacaracteres

Deixando as âncoras mergulhadas em nossa memória, agora já sabemos como casar algo, em alguma quantidade, em algum lugar na linha.

Então vamos ver outros metacaracteres que têm funções específicas e não relacionadas entre si e, portanto, não podem ser agrupados em outra classe fora a tradicional “outros”.

Mas atenção, isso não quer dizer que eles são inferiores, pelo contrário, o poder das ERs é multiplicado com seu uso e um mundo de possibilidades novas se abre a sua frente.

E antes de ver o primeiro deles, o criptonita, uma historinha para descontrair:

.....

Enquanto isso, na sala de justiça...

– Garoto-prodígio, você sabe algo sobre essa pedra verde e brilhante?

– Não sei homem-morcego.

– Foi o homem-do-planeta-bizarro que deixou aqui...

– Um presente! Vamos mostrá-la ao Super-Homem!

E escancarando a porta do banheiro, Robin diz:

– Ei Super-Homem, olhe que legal a pedra que o homem-do-pla...

– Aaaaaaaaaaaaaaaaaaaaaaargh...

– Santa enfermidade Batman, ele derreteu!!!

.....

## Escape: a criptonita \

E tudo estava indo bem na sua vida nova de criador de ERs, quando de repente...



Se você está atento, lembrará que a lista tem suas próprias regras e que...



Isso! Cara esperto. Precisou de um caractere que é um meta, mas você quer seu significado literal, coloque-o dentro de uma lista, então `lua[*] casa lua*`. O mesmo serve para qualquer outro metacaractere. Maaaaaas, para não precisar ficar toda hora criando listas de um único componente só para tirar seu valor especial, temos o metacaractere criptonita `\`, que “escapa” um metacaractere, tirando todos os seus poderes.

Escapando, `\*` é igual a `[*]` que é igual a um asterisco literal. Similarmente podemos escapar todos os metacaracteres já vistos:

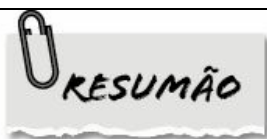
```
\. \[ \] \? \+ \{ \} \^ \$
```

E para você ver como são as coisas, o escape é tão poderoso que pode escapar a si próprio! O `\\` casa uma barra invertida `\` literal.

Ah! É claro, escapar um circunflexo ou cifrão somente é necessário caso eles estejam em suas posições especiais, como casar o texto `^destaque^`, em que ambos os circunflexos são literais, mas o primeiro será considerado uma âncora de começo de linha caso não esteja escapado.

Então, agora que sabemos muito sobre ERs, que tal uma expressão para casar um número de RG no formato `n.nnn.nnn-n`?

```
[0-9]\.[0-9]{3}\.[0-9]{3}-[0-9]
```



- O escape escapa um metacaractere, tirando seu poder.
- `\*` = `[*]` = asterisco literal.
- O escape escapa o escape, escapando-se a si próprio simultaneamente.

## Ou: o alternativo |

É muito comum em uma posição específica de nossa ER termos mais de uma alternativa possível, por exemplo, ao casar um cumprimento amistoso, podemos ter uma terminação diferente para cada parte do dia:

`boa-tarde|boa-noite`

O ou, representado pela barra vertical |, serve para esses casos em que precisamos dessas alternativas. Essa ER se lê: “ou boa-tarde, ou boa-noite”, ou seja, “ou isso ou aquilo”. Lembre-se de que a lista também é uma espécie de ou, mas apenas para uma letra, então:

`[gpr]ato e gato|pato|rato`

São similares, embora nesse caso, em que apenas uma letra muda entre as alternativas, a lista é a melhor escolha. Em outro exemplo, o ou é útil também para casarmos um endereço de Internet, que pode ser uma página ou um servidor FTP.

`http://|ftp://`

Ou isso ou aquilo, ou aquele outro... E assim vai. Podem-se ter tantas opções quantas for preciso. Não deixe de conhecer o parente de 1º grau do ou, o grupo, que multiplica seu poder. A seguir, neste mesmo canal.



- O ou indica alternativas.
- Lista para um caractere, ou para vários.
- O grupo multiplica o poder do ou.

## Grupo: o pop (...)

Assim como artistas famosos e personalidades que conseguem arrastar multidões, o grupo tem o dom de juntar vários tipos de sujeitos em um mesmo local. Dentro de um grupo, podemos ter um ou mais caracteres, metacaracteres e inclusive outros grupos! Como em uma expressão

matemática, os parênteses definem um grupo, e seu conteúdo pode ser visto como um bloco na expressão.

Todos os metacaracteres quantificadores que vimos anteriormente podem ter seu poder ampliado pelo grupo, pois ele lhes dá mais abrangência. E o ou, pelo contrário, tem sua abrangência limitada pelo grupo: pode parecer estranho, mas é essa limitação que lhe dá mais poder.

Em um exemplo simples,  $(ai)^+$  agrupa a palavra “ai” e esse grupo está quantificado pelo mais. Isso quer dizer que casamos várias repetições da palavra, como ai, aiai, aiaiai, ... E assim podemos agrupar tudo o que quisermos, literais e metacaracteres, e quantificá-los:

Expressão	Casa com
$(ha! )^+$	ha!, ha!ha!, ha!ha!ha!, ...
$(\backslash \cdot [0-9] ) \{3\}$	.0.6.2, .2.8.9, .7.7.7, ...
$(www\backslash \cdot )?zz\backslash \cdot .com$	www.zz.com, zz.com

E em especial, nosso amigo “ou” ganha limites e seu poder cresce:

Expressão	Casa com
$boa-(tarde noite)$	boa-tarde, boa-noite
$(\# n\backslash \cdot  núm) \ 7$	# 7, n. 7, núm 7
$(in con)?certo$	incerto, concerto, certo

Note que o grupo não altera o sentido da ER, apenas serve como marcador. Podemos criar subgrupos também, então imagine que você esteja procurando o nome de um supermercado em uma listagem e não sabe se este é um mercado, supermercado ou um hipermercado.

$(super|hiper)mercado$

Consegue casar as duas últimas possibilidades, mas note que nas alternativas super e hiper temos um trecho per comum aos dois, então podíamos “alternativizar” apenas as diferenças su e hi:

`(su|hi)permercado`

Precisamos também casar apenas o mercado sem os aumentativos, então temos de agrupá-los e torná-los opcionais:

`((su|hi)per)?mercado`

Pronto! Temos a ER que buscávamos e ainda esbanjamos habilidade utilizando um grupo dentro do outro.

Ei! E se tivesse  
minimercado  
também?



`(mini|(su|hi)per)?mercado`

E assim vai... Acho que já deu para notar quão poderosas e complexas podem ficar nossas ERs ao utilizarmos grupos, não? Mas não acaba por aqui! Acompanhe o retrovisor na sequência.

Espera!  
E se eu quiser casar  
um par de parênteses literais?



Ah! Lembra-se do escape criptonita? Basta tirar o poder dos parênteses, escapando-os. Geralmente é preciso casar parênteses literais ao procurar por nomes de funções no código de um programa, como por exemplo

`Minha_Funcao()`. A ER que casa esta e outras funções é:

`[A-Za-z0-9_]+\(\)`

Ou ainda, caso acentuação seja permitida em nomes de função (lembre-se das classes POSIX!):

`[[:alnum:]]+\(\)`



**RESUMÃO**

- Grupos servem para agrupar.

- Grupos são muito poderosos.
- Grupos podem conter grupos.
- Grupos são quantificáveis.

## Retrovisor: o saudosista \1 ... \9

Já vimos o poder do grupo e várias utilidades em seu uso. Mas ainda não acabou! Se prepare para conhecer o mundo novo que o retrovisor nos abre. Ou seria mundo velho?

Ao usar um (grupo) qualquer, você ganha um brinde, e muitas vezes nem sabe. O brinde é o trecho de texto casado pela ER que está no grupo, que fica guardado em um cantinho especial, e pode ser usado em outras partes da mesma ER!



Então vamos tentar de novo. Como o nome diz, é retrovisor porque ele “olha pra trás”, para buscar um trecho já casado. Isso é muito útil para casar trechos repetidos em uma mesma linha. Veja bem, é o trecho de texto, e não a ER.

Como exemplo, em um texto sobre passarinhos, procuramos o quero-quero. Podemos procurar literalmente por quero-quero, mas assim não tem graça, pois somos mestres em ERs e vamos usar o grupo e o retrovisor para fazer isso:

(quero)-\1

Então o retrovisor \1 é uma referência ao texto casado do primeiro grupo, nesse caso, quero, ficando, no fim das contas, a expressão que queríamos. O retrovisor pode ser lembrado também como um link ou um ladrão, pois copia



o texto do grupo.



Pois é, lembra que o escape \ servia para tirar os poderes do metacaractere seguinte. Então, a essa definição agora incluímos: a não ser que este próximo caractere seja um número de 1 a 9, então estamos lidando com um retrovisor.

Notou o detalhe? Podemos ter no máximo nove retrovisores por ER, então \10 é o retrovisor número 1 seguido de um zero. Alguns aplicativos novos permitem mais de nove.

**Não era muito mais fácil  
escrever quero-quero direto?!**



Nesse caso, sim. Mas esse é só um exemplo didático. O verdadeiro poder do retrovisor é quando não sabemos exatamente qual texto o grupo casará. Vamos estender nosso quero para “qualquer palavra”:

`([A-Za-z]+)-\1`

Percebeu o poder dessa ER? Ela casa palavras repetidas, separadas por um traço, como o próprio quero-quero, e mais: bate-bate, come-come etc. Mas e se tornássemos o traço opcional?

`([A-Za-z]+)-?\1`

Agora, além das anteriores, nossa ER também casa bombom, lili, dudu, bibi e outros apelidos e nomes de cachorro.

Com uma modificação pequena, fazemos um minicorretor ortográfico para

procurar por palavras repetidas como como estas em um texto:

```
([A-Za-z]+) \1
```

Mas como procuramos por palavras inteiras e não apenas trechos delas, então precisamos usar as bordas para completar nossa ER:

```
\b([A-Za-z]+) \1\b
```

Legal, né? Note como vamos construindo as ERs aos poucos, melhorando, testando e não simplesmente escrevendo tudo de uma vez. Esta é a arte ninja de se escrever ERs.

## Mais detalhes

Como já dito, podemos usar no máximo nove retrovisores. Vamos ver uns exemplos com mais de um de nossos amigos novos:

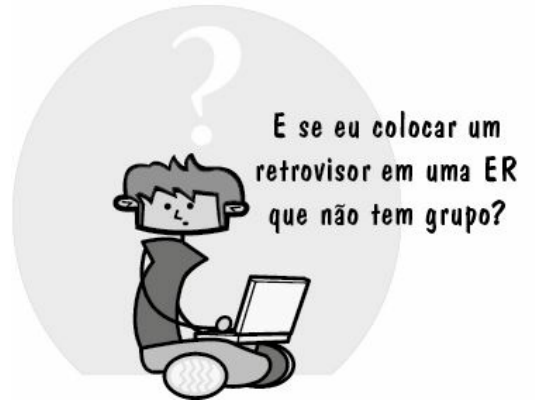
Expressão	Casa com
<code>(lenta)(mente) é \2 \1</code>	lentamente é mente lenta
<code>((band)eira)nte \1 \2a</code>	bandeirante bandeira banda
<code>in(d)ol(or) é sem \1\2</code>	indolor é sem dor
<code>((((a)b)c)d)-1 = \1,\2,\3,\4</code>	abcd-1 = abcd,abc,ab,a

Para não se perder nas contagens, há uma dica valiosa: conte somente os parênteses que abrem, da esquerda para a direita. Este vai ser o número do retrovisor. E o conteúdo é o texto casado pela ER do parêntese que abre até seu correspondente que fecha.



**ATENÇÃO:** O retrovisor referencia o texto casado e não a ER do grupo.

Nos nossos exemplos ocorre a mesma coisa porque a ER dentro do grupo já é o próprio texto, sem metacaracteres. Veja, entretanto, que `([0-9])\1` casa 66 mas não 69.



Vai dar pau :)

Apenas como lembrete, algumas linguagens e programas, além da função de busca, têm a função de substituição. O retrovisor é muito útil nesse caso, para substituir “alguma coisa” por “apenas uma parte dessa coisa”, ou seja, extrair trechos de uma linha. Mais detalhes sobre isso adiante.



- O retrovisor só funciona se usado com o grupo.
- O retrovisor serve para procurar palavras repetidas.
- Numeram-se retrovisores contando os grupos da esquerda para a direita.
- Temos no máximo 9 retrovisores por ER.



## Capítulo 3

# Mais sobre metacaracteres

Ufa! Terminamos nosso “metacaractere tour”. Você gostou do passeio?

Sempre que tiver dúvida ou esquecimento, reveja essa parte e relembre as funções de cada um, as quais você só vai memorizar com o uso contínuo das ERs. Pratique, pratique, pratique.

É importante que o que vimos até aqui esteja bem entendido para seguirmos adiante.

Não deixe de conferir também a tabela com o resumo de todos os metacaracteres no final do livro.

Como a insaciabilidade humana supera qualquer empenho, o assunto metacaracteres ainda não acabou e temos muito mais para conhecer, vamos lá?

## Épocas e aplicativos diversos, metacaracteres distorcidos

Agora que já aprendemos todos aqueles metacaracteres, sua sintaxe, suas regras, seus detalhes, é hora de aplicar na prática esse conhecimento. E aí é que vem a surpresa, quando você percebe que alguns metacaracteres não funcionam direito... O que acontece?

O que acontece é que existe uma diferença brutal de sintaxe entre aplicativos, em que cada um coloca sua personalidade, e temos várias maneiras de representar o mesmo metacaractere.

Por exemplo: o opcional é `?` no Python, Perl e linguagens de programação mais novas. Mas em aplicativos mais antigos como o `grep` e o `sed`, ele é escapado `\?`, sendo `?` uma mera interrogação literal. Já no editor de textos Vim, o opcional é representado pelo esquisito `\=`. Vai entender...

O motivo mais comum para isso acontecer são as razões históricas. Voltando no tempo, na época em que os primeiros programas começavam a ter suporte a expressões regulares, basicamente o que era comum de se fazer em um editor de textos eram códigos de programas, e não textos normais da linguagem falada.

Assim, como era (e ainda é) muito comum de se ter os caracteres `?`, `+`, `{`, `(` e `|` nos códigos de programas, era comum eles fazerem parte da ER por seu valor literal, ou seja, procurar por uma interrogação, por uma barra vertical etc. Então, no começo de tudo, esses metacaracteres eram todos escapados para serem especiais:

```
\? \+ \{ \( \|
```

Já aplicativos e linguagens mais novos, criados em um mundo onde a editoração eletrônica avançou muito e um “texto de computador” não era apenas um código de programa, todos os escapes foram retirados e os metacaracteres ficaram mais simples.

Aqueles aplicativos antigos, porém, continuam até hoje utilizando os escapes, pois têm de manter a compatibilidade com versões anteriores. Alguns são mais espertos e suportam ambas as sintaxes, escolhendo via configuração ou opção de linha de comando.

Toda essa historinha, além de curiosidade, está aqui para que você saiba o porquê das coisas, e isso o ajuda na hora da dúvida, pois, se você sabe que o aplicativo é antigo, provavelmente os metacaracteres devem ser escapados. Ou ainda, se você viu que o opcional `?` precisou ser escapado, outros provavelmente precisarão também.

Concluindo, ora razões históricas, ora vaidade do autor, o fato é que a diversidade impera e você vai ter de se acostumar com isso e aprender as diferenças de cada aplicativo que for utilizar. Bem-vindo ao mundo caótico da implementação das expressões regulares!

Mas para ajudar nesse reconhecimento da diferença, no final do livro há uma daquelas tabelinhas mágicas que parecem simples, mas demoram dias para se fazer, que pode e deve ser consultada em caso de dúvida.

Estão registradas todas as diferenças encontradas em vários aplicativos e linguagens. Com ela em mãos, você não precisa mais se preocupar com isso. O nome é “Diferenças de Metacaracteres entre Aplicativos”.

## Quantificadores gulosos

Como já vimos, todos os quantificadores são gulosos, pois sempre casam o máximo possível. Mas por que isso? Como isso acontece? Acompanhe um passo a passo para esclarecer o assunto e nunca mais sofrer por não entender a gulodice.

Para a demonstração, vamos pegar uma frase:

um **negrito** aqui.

Nosso objetivo é casar os marcadores `<b>` e `</b>` para apagá-los. Mas, ao aplicarmos a ER `<.*>`, vemos que ela casou além, pegando de uma vez todo o trecho `<b>negrito</b>`. O que aconteceu?

Aconteceu que o asterisco, como todo quantificador, é guloso e casou o máximo que conseguiu. Vamos entrar na abstração da abstração e entender como isso aconteceu.

Imaginemos que somos o próprio robzinho, então, como aplicaremos a ER no texto?

um **negrito** aqui.

^

`<.*>`

Logo abaixo da frase, o circunflexo indica onde está o foco da ER, e mais à direita está nossa ER. Estamos no estado inicial, parados no começo da linha,

e agora vamos tentar casar a expressão. A primeira coisa que temos para casar é o <.

Como a primeira letra da frase é um u, mudamos o foco para o próximo caractere, pois este não é o < que procuramos:

```
um <b>negrito</b> aqui.  
.^                <.*>
```

Mmmmmmm... ainda não deu, então continuamos assim, um por um, até conseguirmos um casamento:

```
um <b>negrito</b> aqui.  
..^                <.*>  
...^              <.*>
```

Opa, agora achamos um <! Conseguimos casar o primeiro átomo de nossa ER. Então marcamos como casado esse caractere e seguimos adiante para o próximo:

```
um <b>negrito</b> aqui.  
...x^              "<".*>
```

Os trechos já casados são representados pelos x marcando a frase e as aspas marcam a ER. Os pontinhos representam apenas os “rastros” do foco, as partes já visitadas e não casadas.

A próxima parte da ER a ser casada é o curinga, que casa qualquer caractere em qualquer quantidade. Então, procurando qualquer caractere, nosso curinga segue casando:

```
um <b>negrito</b> aqui.  
...x^              "<.*">  
...xx^             "<.*">  
...xxx^            "<.*">  
...xxxx^           "<.*">
```

Ei! Mas ele passou batido pelo > que a gente queria! Por quê? Lembre-se de que o ponto casa qualquer caractere. E por acaso o > também não é qualquer caractere? É, então o ponto casa ele também, seguindo guloso até o fim da linha:

```
um <b>negrito</b> aqui.
```

```
...xxxxxxxxxxxxxxxxxxxxx^ "<.*">
```

Pronto. Como bateu lá no final e não tem mais caracteres para casar, o asterisco sossega. Mas ainda temos um componente da ER para casar, o >. E agora?

Bem, o asterisco é guloso, mas não é egoísta, então, se ele precisar ceder alguma coisa, ele cede. E assim acontece, ele vai devolvendo até satisfazer o próximo componente da ER:

```
um <b>negrito</b> aqui.  
...xxxxxxxxxxxxxxxxxxxxx^ "<.*">  
...xxxxxxxxxxxxxxxxxxxxx^ "<.*">  
...xxxxxxxxxxxxxxxxxxxxx^ "<.*">  
...xxxxxxxxxxxxxxxxxxxxx^ "<.*">  
...xxxxxxxxxxxxxxxxxxxxx^ "<.*">  
...xxxxxxxxxxxxxxxxxxxxx^ "<.*">  
...xxxxxxxxxxxxxxxxxxxxx^ "<.*">
```

Opa, agora o asterisco devolveu um > que servirá para casar o último átomo de nossa ER:

```
um <b>negrito</b> aqui.  
...xxxxxxxxxxxxxxxxx^ "<.*>"
```

Pronto! Nossa ER agora está casada por inteiro, então não temos mais o que fazer, fim do processo. Agora ficou fácil entender essa gulodice? É sempre assim, casa o máximo possível, e se precisar, devolve alguns caracteres para satisfazer o resto da expressão.

Por causa dessa gulodice e da subsequente procura de trás para frente é que acaba se casando além do desejado. Exatamente assim também funcionam todos os outros quantificadores: mais, chaves e opcional. Sempre casam o máximo possível. Então, em uma visão diferente, vamos ver o que cada parte da ER casou na frase:

```
um <b>negrito</b> aqui.  
...x <  
....xxxxxxxxxxxxx . *  
.....x >
```



Quando o que normalmente se esperava conseguir era:

um **<b>negrito</b>** aqui.

```
...X.....X      <
...X.....XX     .*
.....X.....X    >
```

Para o asterisco ter esse comportamento, ou você faz uma ER mais específica (veja o tópico “Evite o curinga”), ou usa um quantificador não-guloso, se o aplicativo suportá-lo. Vamos conhecê-los!

## Quantificadores não-gulosos

A gulodice dos quantificadores é algo geralmente benéfico, mas, em certas situações, como a do negrito anterior, você quer o oposto: o menor casamento possível.

Apenas presente em linguagens e aplicativos mais recentes, essa opção de metacaracteres tem uma sintaxe fácil de lembrar, basta acrescentar uma interrogação logo após os quantificadores normais:

Metacaractere	Nome
??	Opcional não-guloso
*?	Asterisco não-guloso
+?	Mais não-guloso
{ <i>n</i> , <i>m</i> }?	Chaves não-gulosas

Não há muito que demonstrar, visto que já esmiuçamos a gulodice. A não-gulodice é o efeito contrário, um quantificador tímido, que só casa se o próximo átomo da ER não estiver precisando daquele caractere. O exemplo anterior do negrito fica `<.*?>`.

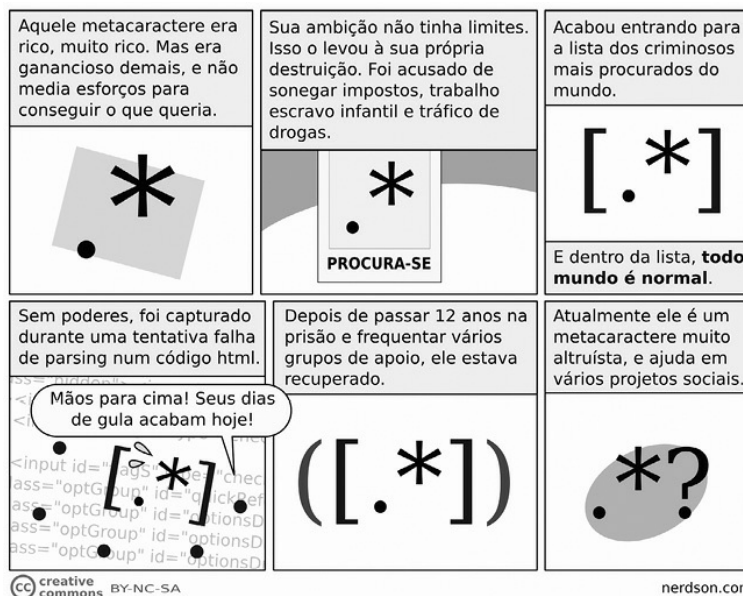
Veja a comparação entre ambos os tipos de gulodice em todos os quantificadores aplicados ao texto `abbbb`:

Gulosos		Não-gulosos	
<code>ab*</code>	<code>abbbb</code>	<code>ab*?</code>	<code>a</code>

ab+	abbbb	ab+?	ab
ab?	ab	ab??	a
ab{1, 3}	abbb	ab{1, 3}?	ab

Para ilustrar bem como isso acontece na prática, gravei uma vídeo-aula de sete minutos, explicando e demonstrando o funcionamento dos metacaracteres gulosos e não-gulosos. Aproveite e já veja esta aula agora, para fixar o aprendizado. A aula está no site do livro [www.piazinho.com.br](http://www.piazinho.com.br).

Toda essa história de gulodice dos quantificadores inspirou uma divertida tirinha do Nerdson (<http://hacktoon.com>), intitulada *Em terra de metacaractere, quem tem um asterisco é rei*. Confira:



## Metacaracteres tipo barra-letra

Os metacaracteres do tipo barra-letra são átomos representados por uma barra invertida \ seguida de uma letra qualquer, como \s e \w. Dependendo da letra, muda-se o significado desse metacaractere.

Contando que o alfabeto nos dá 26 letras e que maiúsculas são diferentes de minúsculas, duplicamos para 52 metacaracteres novos. Calma, nem todas as letras estão ocupadas... Ainda :)

Tudo começou nas linguagens de programação com os printf da vida, que começaram a interpretar coisas, como `\n` e `\t`, para significar quebra de linha e Tab, pois visualmente é ambíguo se o que tem entre “1 2” é um Tab ou vários espaços.

Além destes, havia outros barra-letras para representarem aqueles caracteres de controle chatos que de vez em quando aparecem em nossa vida e são difíceis de ver e representar.

Essa representação simplificada e útil se tornou padrão, e hoje a grande maioria dos aplicativos os entende. Eis a lista:

b-l	Nome	Tradução
<code>\a</code>	Alert	Alerta (bipe)
<code>\b</code>	Backspace	Caractere Backspace
<code>\e</code>	Escape	Caractere Esc
<code>\f</code>	Form feed	Alimentação
<code>\n</code>	Newline	Linha nova
<code>\r</code>	Carriage return	Retorno de carro
<code>\t</code>	Htab	Tabulação horizontal
<code>\v</code>	Vtab	Tabulação vertical

Como esses barra-letras também poderiam ser muito úteis para escrever nossas expressões regulares, os robzinhos começaram a ter suporte a eles também, ou senão a linguagem recebe a ER como um dado do tipo string e os interpreta, passando ao robzinho seus valores literais. Mais detalhes sobre isso no tópico “ERs Pré-Processadas e Cruas”, adiante.


Ah! Note que o `\b` se confunde com a borda. Cada aplicativo trata essa exceção à sua maneira: ou só é válido dentro da lista, ou depende do contexto, ou...

Aí é aquela história: onde passa boi, passa boiada. Observaram bem as classes POSIX, com aquela representação extensa e feia, depois compararam

com os barra-letras, que são curtos e charmosos, e foi um estalo: criaram barra-letras novos para representar as classes mais utilizadas.

São como apelidos, pois têm a mesma funcionalidade de seu equivalente POSIX, levando em conta a localização do sistema (acentuação), o que muito nos interessa.

b-l	Equivalente POSIX	Significa
\d	[[ :digit: ]]	Dígito
\D	[^ :digit: ]	Não-dígito
\w	[[ :alnum: ]_ ]	Palavra
\W	[^ :alnum: ]_ ]	Não-palavra
\s	[[ :space: ]]	Branco
\S	[^ :space: ]]	Não-branco

 **ATENÇÃO:** Geralmente um barra-LETRA é a negação de um barra-letra.

A grande diferença desses barra-letras para as classes POSIX é que eles são válidos fora das listas.

Dependendo do aplicativo, eles são válidos dentro das listas também, bagunçando aquela regra que vimos de que dentro da lista todo mundo é normal. Dependendo também, o sublinhado faz parte ou não do \w.

Com os poderes barra-letrísticos recém-adquiridos, podemos diminuir nossa ER para casar o RG:

```
[0-9]\.[0-9]{3}\.[0-9]{3}-[0-9]
\d\.\d{3}\.\d{3}-\d
```

Mas aí, como já estava feita a festa e ainda sobravam muitas letras do alfabeto, começaram a aparecer barra-letras aleatórios. Segue uma coletânea misturada deles, encontrados apenas em alguns poucos aplicativos, então confirme antes de usá-los:

--	--	--

b-l	Significado	Similar
\a	Alfabeto	[[:alpha:]]
\A	Não-alfabeto	[^[:alpha:]]
\h	Cabeça de palavra	[[:alpha:]]_
\H	Não-cabeça de palavra	[^[:alpha:]]_
\l	Minúsculas	[[:lower:]]
\L	Não-minúsculas	[^[:lower:]]
\u	Maiúsculas	[[:upper:]]
\U	Não-maiúsculas	[^[:upper:]]
\o	Número octal	[0-7]
\O	Não-número octal	[^0-7]
\B	Não-borda	
\A	Início do texto	
\Z	Fim do texto	
\l	Torna minúscula	
\L	Torna minúscula até \E	
\u	Torna maiúscula	
\U	Torna maiúscula até \E	
\Q	Escapa até \E	
\E	Fim da modificação	
\G	Fim do casamento anterior	

**Conclusão:** agora lhe resta procurar, na documentação de seu aplicativo, se os barra-letras são suportados, se sim, se pelo robzinho ou pela interpretação de strings, quais barra-letras, e se são válidos dentro e/ou fora das listas.

Nossa! Essa “padronização” das implementações de expressões regulares realmente impressiona...

# Metacaracteres modernos

Bem, tem gente que não sossega mesmo.

Talvez até pelo fato de ser gostoso brincar com ERs, com o passar do tempo, desenvolveu-se uma tendência de “ERs são a solução da fome mundial, elas têm de fazer tudo”, e o conceito foi maculado. Hoje, além de simplesmente casar um trecho de texto, criaram conceitos como:

- Case esta ER somente se seguida de tal coisa.
- Case esta ER somente se não precedida de tal coisa.
- O “tal coisa” pode ser uma ER mais complicada que a original.
- Grupos nomeáveis que geram pseudovariáveis.
- Grupos que não são acessíveis por retrovisores.
- Case isso somente se o grupo anterior também tiver casado.
- Estrutura de if-then-else dentro de ERs.
- A “configuração” de apenas partes da ER.
- Execução de trechos de linguagem de programação no meio de ERs.

E como tudo isso exige vários metacaracteres novos, alguns compostos de até cinco caracteres, exceto o conteúdo, as ERs ficaram muito feias e complexas, exercendo funções que a linguagem de programação deveria fazer, como condicionais, atrelamentos e definição de variáveis.

Daí, para tentar domar o monstro ilegível que eles próprios criaram (como “eles”, entenda Perl e Python), agora temos também:

- Comentários no meio das ERs.
- ERs estruturadas (com “indent”), ocupando várias linhas.

Pela mãe do guarda! É o progresso, daqui a pouco teremos ERs executáveis, ERs orientadas a objeto, ERs com botões e janelas, jogos 3D multijogadores pela Internet feitos somente com ERs...

Tá, não vou reclamar mais. Que fique aqui registrado meu protesto contra essa generalização das expressões regulares, já que elas não podem se

defender.

Primeiro, todos esses metacaracteres novos só foram possíveis porque as ERs têm certas brechas em construções antes impossíveis, que agora viraram a base para criações novas. Aqui está:

(?...)

Esta era uma estrutura inválida, pois você não pode tornar opcional a abertura de um grupo. Então, usá-la não teria problemas de compatibilidade, pois ainda não existia, foi esta a escolhida. Os metacaracteres novos têm a estrutura:

(?<identificador><conteúdo>)

Onde *identificador* é o que realmente diz com que tipo de metacaractere estamos lidando e *conteúdo* é o que será manipulado por esse metacaractere, e pode ser texto normal, outros metacaracteres, opções e até códigos de linguagem externa. Vamos utilizar os Simpsons como nossas cobaias de testes.

(?#*texto*)

Onde *texto* é um comentário, que é simplesmente ignorado pelo robzinho. Assim, podem-se colocar lembretes na ER como em

(?#o nome)Homer (?#e agora o sobrenome)Simpson

que sem os comentários é

Homer Simpson

(?:ER)

É como um grupo normal ( ) só que não é guardado nem incluído na contagem de grupos, ou seja, não é acessível com retrovisores ou \$1, pode ser considerado um grupo fantasma. Assim

^(Homer) (? : J \ . ) (Simpson)

casa o nome completo, mas \1 e \2 contêm Homer e Simpson, respectivamente.

(?=ER)

Não casa caracteres na posição atual, mas dá uma “espiada” adiante, e caso a ER embutida case, retorna sucesso. É como só apostar na loteria se você já souber o resultado. Por exemplo, a ER `Homer (?=Simpson)` só casará o Homer se for seguido de Simpson. Mas o sobrenome não faz parte do trecho casado, serviu apenas para checagem.

## **(?!ER)**

É o contrário do anterior, só casando um trecho se este não for seguido da ER embutida. Então `Homer (?!morreu)` casa o Homer do texto “Homer comeu”, mas não do “Homer morreu”. Para memorizar os dois últimos metacaracteres, veja seus identificadores: `=` e `!`, que lembram os operadores `==` e `!=`.

## **(?<=ER)**

## **(?<!ER)**

Estes dois são complementares aos dois anteriores, a diferença é que, em vez de espiar para frente, eles espiam para trás (note o `<` apontando para a esquerda). Então `(?<!Barney) Simpson` casará Simpson em “Homer Simpson”, mas não em “Barney Simpson”.

## **(?<nome>ER)**

Este é um grupo que possui um nome. Basta colocar este nome entre os sinais de menor e maior. Assim, você pode obter o trecho casado pelo grupo usando seu nome em vez de seu número. Útil para não confundir-se com os vários retrovisores `\1`, `\2`, `\3` e amigos. Um bom exemplo de uso é ao fazer uma expressão para casar horários, que de `([0-9][0-9]:[0-9][0-9])` cresce para `(?<horas>[0-9][0-9]):(?<minutos>[0-9][0-9])`.

## **(?*modificador*)**

Este metacaractere serve para mudar o comportamento padrão de uma expressão. Geralmente colocado bem no início da expressão, o modificador pode ser uma ou mais letras, sendo que cada letra liga uma funcionalidade diferente.

Letra	Significado



i	Ignorar a diferença entre maiúsculas e minúsculas
m	Faz o metacaractere ponto casar o \n
s	Faz as âncoras ^ e \$ casarem o \n
x	Permite inclusão de espaços e comentários
L	Levar em conta a localização do sistema (somente Python)
u	Levar em conta a tabela Unicode (somente Python)

O modificador (*?i*) é o mais utilizado, muito prático quando você não quer se preocupar com maiúsculas e minúsculas. Com ele, tanto faz se você usa `[A-Z]` ou `[a-z]`; de qualquer jeito, ambos casarão todas as letras.

Ao lidar com textos multilinha, use (*?s*) para fazer o metacaractere ponto casar a quebra de linha `\n`, o que normalmente não acontece. Assim, o seu curinga `.*` vai casar todo o texto. Na prática, é como se todo o texto virasse uma única linha. Já o (*?m*) faz as âncoras `^` e `$` casarem cada uma das linhas do texto. Assim, você pode usar `(?m)^[0-9]` para casar todas as linhas que começam com números.

Se a sua expressão ficar muito complicada, use o modificador (*?x*) para poder quebrá-la em várias linhas, alinhar metacaracteres e colocar comentários para ajudar a esclarecer o que faz cada trecho. Os espaços em branco e Tabs são ignorados e o caractere `#` é usado para iniciar um comentário. Para inserir espaços literais, você deve escapá-los ou usar `\s`. Veja um exemplo:

```
(?x)
# Expressão para casar horas hh:mm
^           # Início da string
\s*        # Espaços opcionais
[0-9]{2}    # Dois dígitos
:          # Separador
[0-9]{2}    # Dois dígitos
\s*        # Espaços opcionais
$          # Fim da string
```

A linguagem Python trouxe uma dupla de modificadores que ajudam a

casar nossas palavras em português, alterando o significado do barra-letra \w para incluir letras acentuadas: (?L) para levar em conta a localização de seu sistema, e (?u) para levar em conta a tabela Unicode.

Dependendo da linguagem, somente alguns destes modificadores são suportados. Uma alternativa para o ponto casar tudo, caso não haja o (?s), é a lista [\s\s] que casa qualquer caractere, incluindo o \n. Você pode ligar mais de um modificador ao mesmo tempo, por exemplo, (?mx) indica uma expressão multilinha e comentada.

### (?(*condição*)ER-sim|ER-não)

E aqui está o transgênico mutante, o if-then-else dos metacaracteres.

A condição geralmente é um número que referencia um grupo prévio. Se esse grupo casou, a condição é verdadeira e ER-sim é a ER da vez. Se a condição falhar, a ER-não é utilizada. Isso é basicamente usado para fazer amarrações e balanceamentos, utilizando condicionais.

É algo como “case um número entre possíveis parênteses, mas se tiver tem de ter o parênteses que abre E o que fecha”, ou seja, 69 e (69) são válidos, mas (69 e 69) não. Veja como fica a ER:

```
(\()?[0-9]+(?(1)\))
```

Isso porque nem preenchamos a possibilidade ER-não...

Se isso não é exagero, eu quero ser jardineiro, pois as plantas não terão teclados no futuro (será?). Ih, eu tinha dito que não reclamaria mais, né? Foi mal.

### (?{*código*})

E agora, a prova de que isso já foi longe demais, e o marco da perda definitiva de compatibilidade de ERs entre aplicativos: a possibilidade de colocar códigos Perl para serem executados no meio da ER. Por enquanto é só o Perl que ousou fazer isso.

Dá até coceira de estar escrevendo sobre isso, mas você pode colocar trechos de código, como um contador incremental, em várias partes de uma

mesma ER.

Vou colocar aqui um exemplo tirado do manual do Perl, estruturado e comentado, veja com seus próprios olhos:

```
$_ = 'a' x 8;
m<
  (?{ $cnt = 0 })          #inicializa
  (
    a
    (?{
      local $cnt = $cnt + 1;  #incrementa
    })
  )*
aaaa
  (?{ $res = $cnt })      # se ok, copia para uma var não-loca
>x;
```

## Precedência entre metacaracteres

Falamos, falamos, mas para fechar o estudo dos metacaracteres, faltou conhecer os relacionamentos entre eles, quem é mais forte, mais fraco, quem arrasta os outros...

É bem simples... Sabe na matemática, onde temos as ordens de precedência em que a multiplicação é mais forte do que a adição?

Por exemplo  $2+4*6$  é “quatro vezes seis, e depois soma com o dois”. Mesmo vindo depois, a multiplicação tem preferência. Com as ERs acontece o mesmo, seguindo estas regras:

Tipo de meta	Exemplo	Precedência
Quantificador	ab+	Maior
Concatenação	ab	Média
Ou	ab c	Menor

Ou seja, na situação  $ab^*$  não é “a com b, em qualquer quantidade”, mas, sim, “a, seguido de b em qualquer quantidade”, ou seja, a concatenação a

seguido de b não é mais forte que a quantificação, que rouba o b para ela.

Na última  $ab|c$ , em vez de “a, seguido de b ou c” é na verdade “ab ou c”, pois o ou é o mais fraquinho de todos, não puxa nada para o cesto dele.

Por isso que  $boa-tarde|boa-noite$  funciona, pois os caracteres se juntam e se grudam uns com os outros, e o ou não tem força para quebrar isso.

Como na matemática também, os parênteses servem para juntar na marra e dar força aos fracos. Por isso se diz que o grupo aumenta o poder do ou.

Só com a dobradinha  $grupo+ou$  é possível algo como  $boa-tard(e|b)oa-noite$ , o que não faz sentido, mas mostra que juntos eles conseguem quebrar a união da concatenação.

Com isso em mente, como fica a relação de forças em  $ab|cd^*$ ? Vamos colocar os “amigos” entre chaves para ilustrar essa quebra de braço:

**$ab|c\{d^*\}$**

O  $d$  é do  $*$  pela quantificação ser mais forte que a concatenação, então o c coitado, não tem força para puxar o seu amigo d para seu lado.

**$ab|\{c\{d^*\}\}$**

Agora o c fica na dúvida, mas se juntar com o  $|$  não dá, pois ele é o mais fraquinho de todos, então ele se junta com o  $d$  quantificado. Como o  $|$  já perdeu a briga à direita, ele olha para o outro lado e...

**$\{ab\}|\{c\{d^*\}\}$**

O b rapidinho se junta com o a (concatenação) para fugir do fracote. É como no primário onde sempre tem aquele cara desengonçado que fica por último na escolha dos jogadores para o futebol com bola de meia, o  $|$  fica com o que sobrou, de um lado  $ab$  e do outro  $cd^*$ .

Este mundo é mesmo muito cruel com os mais fracos, até os metacaracteres sofrem com essa discriminação :)



## Capítulo 4

# Os 6 mandamentos do Criador

No mundo das ERs, temos diversas leis não escritas que hora ou outra vão bater à sua porta e você verá que segui-las fará com que suas ERs sejam mais precisas e não falhem.

Essas leis são um misto de dicas de prevenção de problemas e ganhos de performance. Se você está começando, não se preocupe com essas regras. Mas se você já tem experiência com ERs, verá que essas leis podem lhe poupar estresse.

Sobre a performance, em situações normais, não é necessário se preocupar com a velocidade de uma ER, pois, independentemente de como você a faça, o resultado virá instantaneamente. Mas, quando aplicadas a vários arquivos, ou a um arquivo muito extenso, a demora pode ser grande.

Temos técnicas simples e complicadas de otimização. Como as complicadas são de difícil implementação e manutenção, às vezes, não compensando o custo-benefício de seu uso, não vamos vê-las. O assunto é simples e vamos tratá-lo de forma simples, sem testes gratuitos de performance (“benchmark”), detalhes específicos de certos programas e exceções raras.

As dicas que seguem, no geral, podem ser usadas em qualquer tipo de programa ou linguagem, pois são detalhes conceituais e não dependentes de

implementação.

## Não complique

Ao construir uma ER, lembre-se de que um dia alguém, provavelmente você mesmo, terá de dar manutenção a ela, para arrumar algum problema ou melhorá-la. Tendo isso em mente, evite fazer construções complicadas desnecessariamente.

Nem sempre a menor ER é a melhor, tudo vai depender do quão comentada ela está ou das habilidades de quem for mantê-la. Vamos ver um exemplo bem simples. Lembra na explicação do escape, quando vimos uma ER que casava um número de RG?

```
[0-9]\.[0-9]{3}\.[0-9]{3}-[0-9]
```

Note que o trecho para casar um ponto e três números seguidos `\.[0-9]{3}` se repete duas vezes, então, podemos agrupá-lo e aplicar as chaves, diminuindo o tamanho da ER:

```
[0-9](\.[0-9]{3}){2}-[0-9]
```

Note que nesse caso, essa “simplificação” da expressão acabou ficando não tão simples assim e exige um pouco de reflexão até você pescar exatamente o que ela faz. Foi vantagem ter diminuído seu tamanho? Não. E aquela do mercado?

```
(mini|(su|hi)per)?mercado
```

Será que, se a deixássemos mais simples, não ficaria mais fácil entendê-la? A mudança é pequena, mas veja como visualmente fica mais agradável e fácil:

```
(mini|super|hiper)?mercado
```



**ATENÇÃO:** Nem sempre a ER menor é a melhor.

Então muito cuidado ao colocar grupos dentro de grupos, quantificar grupos, usar chaves quando se pode usar o asterisco, entre outros. Procure

manter sua ER simples. Como dizem os gringos: KISS (Keep It Simple, Stupid), traduzindo: deixe simples, mané.

## Use o circunflexo

Sempre que possível, comece sua ER com o circunflexo. Como já vimos, o robzinho vai tentando casá-la, caractere por caractere, da esquerda para a direita, a partir do começo da linha. Então, o ponto inicial de pesquisa é o começo de linha.

Se você não colocar o circunflexo em sua ER, o robzinho tentará casá-la em qualquer parte da linha. Isso significa ir varrendo a linha, um por um, até chegar ao final; e caso não encontre o padrão, retorna falha na pesquisa.

Se você coloca o circunflexo na sua ER, forçando o casamento do começo de linha, se o primeiro componente da ER após o ^ já não casar com o primeiro caractere da linha, dali mesmo já retornará falha de pesquisa, sem precisar varrer o resto da linha.

Por exemplo, se você procura valores em reais, pode simplesmente dizer `R\$.` Mas se você sabe que os reais que lhe interessam estão sempre no começo da linha, diga isso com sua ER: `^R\$.` Assim, em um exemplo como:

```
R$ 200,00 : fósforos e velas  
           essenciais na crise de energia.  
           comprados das marcas mais baratas.  
R$ 100,00 : caixas de ovos vazias
```

Ambas as ERs casam as linhas 1 e 4 imediatamente, pois têm o `R$` já no começo. Mas nas linhas 2 e 3, onde não há nossos reais desejados, a primeira ER seria tentada em ambas, em todas as posições, até o final, para ver que falhou. Já a segunda, ao encontrar um espaço em branco no começo da linha, já retorna falha, pois ele não é um `R`.

Em um exemplo mais palpável, suponha que seu chefe tenha uma mesa enorme, com oito gavetas. E se ele lhe falar: “Me traga a pasta verde, está na minha gaveta”, ou então “Me traga a pasta verde, que está na última gaveta à direita. Na última, hein? Não fuça no resto de minhas coisas!”. Tem uma

diferença, não? :)

## Evite a lista negada

A lista negada é um grande aliado quando não se sabe exatamente que tipo de dado está em uma determinada posição.

Mas lembre-se: a tabela ASCII estendida tem 255 caracteres. Dizer algo como `[^:]` significa negar um caractere e permitir outros 254, o que muitas vezes é um exagero.

Essa abrangência toda pode trazer resultados negativos, casando partes incorretas. Sempre que possível, tente descobrir quais as possibilidades válidas de dados em uma determinada posição e cadastre todas elas dentro de uma lista normal.

Nesse exemplo, se o tipo de dado que não pode ser os dois-pontos forem letras, números e alguns símbolos, liste-os:

`[A-Za-z0-9,.( )%!]`

Assim, mais descritivo e preciso, se tiver algum caractere que não os listados, a ER vai falhar e você saberá que alguém fez caca onde não devia. Do contrário, o erro passaria despercebido.



**ATENÇÃO:** Não tenha preguiça de descobrir todas as possibilidades de uma posição.

## Evite o curinga

Quando pegamos o jeito com expressões regulares, é comum usar o `.` para qualquer situação, pois, como todo curinga que se preze, é uma mão na roda. Mas à medida que você vai usando ERs para coisas mais complicadas, você começa a perceber que a causa de grande parte de seus problemas foi ter usado o curinga guloso e genérico onde você poderia ter sido mais específico, e que ele casou o que não devia.



Nem sempre é fácil trocar um curinga por outra coisa. Supõe-se que se você já o usou, é porque precisava de “qualquer coisa”. Mas pare para pensar, esse qualquer coisa é realmente QUALQUER coisa? Lembre-se de que isso é muito abrangente, o tudo e o nada. Não seria apenas “qualquer letra em qualquer quantidade” ou “quaisquer caracteres fora espaços em branco”?

Percebeu? As listas são nossa opção para tirar o curinga, trocando-o por algo não tão abrangente. Então, se em um texto normal você procura parte de uma frase, o restante dela até o ponto-final não diga que é `.*\.`, mas `[^\.]*\.`, ou melhor: `[A-Za-z ,]*\.` Isso evita de o curinga casar além do ponto final da frase e ir até o ponto-final do parágrafo.

Lembra a nossa demonstração da gulodice em que o asterisco casou demais? Podemos evitar isso sendo mais específicos em nossa ER. Em vez de dizer `<.*>`, ou seja, uma marcação pode ter “qualquer coisa” antes do `>`, dizemos que pode ter “qualquer coisa fora o fechamento da marcação”. Invocaremos a lista negada para nos ajudar nessa supertarefa, assim: `<[^\>]*>`, ou mais visual:

```
um <b>negrito</b> aqui.  
...xxxxxxxxxxxxxxxxx <.*>  
...xxx.....xxx <[^\>]*>
```

## Seja específico

E agora a regra de ouro, aquela que acaba resumindo as outras, a mãe de todas: seja específico. Memorize bem isto: seja específico. De novo: seja específico.

Se você sempre tiver esta regra em mente ao construir uma ER, as chances de falha ficam muito reduzidas. Os metacaracteres são vários e servem para criarmos um universo de possibilidades para casarmos um texto, então o quente é fazer um universo restrito, onde todos os componentes fazem sua parte no todo, cada um com seu pedacinho.

Algumas regrinhas e dicas de como ser específico já foram vistas, mas basicamente, para isso, primeiro você deve saber exatamente que tipo de

dado procura. Um conhecimento do trecho que se quer casar acontece quando se pode responder a estas três perguntas:

- O que você quer casar?
- Em que quantidade?
- Em qual contexto ou posição?

Sabendo o que se quer, basta traduzir isso para uma ER, lembrando sempre de evitar generalizações como o ponto, o curinga, a lista negada, ignorar maiúsculas e minúsculas, não usar âncoras. Sempre descreva em detalhes suas intenções, delimitando e especificando bem sua ER.

Em outras palavras, se você está com fome, não diga simplesmente “Quero uma pizza”, diga: “Quero uma pizza de calabresa, sem cebola, tamanho médio, cortada em oito pedaços e com borda de Catupiry”. Percebeu a diferença? :)

## **Não seja afobado, seja ninja**

“Encere à direita, lixe à esquerda e pinte para cima e para baixo.”

Vamos ver uma maneira diferente e interessante de mostrar exemplos de expressões regulares: mostrando como funciona o processo criativo, passo a passo. A arte ninja milenar de criar ERs do nada, pela primeira vez demonstrada.

Mentalizando seu objetivo (horário, data, e-mail, número, telefone), comece a primeira tentativa tímida e abrangente, usando o ponto para se ter um esqueleto genérico do que se quer casar. Teste a ER assim mesmo.

Deu certo? Então agora você trocará alguns dos pontos para ser mais específico, de acordo com as regras do tipo de dado que você quer casar. E assim segue, devagar, sempre testando cada modificação e seguindo a passos curtos.

Ao chegar a um ponto em que já está bem específico, procure por alternativas, exceções – elas sempre existem. Aquele trecho da ER é realmente obrigatório? Não seria opcional?

E, quando você acha que a ER está pronta, chega um dado novo um pouquinho diferente e você vê que tinha esquecido que aquilo também era válido. Para incrementar a ER, suas armas são os grupos, o ou, o opcional e as chaves.

Se a sua ER ficar grande e cheia de alternativas, é sinal de que você está conseguindo dizer ao robzinho exatamente o que quer. E assim é que se fazem ERs complicadas, Daniel San, de grão em grão!

Tolo daquele que senta e quer escrever o todo de uma vez! A arte de criar ERs deve ser saboreada, sem pressa e com inspiração.

“Dê um passo após o outro, pequeno gafanhoto.”

```
hh:mm
|
| . . . .
|
| [0-9]{2}:[0-9]{2}
|
| [012][0-9]:[0-9]{2}
|
| [012][0-9]:[0-5][0-9]
|
| ([01][0-9]|2[0-3]):[0-5][0-9]
|
dd/mm/aaaa
|
| ..../.../....
|
| [0-9]{2}/[0-9]{2}/[0-9]{4}
|
| [0123][0-9]/[0-9]{2}/[0-9]{4}
|
| [0123][0-9]/[01][0-9]/[0-9]{4}
|
| [0123][0-9]/[01][0-9]/[12][0-9]{3}
|
| ([012][0-9]|3[01])/[01][0-9]/[12][0-9]{3}
|
| ([012][0-9]|3[01])/(0[1-9]|1[012])/[12][0-9]{3}
|
| (0[1-9]|1[12][0-9]|3[01])/(0[1-9]|1[012])/[12][0-9]{3}
|
usuario@dominio.zz
|
| . * @ . *
```

[^@]\*@[^@]\*

[^@]+@[^@]+

[A-Za-z0-9\_.-]+@[A-Za-z0-9\_.-]+

[A-Za-z0-9\_.-]+@[A-Za-z0-9\_]+\\.[a-z]{2,3}

[A-Za-z0-9\_.-]+@[A-Za-z0-9\_]+\\.[a-z]{2,3}

[A-Za-z0-9\_.-]+@[A-Za-z0-9\_]+\\.[A-Za-z]{2,3}

[A-Za-z0-9\_.-]+@[A-Za-z0-9\_]+\\.[A-Za-z]{2,4}

#### números

[0-9]+

-?[0-9]+

[-+]?[0-9]+

[-+]?[0-9]+,[0-9]{2}

[-+]?[0-9]+(,[0-9]{2})?

[-+]?[0-9]+\\.[0-9]+(,[0-9]{2})?

[-+]?[0-9]+\\.[0-9]{3}(,[0-9]{2})?

[-+]?[0-9]{3}\\.[0-9]{3}(,[0-9]{2})?

[-+]?[0-9]{1,3}\\.[0-9]{3}(,[0-9]{2})?

[-+]?[0-9]{1,3}(\\.[0-9]{3})?(,[0-9]{2})?

#### telefone

....-....

[0-9]{4}-[0-9]{4}

\\(\\.\\)[0-9]{4}-[0-9]{4}

\\(\\.\\) ?[0-9]{4}-[0-9]{4}

\\(0xx\\.\\) ?[0-9]{4}-[0-9]{4}

\\(0xx[0-9]{2}\\) ?[0-9]{4}-[0-9]{4}

| | (\\(0xx[0-9]{2}\\) ?)?[0-9]{4}-[0-9]{4}



## Capítulo 5

# Como lidar com...

Esta seção nos traz várias táticas e dicas para tratarmos de problemas genéricos e frequentes que, apesar de comuns, geram muitas dúvidas na hora de construir a ER.

Nada muito complicado, é simples até, mas são dicas que geralmente a documentação dos programas não nos dá.

Ao contrário dos mandamentos anteriores, em que sua vida pode seguir tranquila sem nunca conhecê-los, as dicas que seguem podem fazer a diferença de uma noite bem-dormida ou maldormida :)

### **Problemas com maiúsculas e minúsculas**

Verdade absoluta: as ERs são sensíveis a letras maiúsculas e minúsculas, levando em conta sua diferença. Esse detalhe pode ajudar ou atrapalhar, dependendo da atenção do criador da ER ou da estabilidade dos dados pesquisados.

Acontece que, às vezes, após meses de funcionamento sem problemas, uma ER falha. Depois de quebrar a cabeça nas partes mais complicadas dela, você percebeu que foi uma falha simples de maiúsculas e minúsculas, pois uma parte de sua ER era o trecho “jipe 4x4” e no texto agora estava “jipe 4X4”.

Este é um erro muito difícil de perceber, por ser tão trivial e visualmente

difícil de detectar, sobretudo se você estiver sob pressão para “arrumar isso logo”. Encontrado o problema, usamos a lista para saná-lo: `jipe 4[xx]4`. Mas um mês depois o texto muda novamente para “Jipe 4X4”, e assim vai... Como descobrir isso rapidamente?

Vários aplicativos e linguagens, como veremos adiante, têm modificadores para ignorar essa diferença entre maiúsculas e minúsculas (“ignore case”), e essa é nossa chave de ouro para descobrir se o problema é esse.

É simples. Se há algo errado e você não sabe o que é, dê este chute, colocando a opção de ignorar essa diferença, e veja se o problema some. Se sim, bingo! Basta revisar cada parte de sua ER que contenha letras ou listas e conferir o texto pesquisado para ver o que mudou.

Encontrado o problema, arrume-o e desligue a opção “ignorante”. São várias letras? Não tem `[Pp][Rr][Oo][Bb][Ll][Ee][Mm][Aa]`. Fica feio? Fica, mas é seguro, portanto, desligue a opção.

Conclusão: ignore apenas se for algo temporário, para testes, ou se você tiver muita certeza do que está fazendo.

## ERs pré-processadas e cruas

Algumas linguagens recebem a ER como um dado do tipo string, e não simplesmente como uma ER pronta. Essa string é primeiro interpretada pela linguagem, e só depois é passada ao robzinho. Mas o que exatamente isso quer dizer? Muitas coisas.

Primeiro, esse tratamento prévio não é algo específico das ERs, pois também acontece com qualquer string na linguagem, seja para ecoar uma mensagem na tela, seja para fazer indexação. Trechos da ER podem ser confundidos com variáveis e outras estruturas especiais, como, por exemplo, a ER `$nome` poderia ser expandida para o conteúdo da variável `$nome`. Mas o que geralmente pega mesmo é a interpretação de escapes, incluindo os barra-letas.

Isso nos afeta diretamente ao escrever uma ER, pois imagine que queremos

casar um `\t` literal, então escapamos o escape: `\\t`. Mas ao receber essa string, a linguagem primeiro a interpreta, e quando vê dois escapes seguidos, o que faz? O troca por apenas um, pois `\\` representa um escape literal. Com isso nosso robzinho recebe a ER `\t`, que por sua vez será interpretado como um Tab literal e nossa busca falhará. Nesse caso, temos de prever o pré-processamento e escapar duplamente `\\\\t`, para que o robzinho receba o `\t` que queríamos.

Felizmente, para que não precisemos ficar escapando tudo duplicado, a maioria dessas linguagens tem maneiras de se especificar uma “string crua” (“raw string”), que não é interpretada e é passada diretamente ao robzinho. Detalhes de como fazer isso estão no capítulo específico da cada linguagem, mais adiante. Mas essa característica também tem seu lado bom. Como alguns robzinhos não suportam os barra-letras, esse pré-processamento os reconhece e os converte, passando-os literais para o robô.

Concluindo, se a linguagem que você usa recebe as ERs como strings, descubra como deixá-las cruas ou fique com dor de cabeça de tanto escapar os escapes...

## Multilinha

Algumas linguagens possuem modificadores para que sua ER consiga lidar com um texto de várias linhas. Geralmente são dois os modificadores, um para tratar estas várias linhas como apenas uma, em que o metacaractere ponto também casa o `\n` (quebra de linha), e outra complementar para tratar como várias linhas, onde o circunflexo e o cifrão podem casar começo e final de qualquer uma dessas linhas contidas no texto, chamado multilinha. Vejamos:

```
$a = $b = $c = "linha 1\nlinha 2\nlinha 3";  
$a =~ s/^\./!!/g ; print "$a\n-----\n";  
$b =~ s/^\./!!/gs; print "$b\n-----\n";  
$c =~ s/^\./!!/gm; print "$c\n-----\n";  
# RESULTADO (normal, uma linha, multilinha)  
!!  
linha 2
```



linha 3

```
-----  
!!  
-----  
!!  
!!  
!!  
-----
```

Ah, nada como um exemplo para ilustrar conceitos... Esse trecho em Perl casa a ER `^.*` no texto de três linhas, fazendo substituições globais (modificador `g`), primeiro normal, depois com o modificador uma linha, e depois o multilinha.

Como era de se esperar, o primeiro, como não sabe o que é multilinha, casou apenas a primeira e não tocou nas outras linhas.

O segundo, como o ponto casa a quebra de linha, considerou o texto todo como apenas uma única linha e casou tudo, trocando todas por apenas um `!!`.

Já o último, que é multilinha, considerou o `\n` o fim de uma linha e casou as três linhas separadamente.

## Acentuação

Use classes POSIX. Use `\w`. Ponto final.

Maaaaaaas... se o seu aplicativo não reconhece ambos, ou seu sistema não está configurado para a localidade correta, há um remendo que não é 100%, mas pode ajudar.

Você pode usar uma lista com um intervalo aproximado, que pega todos os caracteres acentuados que queremos, porém traz consigo alguns lixinhos no meio.

Confira na tabela ASCII no fim do livro, que podemos usar o intervalo `À-ü` para englobar todos os acentuados, ou ainda `À-Û` e `à-ü` caso se queria só maiúsculas ou minúsculas. Os lixos que ganhamos de brinde usando esses intervalos são coisas como “`äåæËÏÎÐðÑ×÷ØÝÞß`”, mas como eles não são muito comuns em nossos documentos, a princípio não há muito problema.

Porém fique atento, se alguma coisa estiver errada pode ser que por azar o texto contenha um desses caracteres e você não possa usar o remendo, mas é difícil. Então, vamos à listagem:

Classe POSIX	Remendo
<code>[:lower:]</code>	<code>[a-zà-ü]</code>
<code>[:upper:]</code>	<code>[A-ZÀ-Ü]</code>
<code>[:alpha:]</code>	<code>[A-Za-zÀ-Ü]</code>
<code>[:alnum:]</code>	<code>[A-Za-zÀ-Ü0-9]</code>



## Capítulo 6

# Editores de texto e planilhas

Como já vimos em sua história, as expressões regulares surgiram como parte de um editor de textos, e hoje, décadas depois, são utilizadas nos mais diversos programas, como navegadores de Internet, leitores de e-mail, linguagens de programação e várias outras tarefas que envolvam manipulação de dados.

Porém, como poucas outras coisas neste planeta, as expressões regulares ainda são fiéis à sua origem e ainda reinam absolutas nos editores de texto, auxiliando as tarefas de procura de texto, na substituição de um texto por outro ou na aplicação de comandos diversos em partes específicas do documento.

Temos como exemplo o `ed`, um editor que ainda hoje só altera texto por meio de comandos de substituição utilizando ERs; mesmo que seja só para colocar um simples ponto-final esquecido, você tem de fazer um `s/$/. /.`.

Alguns outros editores atuais ainda têm um suporte bem tímido às ERs, principalmente os gráficos e voltados ao usuário doméstico, como é o caso do MS Word. Já outros, como o Vim e o Emacs, as utilizam massivamente para as mais diversas tarefas, tendo os programadores como seu público-alvo. Vamos dar uma olhadinha neles?

## Emacs

Chamar o Emacs de editor de texto é ser superficial, pois editar texto parece ser a coisa menos interessante para fazer neste programa que também lê e-mails, acessa a Internet, faz ftp, entre outros. Mas como nosso interesse aqui é ERs, é isso o que veremos.

As ERs são tratadas como strings, então valem as dicas já vistas para lidar com isso. Para complicar, ele usa a notação antiga, em que a maioria dos metacaracteres deve ser escapada para ser especial.

Então, juntando esses dois fatos, precisamos fazer `\\[\\(.*\\)\\]` para agrupar o conteúdo de um par de colchetes, o que normalmente seria `[(.*)\\]`.

Há vários comandos que trabalham com ERs, sendo `re-search-forward` e `re-search-backward` os comandos de busca nas linhas adiante e anteriores, respectivamente. Mas melhores que estes são os comandos similares que procuram enquanto você digita, já sabendo se sua ER está funcionando ou não antes de finalizá-la: `isearch-forward-regexp` e `isearch-backward-regexp`.

Como um editor de texto serve para alterar texto, temos o comando `replace-regexp` que se encarrega de fazer a substituição de padrões:

```
M-x replace-regexp <enter> \\(Gentalha!\\) <enter> \\& \\1 Prrrr! <enter>
```

Com essa sequência agrupamos a palavra `Gentalha!`, e com o escape especial `&` que referencia todo o trecho casado e o retrovisor `um`, que neste caso têm mesmo conteúdo, obtemos a frase clássica que o Seu Madruga ouve após apanhar: “Gentalha! Gentalha! Prrrr!”.

O que é realmente diferente de tudo no Emacs são suas “classes de sintaxe”, que são seus similares para as classes POSIX e um algo mais.

A sintaxe para acessar essas classes é `\\s<identificador>`, em que o identificador pode ser:

Identificador	Nome	Descrição
/	charquote	Escapa o próximo caractere

\	escape	Inicia um escape tipo C
(	open	Abre um bloco
)	close	Fecha um bloco
<	comment	Inicia um comentário
>	endcomment	Termina um comentário
'	quote	Marca um texto normal
"	string	Delimita uma string
-	whitespace	É branco
.	punct	É pontuação
w	word	É parte de uma palavra
—	symbol	Não é parte de palavra

O detalhe é que você mesmo pode alterar o conteúdo dessas classes antes de utilizá-las, dependendo de suas necessidades. O comando `describe-syntax` mostra os valores atuais dessas classes, para conferência.

E como era de se esperar `\s` (com S maiúsculo) casa exatamente o oposto, sendo `\s-` qualquer coisa fora brancos, e assim vai...

Ops! Quase me esqueci do mais importante: é GNU Emacs.

Mais informações são encontradas em:

<http://aurelio.net/regex/emacs/>

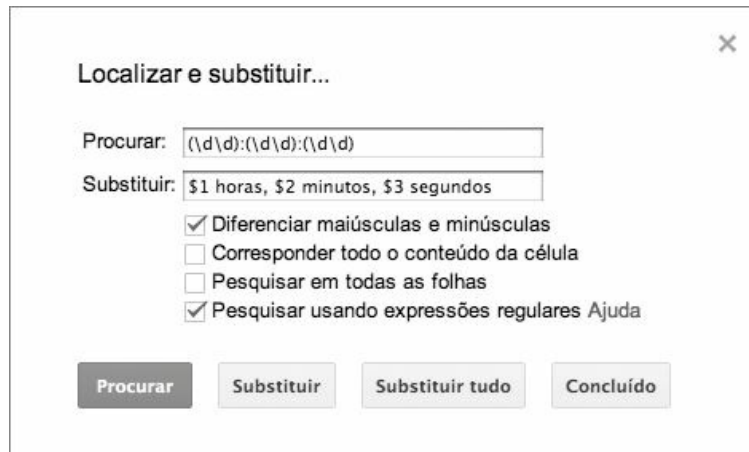
## Google Docs: Planilhas

Em edições anteriores deste livro, este tópico falava sobre o editor de textos do Google Docs, que em 2009 adicionou o suporte às expressões regulares. Porém, já no ano seguinte o editor foi reformulado e as expressões foram embora. Uma pena...

Mas a boa notícia é que nas planilhas o suporte às expressões chegou com toda força, pois, além de poder fazer pesquisas e substituições no conteúdo

das células, você também pode usá-las em suas fórmulas! Há três funções novas bem poderosas, específicas para expressões regulares, que veremos em detalhes adiante.

Mas, para começar leve, vamos brincar de pesquisar? Abra uma planilha, entre no menu **Editar** e escolha **Localizar e substituir....** Vai aparecer a janelinha de pesquisa, que será nossa companheira durante os experimentos.



Em primeiro lugar, marque a opção **Pesquisar usando expressões regulares** para turbinar a busca. O link **Ajuda** ao lado traz uma colinha com os metacaracteres, mas você não precisa mais disso, certo? :)

Tenha em mente que cada célula da planilha é considerada um trecho isolado de texto, como se fosse um parágrafo. Por isso não é possível fazer uma expressão para casar duas ou mais células adjacentes de uma vez.

O comportamento padrão da pesquisa é ignorar a diferença entre maiúsculas e minúsculas, então, se você buscar por [A-Z]+, as letras minúsculas também serão casadas. Para fazer uma pesquisa normal, onde M é diferente de m, selecione a opção **Diferenciar maiúsculas e minúsculas**.

A opção **Corresponder todo o conteúdo da célula** define se o casamento será completo ou parcial. Se ativada, todas as suas expressões deverão ser completas `^...$`, casando todo o conteúdo da célula. Para evitar surpresas, recomendo sempre usar expressões completas, independente do estado desta opção.

Uma vez casado o texto, você pode trocá-lo por outro no campo **Substituir:**. Use o botão **Substituir** para ir trocando uma ocorrência por vez, confirmando a cada alteração. Se você quiser substituir tudo em um único passo, use o botão **Substituir tudo**. Não tenha medo de perder textos, se algo sair errado, basta desfazer a substituição com o menu **Editar > Desfazer**.

No resultado das buscas, sempre é selecionada a célula toda, mesmo que o texto casado seja só uma pequena parte do conteúdo da célula. Mas não se preocupe, ao fazer uma substituição, somente o texto casado será trocado.

Para poder fazer buscas nas fórmulas, primeiro acesse o menu **Visualizar > Todas as fórmulas** para que elas sejam mostradas. Agora sim, pode procurar que ele vai encontrar. Só que infelizmente ele ainda não deixa fazer substituições em fórmulas. Quem sabe numa versão futura...

Agora que você já sabe como pesquisar e substituir textos, está na hora de aprender alguns detalhes importantes sobre o funcionamento dos metacaracteres:

- Use ^ e \$ para casar o início e o fim de cada célula.
- Use ^ e \$ para casar o início e o fim de cada linha interna de uma célula multilinha. Use Alt+Enter (ou Option-Enter no Mac) para inserir quebras de linha dentro de células.
- O ponto . não casa a quebra de linha manual em células multilinha. Assim sendo, o curinga .\* só irá casar a primeira linha em uma célula multilinha. Use o modificador (?s) no início da expressão para que o ponto case tudo.
- Não há como casar células vazias, o ^\$ não casa nada.
- O curinga .\* casa todas as células com conteúdo, ignorando as células vazias.
- O barra-letra \n tem um comportamento peculiar. Se usado no campo de pesquisa, casa a quebra de linha manual (Alt+Enter). Se usado no campo de substituição, insere literalmente os caracteres \ e n.

- Use `\b` para casar bordas de palavra. Assim, `\btecla\b` casa a palavra tecla, mas não teclado.
- Os retrovisores no texto substituto são indicados com o cifrão (\$1, \$2 etc.), porém você não pode utilizar retrovisores na própria expressão. Uma expressão para procurar números repetidos, como `(\d)\1`, dá erro.
- Use o retrovisor especial `&` para obter todo o trecho de texto casado pela expressão. Assim, substituindo um texto por `&&&`, ele será duplicado, e por `(&)`, ele será colocado entre parênteses.
- Tanto o `\w` quanto o `[[:alpha:]]` não casam letras acentuadas, mesmo que a planilha esteja configurada para o Brasil em **Arquivo > Configurações de planilha**. Mas há o metacaractere `\pL`, que casa letras normais e acentuadas, maiúsculas e minúsculas, baseado na tabela Unicode. Para negá-lo, casando tudo menos letras, use o P maiúsculo: `\PL`. Veja um exemplo, para casar palavras entre aspas: `"\pL+"`.
- Para implementar expressões regulares nas planilhas, o Google usa a RE2 (<http://code.google.com/p/re2/>), uma biblioteca em C++ que preza pela eficiência na execução. Os metacaracteres são os mesmos da linguagem Perl.

Tá, fazer buscas é legal, mas sabe o que é mais legal ainda? Fórmulas! O Google criou três funções específicas para usarmos as expressões: `REGEXMATCH` para testar se a expressão casou ou não, `REGEXREPLACE` para fazer substituições e `REGEXEXTRACT` para extrair o texto casado.

A função `REGEXMATCH` é a mais simples do trio. Você informa o texto (ou a célula que contém o texto) e a expressão, e a função testa se casou ou não, retornando `TRUE` ou `FALSE`. Boa para usar em funções que recebem testes, como a `IF`.

<code>=REGEXMATCH("1234"; "^d+\$")</code>	→ <code>TRUE</code>
<code>=REGEXMATCH("abcd"; "^d+\$")</code>	→ <code>FALSE</code>
<code>=IF(REGEXMATCH("1234"; "^d+\$"); "ok"; "falhou")</code>	→ <code>ok</code>
<code>=IF(REGEXMATCH("abcd"; "^d+\$"); "ok"; "falhou")</code>	→ <code>falhou</code>

Mas, claro, na vida real, o texto a ser casado vai estar em outra célula. E



quem sabe até a expressão também. Aí você pode referenciar direto pelo endereço da célula. Supondo a seguinte tabela, as três fórmulas são idênticas:

	A	B
1	1234	^\d+\$
2	abcd	^\w+\$

=REGEXMATCH("1234"; "^\d+\$") → TRUE

=REGEXMATCH(A1; "^\d+\$") → TRUE

=REGEXMATCH(A1; B1) → TRUE

Importante ressaltar que estas funções lidam apenas com textos (strings). Se você tentar aplicá-las em números, datas ou outros tipos de dados, vai dar erro. Primeiro, converta para texto antes de casar.

=REGEXMATCH(1234; "^\d+\$") → erro

=REGEXMATCH(TEXT(1234; "0"); "^\d+\$") → TRUE

A função REGEXREPLACE faz a substituição em textos, sendo muito útil para formatar informações. A substituição é sempre global, trocando todas as ocorrências, e conta com o suporte a retrovisores: \$0 para todo o trecho casado, e \$1, \$2, e amigos para os grupos.

- **Remover caracteres indesejados:**

=REGEXREPLACE("(xx 47) 1234-5678"; "[^0-9]"; "")  
→ "4712345678"

- **Formatar telefone:**

=REGEXREPLACE("4712345678"; "^(..)(....)(....)\$"; "(\$1) \$2-\$3")  
→ "(47) 1234-5678"

- **Truncar textos muito compridos:**

=REGEXREPLACE("Texto muito longo"; "^(.{8}).\*\$"; "\$1...")  
→ "Texto mu..."

A expressão regular também pode ser “montada”, aproveitando o conteúdo de outras células. Na expressão do último exemplo, o número 8 define o tamanho máximo do texto truncado. Que tal guardar este número numa célula

(digamos, B4), para poder aumentar/diminuir quando quiser?

```
=REGEXREPLACE("Texto muito longo"; "^(.{ " & B4 & "}).*$"; "$1...")
```

A função `REGEXEXTRACT` é uma novidade interessante no mundo das planilhas. Ela casa a expressão e retorna somente o trecho de texto casado. Se não casar nada, retorna erro.

```
=REGEXEXTRACT("Blink-182"; "\d+")      → "182"  
=REGEXEXTRACT("Blink-182"; "\d.*-")    → "Blink-"  
=REGEXEXTRACT("Blink-182"; "\d\d")     → erro: #N/A
```

Mas a melhor parte vem agora: se você usar grupos em sua expressão, então, em vez de texto, a função irá retornar um array (matriz) com o conteúdo casado por cada um dos grupos. Este array você usa em outras funções, diretamente ou via `ArrayFormula()`.

```
=REGEXEXTRACT("1:2:3"; "(.):(:):(.)")  
→ {"1", "2", "3"}  
=ArrayFormula(VALUE(REGEXEXTRACT("1:2:3"; "(.):(:):(.))))  
→ {1, 2, 3}  
=ArrayFormula(SUM(VALUE(REGEXEXTRACT("1:2:3"; "(.):(:):(.))))  
→ 6
```

Você também pode usar o array resultante diretamente em suas células. É uma maneira prática de extrair dados de um texto e já colocá-los em suas respectivas colunas. Eu DU-VI-DO que você não fique empolgado(a) ao ver este exemplo:

$f_x$	=regexExtract(A2; "^(.{*?}), (\d+), (.*) *[-/] *(\.)*\$")				
	A	B	C	D	E
1	Dados originais	Rua	Número	Cidade	Estado
2	Rua das Palmeiras, 123, Curitiba-PR	Rua das Palmeiras	123	Curitiba	PR
3	Avenida Brasil, 45, São Paulo - SP	Avenida Brasil	45	São Paulo	SP
4	Avenida Central, 678, Joinville/SC	Avenida Central	678	Joinville	SC
5					

Os dados estão na coluna A. Na coluna B é colocada a fórmula com a `REGEXEXTRACT`, que preenche automaticamente as colunas B, C, D e E com o conteúdo dos grupos da expressão regular. Mágica!

O funcionamento e os detalhes das expressões regulares dentro das funções é similar ao que já vimos para a busca. Mas há algumas diferenças importantes:

- As expressões `^$` e `.*`, que na busca não casam células vazias, nas funções, voltam ao seu comportamento normal: casam tanto células vazias quanto a string vazia `""`.
- Use o modificador `(?i)` no início da expressão para ignorar a diferença entre maiúsculas e minúsculas. Caso contrário, o casamento é sempre exato: `M` não casa `m`.
- Use o modificador `(?m)` no início da expressão para que as âncoras `^` e `$` casem o início e o fim de cada linha interna em células multilinha.
- Use o modificador `(?s)` no início da expressão para que o ponto `.` case a quebra de linha `\n` em células multilinha.
- O retrovisor especial para obter todo o trecho de texto casado pela expressão é `$&` na busca e `$0` nas funções. Esta é uma inconsistência que deve ser corrigida no futuro.

Mais informações são encontradas em:

<http://aurelio.net/regex/googledocs/>

## Microsoft Word

A partir do Office 97, usuários do Word também podem usufruir dos poderes das ERs. Faltam funções, alguns metacaracteres são diferentes, outros são iguais, mas agem diferente, outros são mais limitados, enfim, há muitas diferenças. Normal.

As ERs no Word são usadas nas funções **Localizar** e **Substituir**, ambas dentro do menu **Editar**. Basta selecionar a opção usar caracteres curinga. Aqui vai uma tabela comparativa dos metacaracteres normais e os do Word:

Metacaractere	Word	Nome
.	?	Ponto
[ ]	[ ]	Lista
[ ^ ]	[ ! ]	Lista negada
\	\	Escape

?	não tem	Opcional
*	não tem	Asterisco
+	@	Mais
{, }	{; }	Chaves
^	não tem	Circunflexo
\$	não tem	Cifrão
\b	<>	Borda
	não tem	Ou
( )	( )	Grupo
\N	\N	Retrovisor
. *	*	Curinga

Aqui vão 15 dicas valiosas se você pretende usar ERs no Word:

- Alguns metacaracteres são diferentes, como ., [^], e +, que são ?, [!] e @.
- A sintaxe das chaves usa ponto e vírgula em vez de vírgula para separar os números.
- Dentro da lista, o !, se não for o primeiro caractere (lista negada), deve ser escapado. O traço pode ser colocado no final ou escapado também. Assim, [abc\! -] e [!abc\ -\!] estão corretos.
- O opcional e o asterisco não existem, e não se podem usar as chaves para simulá-los, pois elas não aceitam zero, então {0;1} e {0;} são inválidos. Assim, não há como dizer que algo é opcional, ou pode não existir.
- O ou também não existe, então não é possível colocar alternativas simples como sim|não.
- Não há âncoras de linha! É impossível procurar uma palavra no começo ou no fim da linha. Há um ^| para indicar quebra de linha, e um ^n que indica quebra de coluna, mas ambos não funcionaram como âncora de fim de linha.

- É impossível colocar um grupo dentro do outro, então (sala ([0-9])) é inválido.
- Não temos o asterisco, mas temos .\*. O curinga no Word é somente \*, como para especificar arquivos: \*.txt.
- O \* e o @ só são gulosos quando necessário. Assim, em uma palavra como aaab, tanto a\* quanto a@ casam somente a, enquanto a\*b e a@b casam aaab.
- E contrariando o \* e o @, as chaves são sempre gulosas. Assim, na mesma palavra aaab, o a{1;3} casará aaa.
- A borda é feita com < e >, então <tecla> casa somente a palavra tecla, e não teclado.
- No botão **Especial** há vários metacaracteres diferentes que se pode incluir como travessão, elemento gráfico e espaço não separável.
- Na ajuda do Word (em português), diz-se que as bordas devem ser usadas com os grupos, assim: <(tecla)> – mas apenas ignore, não é necessário.
- Na ajuda do Word também se diz que é impossível procurar por um espaço em branco e que isso deve ser feito na forma {1;}. Ignore e inclua o espaço normalmente.
- E, por fim, na ajuda também em todos os exemplos de uso das chaves, os números são separados por vírgulas. Ignore e use o ; que é o correto.

Para obter mais informações no próprio Word sobre o assunto, procure na ajuda por “caracteres curinga”, ou siga o roteiro:

- Aperte F1.
- Espere o clipe animado fazer os fru-frus.
- Pesquise por: caracteres curinga.
- Escolha localizar e substituir texto ou formatação.
- Escolha ajustar uma pesquisa usando caracteres curinga.
- Clique em digite um caractere curinga.

Mais informações são encontradas em:

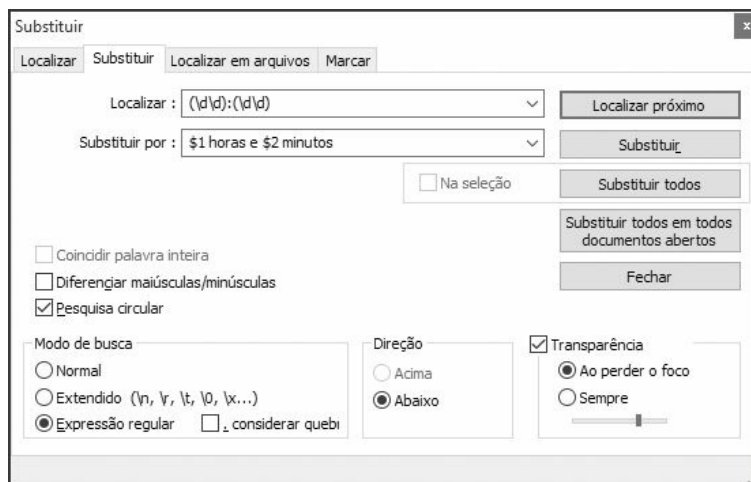
<http://aurelio.net/regex/word/>

## Notepad++

O Notepad++ é um editor de textos para Windows gratuito, lançado em 2003. Com uma interface simples e recursos poderosos, ele conquistou a preferência de programadores e usuários avançados que querem algo além do que o Bloco de Notas padrão oferece.

Versões mais antigas do editor já traziam o suporte às expressões regulares, porém elas funcionavam de maneira limitada. Felizmente isso mudou em 2012 com o lançamento da versão 6.0, que integrou ao editor a poderosa biblioteca PCRE (Perl Compatible Regular Expressions), trazendo todos os metacaracteres avançados do Perl para o Notepad++. Se você ainda usa uma versão anterior a essa, atualize-a o quanto antes!

Para ter acesso às expressões regulares, use as funções **Localizar** (Ctrl+F) ou **Substituir** (Ctrl+H). O painel que aparece traz as opções relacionadas. Certifique-se de que o item **Modo de busca** está marcado como **Expressão regular**. Pronto! Basta digitar suas expressões e ser feliz.



O comportamento padrão da busca é tratar letras maiúsculas e minúsculas como iguais, então a expressão [a-z] também vai casar letras maiúsculas. Se você preferir uma busca mais restrita, em que a expressão anterior case apenas as letras minúsculas, marque a opção **Diferenciar**

## maiúsculas/minúsculas.

Outra maneira de escolher como casar maiúsculas/minúsculas é colocar dentro da própria expressão, já no início, o modificador `(?i)` para casar maiúsculas e minúsculas, ou o modificador `(?-i)` para casar somente as letras especificadas na expressão. Por exemplo, `(?i)win` casa WIN, win, Win, wIN e outras variações, enquanto `(?-i)win` casa somente Win.

Sempre que você estiver lidando com caracteres invisíveis ou brancos, como tabulações (Tab), quebras de linha e espaços em branco, ligue a opção de menu **Visualizar > Mostrar símbolo > Exibir todos os caracteres** para poder enxergá-los. Assim, ficará mais fácil perceber o que está sendo (ou não) casado.

Ao ligar esta opção, você perceberá que no final das linhas irá aparecer LF ou CRLF. Este é o terminador de linha, que é diferente no Unix e no Windows. Para casar estes caracteres especiais, use `\n` para representar o LF e `\r\n` para o CRLF. A tabulação (Tab) é casada pelo `\t`.

Falando em quebras de linha, lá no painel da busca há uma opção estranha chamada `. considerar queb.` Apesar do nome truncado, o seu significado é “o metacaractere ponto também deve casar a quebra de linha”. Esta opção muda o comportamento do ponto, que normalmente casa tudo, exceto a quebra de linha.

E qual a utilidade disso? Em geral, serve para pesquisar em múltiplas linhas, como por exemplo usar `<p>.*?</p>` para casar parágrafos num arquivo com códigos HTML. Com a opção ligada, esta expressão irá funcionar mesmo quando a tag de fechamento `</p>` estiver várias linhas abaixo da tag de abertura `<p>`.

Assim como no caso das maiúsculas/minúsculas, este comportamento do ponto também pode ser definido diretamente dentro da própria expressão, usando modificadores no início. Use `(?s)` para fazer o ponto casar quebras de linha, ou `(?-s)` para que tenha seu comportamento normal.

Felizmente, a acentuação não é um problema. Tudo funciona normalmente,

conforme esperado. O metacaractere `\w` também casa letras acentuadas, então a expressão `\w+` pode ser usada para casar as palavras de um texto em português. Se precisar de algo mais específico, há o `\l` para casar somente as letras minúsculas, inclusive acentuadas, e o `\u` para as maiúsculas. Se preferir, você também pode usar as classes POSIX, como `[:alpha:]`, `[:lower:]` e `[:upper:]`.

Ainda sobre acentuação, há um metacaractere interessante para ser usado dentro de listas, que casa todas as variações de uma letra: maiúsculas, minúsculas, com ou sem acento. O formato é `[=x=]`, em que `x` é uma letra qualquer. Por exemplo, `[=a=]` casa A, a, À, Á, Â, Ã, à, á, â, ã. Assim, para casar todas as variações da palavra maçã, você pode fazer `[Mm][Aa][=c=]` `[=a=]`. Para casar todas as vogais, inclusive acentuadas, faça: `[=a=]` `[=e=]` `[=i=]` `[=o=]` `[=u=]`.

Uma funcionalidade interessante do painel de busca é a aba **Marcar**. Você escreve sua expressão e, ao apertar o botão **Localizar todos**, serão colocados em destaque todos os trechos de texto que casaram com ela. É um comportamento parecido com ferramentas online como o [RegexPal.com](http://RegexPal.com), útil para visualizar rapidamente tudo o que sua expressão está casando.

Na aba **Substituir** há um campo adicional para especificar qual texto irá substituir os trechos casados pela expressão. Ali é basicamente texto puro, com algumas exceções:

- Se você usou grupos na expressão, pode referenciá-los com os retrovisores `$1`, `$2`, `$3` etc. Por exemplo, para trocar datas no formato brasileiro (dd/mm/aaaa) para o formato internacional (aaaa-mm-dd), procure por `(..)/(..)/(...)` e substitua por `$3-$2-$1`.
- Use a notação com chaves (`${1}`, `${2}` etc.) se precisar grudar algum número logo após o retrovisor; por exemplo, `${1}00`.
- Use o retrovisor especial `$0` para obter todo o trecho de texto casado pela expressão. Assim, substituindo um texto por `$0$0`, ele será duplicado, e por `[$0]`, ele será colocado entre colchetes.



- Grupos nomeados são referenciados com uma sintaxe diferente: `${nome}`. O exemplo anterior da data internacional ficaria `${ano}-${mes}-${dia}`.
- Use `$$` ou `\$` para inserir um cifrão literal.
- Use `\(` e `\)` para inserir parênteses literais.
- Use `\\` para inserir uma contrabarra literal.
- Use `\t`, `\r` e `\n` para inserir Tabs e quebras de linha.
- Use `\l` para converter a próxima letra em minúscula.
- Use `\u` para converter a próxima letra em maiúscula.
- Use `\L` e `\U` para converter o próximo texto em minúscula ou maiúscula, respectivamente. Use `\E` para finalizar a conversão. Por exemplo, para converter em maiúsculas o conteúdo dos grupos 1 e 2, use `\U$1$2\E`.

Para finalizar o assunto, mais alguns detalhes importantes sobre o uso das expressões neste editor:

- Use `\b` para casar bordas de palavra. Assim, `\btecla\b` irá casar a palavra `tecla`, mas não `teclado`.
- Os barra-letras especiais como `\n`, `\r` e `\t` também funcionam dentro de listas. Por exemplo, use `[\t]` para casar tabulações e espaços em branco.
- Os retrovisores no texto substituto são indicados com cifrão (`$1`, `$2` etc.), porém, se você for utilizar retrovisores na própria expressão, use contrabarras (`\1`, `\2` etc.). Por exemplo, para procurar números repetidos como 86-86, use `(\d+)-\1`.
- A busca especial **Localizar > Busca incremental** (Ctrl+Alt+I) não suporta expressões regulares, somente texto normal.
- No painel de pesquisa, há três opções para o modo de busca: Normal, Estendido e Expressão Regular. O modo estendido era necessário no passado, quando as expressões não suportavam os barra-letras como `\n` e

\r. Hoje, com PCRE, tudo é suportado normalmente, e este modo estendido pode ser ignorado.

Mais informações são encontradas em:

<http://aurelio.net/regex/npp/>

## (Libre|Br|Open)Office Writer

A suíte OpenOffice, que no Brasil já foi chamada de BrOffice e agora mudou para LibreOffice, conta com alguns programas para a edição de documentos, planilhas e apresentações. Isso talvez não seja novidade para você. Mas você sabia que também pode usar as expressões regulares nesses programas?

No processador de textos (Writer), você pode pesquisar por expressões em seu documento, além de poder fazer alterações automáticas (substituições) de textos com facilidade. Você também pode usar expressões regulares em macros, na pesquisa de registros do Base e em várias funções do Calc, incluindo COUNTIF, HLOOKUP, LOOKUP, MATCH, SEARCH, SUMIF e VLOOKUP.

Agora nosso foco será o processador de textos, e no próximo tópico veremos as planilhas em detalhes. Abra o Writer para poder acompanhar e testar as explicações. Pronto? Então, abra um documento, entre no menu **Editar** e escolha **Localizar e substituir....**

O painel que aparece será seu companheiro no uso das expressões. Mas, em seu comportamento padrão, ele apenas procura texto normal. Para mudar isso, aperte o botão **Mais opções** e marque o item **Expressões regulares**.



Agora sim, estamos em casa. Digite as expressões no campo **Procurar por** e use os botões **Localizar** e **Localizar todos** para encontrar os trechos do texto que casam com a expressão.

O comportamento padrão da busca é tratar letras maiúsculas e minúsculas como iguais, então a expressão `[a-z]+` também vai casar letras maiúsculas. Se você preferir uma busca mais restrita, em que a expressão anterior case apenas as letras minúsculas, marque a opção **Diferenciar maiúsculas de minúsculas**.

Uma vez casado o texto, você pode trocá-lo por outro no campo **Substituir por**. Use o botão **Substituir** para ir trocando uma ocorrência por vez, confirmando a cada alteração. Se você quiser substituir tudo em um único passo, use o botão **Substituir todos**. Não tenha medo de perder textos, se algo sair errado, bastará desfazer a substituição com o menu **Editar**

## > Desfazer.

A partir da versão 2.4 da suíte, você também pode usar retrovisores na substituição, referenciados por \$1, \$2, \$3 etc. Por exemplo, para trocar datas no formato brasileiro (dd/mm/aaaa) para o formato internacional (aaaa-mm-dd), procure por (...) / (...) / (...) e substitua por \$3-\$2-\$1. Use o retrovisor especial \$0 para obter todo o trecho de texto casado pela expressão. Assim, substituindo um texto por \$0\$0, ele será duplicado, e por (\$0), ele será colocado entre parênteses.

Sempre que você estiver lidando com caracteres invisíveis ou brancos, como tabulações (Tab), quebras de linha, quebras de parágrafo e espaços em branco, ligue a opção **Exibir > Caracteres não-imprimíveis** para poder vê-los. Isso manterá seus cabelos no lugar por mais tempo. :)

Esse é o modo de uso básico das expressões. Agora, você já sabe como pesquisar e substituir textos. Aprenda alguns detalhes importantes sobre os metacaracteres:

- Use ^ e \$ para casar o início e o fim dos parágrafos. Lembre-se: cada parágrafo é, na verdade, uma longa linha quebrada automaticamente, de acordo com o tamanho da página.
- Use ^ e \$ para casar o início e o fim das linhas, caso você tenha usado a quebra de linha manual (Shift+Enter) para separá-las.
- Use ^\$ para casar parágrafos vazios.
- Use o \$ sozinho para casar a marcação de fim de parágrafo. Assim, você pode trocá-lo por um espaço em branco, juntando todos os parágrafos em apenas um.
- Use \< e \> para casar bordas de palavra. Assim, \<tecla\> casa a palavra tecla, mas não teclado.
- Os barra-letas \d e \w não funcionam, então use [0-9] e [A-Za-z].
- Use o \x para casar um caractere pelo seu código hexadecimal, com quatro dígitos. Por exemplo, o metacaractere \x0041 casa a letra A.

Funciona dentro e fora de listas. Veja os códigos no menu **Inserir > Caractere especial....**

- Use o \t para casar o caractere de tabulação (Tab). Se for dentro de uma lista, não vai funcionar, então será preciso usar a notação hexadecimal [\x0009].
- O barra-letra \n é mais complicado. Se usado no campo de pesquisa, ele casa a quebra de linha manual (Shift+Enter). Se usado no campo de substituição, insere uma quebra de parágrafo (Enter). É um comportamento inconsistente e confuso.
- O metacaractere ponto . não é multilinha. Então, os curingas .\* e .+, apesar de gulosos, não casam a quebra de linha manual (\n).
- Os retrovisores no texto substituto são indicados com cifrão (\$1, \$2 etc.), porém, se você utilizar retrovisores na própria expressão, use contrabarras (\1, \2 etc.). Por exemplo, para procurar números repetidos como 86-86, use ([0-9]+)-\1.
- Use \\$ para inserir um cifrão literal na substituição. Do mesmo modo, use \\ para inserir uma contrabarra.
- Use as classes POSIX para casar acentuação. Porém, em vez de colocá-las dentro de uma lista como de costume [[:alpha:]], elas só funcionam se estiverem fora, sozinhas [[:alpha:]]. Há alguns casos em que mesmo assim não funciona, então, para garantir, sempre coloque dentro de um grupo ([[:alpha:]]). Bizarro!

Alguns desses detalhes são, na realidade, problemas de funcionamento. A equipe de desenvolvimento sabe disso e tem como meta fazer uma “reforma geral” em seu código, ainda na série 3.x da suíte. Pode ser que, em um futuro próximo, alguns desses itens sejam corrigidos.

Mais informações são encontradas em:

<http://aurelio.net/regex/ooo/>

## (Libre|Br|Open)Office Calc

Agora vamos falar sobre o Calc. Você também pode usar expressões regulares no painel **Localizar e substituir**, fazendo buscas e edições avançadas nas células de suas planilhas.

Como o Calc usa a mesma biblioteca que o Writer, todos os metacaracteres e detalhes vistos até agora continuam valendo. A grande diferença é que cada célula da planilha é considerada um trecho isolado de texto, como se fosse um parágrafo do Writer. Isso nos traz alguns detalhes novos:

- Use ^ e \$ para casar o início e o fim de cada célula.
- Use ^ e \$ para casar o início e o fim de cada linha interna de uma célula multilinha. Use Control+Enter (ou Command-Enter no Mac) para inserir quebras de linha dentro de células.
- Não há como casar células vazias, o ^\$ não casa nada. Este comportamento é intencional, para evitar casar a planilha toda, já que todas as células iniciam-se vazias.
- O curinga .\* casa todas as células com conteúdo, ignorando as células vazias.
- O barra-letra \n tem um comportamento peculiar. Se usado no campo de pesquisa, casa a quebra de linha manual (Control+Enter). Se usado no campo de substituição, insere literalmente os caracteres \ e n.
- No resultado das buscas, sempre é selecionada a célula toda, mesmo que o texto casado seja só uma pequena parte do conteúdo da célula. Mas não se preocupe, ao fazer uma substituição, somente o texto casado será trocado.

Entretanto, não para por aqui. No Calc, além da busca, você também pode usar expressões regulares nos filtros e em algumas funções.

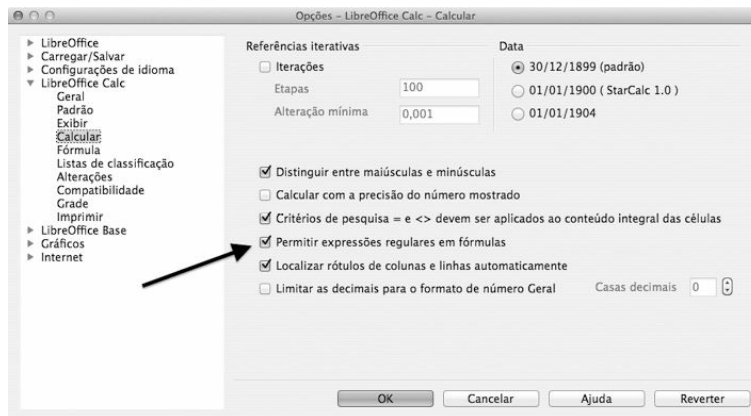
Para acessar o painel de filtros, vá ao menu **Dados > Filtro**. Tanto o filtro padrão quanto o avançado possuem uma opção para ligar o suporte às expressões regulares. No campo **Condição**, escolha uma opção apropriada para a comparação de textos: use = ou <> para casar todo o conteúdo da célula, e **Contém** ou **Não contém** para casamentos parciais.

As funções são usadas dentro de fórmulas, como por exemplo =SOMA(B7:B12). O suporte às expressões regulares foi adicionado às funções que fazem pesquisa ou comparação de texto, são elas (nomes em português e inglês):

Texto	
PESQUISAR	SEARCH
Matemática	
CONT.SE	COUNTIF
SOMASE	SUMIF
Planilha	
CORRESP	MATCH
PROC	LOOKUP
PROCH	HLOOKUP
PROCV	VLOOKUP
Banco de dados	
BDCONTAR	DCOUNT
BDCONTARA	DCOUNTA
BDDSVPA	DSTDEVP
BDEST	DSTDEV
BDEXTRAIR	DGET
BDMULTIPL	DPRODUCT
BDMÁX	DMAX
BDMÉDIA	DAVERAGE
BDMÍN	DMIN
BDSOMA	DSUM
BDVAREST	DVAR

BDVARP	DVARP
--------	-------

Em primeiro lugar, entre nas opções do programa (**Ferramentas > Opções** ou **OpenOffice > Preferências...** no Mac), clique em **OpenOffice Calc > Calcular** e certifique-se de que a opção **Permitir expressões regulares em fórmulas** está ligada.



Neste mesmo painel, logo acima está a opção **Critérios de pesquisa = e <> devem ser aplicados ao conteúdo integral das células**, que significa que casamentos parciais não serão válidos. Com esta opção ativada, supondo uma célula com o conteúdo “Rua das Palmeiras, 123”, a expressão parcial `[0-9]+` falhará, mas a completa `^[0-9]+$` casará. Com a opção desativada, ambas casarão.

Não é muito sábio vincular o funcionamento de suas fórmulas a uma opção tão fácil de ser alterada. Por isso, evite fazer expressões parciais. Sempre faça expressões completas `^[0-9]+$`, pois elas funcionarão da mesma forma, independentemente do estado dessa opção.

Ainda neste mesmo painel de configuração, note que a primeira opção é **Distinguir entre maiúsculas e minúsculas**. Você pode ignorá-la, pois ela não vale para expressões regulares em funções. Nas funções, o casamento é sempre feito de forma “ignorecase”, tratando OFFICE e office como iguais. Também não adianta usar `[ :upper: ]` e `[ :lower: ]`, aqui ambos casam `[A-Za-z]`.



	A	B	C	D	E
1	Endereço	Moradores		Fórmula	Resultado
2	Rua das Palmeiras, 123	4		=CONT.SE(A2:A4; "^.*[0-9]\$" )	3
3	Avenida Brasil, 45	5		=SOMASE(A2:A4; "^.*[0-9]\$" ; B2:B4)	15
4	Avenida Central, 678	6		=PESQUISAR("^rua.*\$" ; A2)	1
5				=PROC("^avenida b.*\$" ; A2:A4)	Avenida Brasil, 45
6				=PROCV("^.*central.*\$" ; A2:B4; 2)	6
7					

Veja nos exemplos como as expressões são sempre colocadas entre aspas, estão na forma completa `^...$` e sempre casam maiúsculas e minúsculas.

As expressões regulares só funcionarão nas funções mencionadas, mas não em outros lugares de sua fórmula. Por exemplo, para fazer um teste simples para verificar se uma expressão casou ou não com uma célula, não use a comparação direta (`A2="..."`), use a função `CONT.SE`.

`=SE(A2="^rua.*$")` → FALSO  
`=SE(CONT.SE(A2; "^rua.*$"))` → VERDADEIRO

Eu sempre usei as fórmulas em inglês e minha cabeça deu um nó para fazer todos estes exemplos com os nomes em português. Aqui vai uma colher de chá para quem está no mesmo barco:

`=IF(A2="^rua.*$")` → FALSE  
`=IF(COUNTIF(A2; "^rua.*$"))` → TRUE  
`=COUNTIF(A2:A4; "^.*[0-9]$" )` → 3  
`=SUMIF(A2:A4; "^.*[0-9]$" ; B2:B4)` → 15  
`=SEARCH("^rua.*$" ; A2)` → 1  
`=LOOKUP("^avenida b.*$" ; A2:A4)` → "Avenida Brasil, 45"  
`=VLOOKUP("^.*central.*$" ; A2:B4; 2)` → 6

Mais informações são encontradas em:

<http://aurelio.net/regex/ooo/>

## Vim

O Vim herdou a base das expressões do vi e estendeu muito seu poder, criando metacaracteres novos e permitindo o uso de ERs na maioria de seus comandos, como endereçamento. Tudo o que você precisa saber sobre ERs no Vim está documentado no próprio programa, bastando digitar `:help regexp` para ter acesso a essas informações.

A primeira dica é ligar duas opções imbatíveis, que são a `hlsearch` e `incsearch`. A primeira deixa iluminado (“highlight”) o texto casado, para termos uma visualização do que nossa ER casou, e a segunda faz o Vim ir iluminando e mostrando o texto casado dinamicamente, enquanto você digita a ER! Experimente, é muito bom. Digite `:set hls is`.

Além destas, temos opções que modificam a sintaxe dos metacaracteres, são elas: `magic` e `nomagic`. A primeira é a opção-padrão, e é a que se aconselha usar, não mude. Ao usar a segunda, suas ERs serão tão cheias de escapes que serão imprestáveis em outros programas.

A função de pesquisa é feita pelo comando `/` e o `?`, e a de substituição, pelo `:s///`. O detalhe é que a substituição vale apenas para a linha atual onde está o cursor, a não ser que você selecione o texto desejado ou aplique um endereçamento que dirá ao comando `s` em quais linhas fazer a substituição.

Esse endereçamento vem imediatamente após o `:` e pode ser números de linha, padrões de texto entre barras e alguns caracteres representantes de: linha atual (`.`), última linha (`$`) e arquivo todo (`%`). Veja alguns exemplos:

Comando	Descrição
<code>:s/a/b/g</code>	Troca todos os a por b na linha atual
<code>:1,5s/^/#/</code>	Comenta as cinco primeiras linhas
<code>:.,\$s/^/#/</code>	Comenta até o final do arquivo
<code>:%s/^./</code>	Apaga a primeira letra de cada linha
<code>:%s/^./\u&amp;/c</code>	Torna maiúsculo o primeiro caractere

Esse último é utilíssimo. Colocando um `c` no final de qualquer substituição, o Vim lhe pedirá uma confirmação antes de fazer qualquer uma das substituições. Você vai respondendo sim ou não e vai vendo o texto ser alterado. Evita substituições cegas e vicia, cuidado! Veja `:help :s_flags` para conhecer outros modificadores.

No Vim, apenas as chaves têm similares não-gulosos. Mas como já vimos que com elas conseguimos emular todos os outros quantificadores, isso não é

problema. A sintaxe é simplesmente adicionar um traço como primeiro caractere dentro delas.

Metacaractere	Nome
\{ -, 1 }	Opcional não-guloso
\{ - }	Asterisco não-guloso
\{ - 1, }	Mais não-guloso
\{ - n, m }	Chaves não-gulosas

O editor possui também diversos arquivos de sintaxe para vários tipos de arquivo, onde são guardadas as regras complicadas que dizem “o que é válido onde”, e fazem os arquivos ficarem coloridos, respeitando a sintaxe da linguagem de programação ou o formato do arquivo.

Adivinha como são feitas essas regras complicadas? Com ERs, é claro! Este é um tópico muito interessante e exige conhecimentos sólidos do assunto. Caso queira dar uma olhada, todos os arquivos de sintaxe estão no subdiretório `syntax` e têm a extensão `.vim`.

Mais informações são encontradas em:

<http://aurelio.net/regex/vim/>



## Capítulo 7

# Aplicativos

### Apache HTTPd

O Apache é há anos o servidor HTTP mais utilizado da internet. Lançado em 1995, hoje ele traz uma vasta gama de funcionalidades. Seu comportamento é controlado por centenas de configurações, chamadas diretivas, e a boa notícia para nós é que várias delas aceitam expressões regulares:

Módulo	Diretiva	Formato
core	<Directory>	~ er
	<DirectoryMatch>	er
	<Files>	~ er
	<FilesMatch>	er
	<Location>	~ er
	<LocationMatch>	er
	<If>	... =~ /er/
	<ElseIf>	... =~ /er/
mod_authz_core	Require	... =~ /er/
	AliasMatch	er

mod_alias	RedirectMatch	er
	ScriptAliasMatch	er
mod_headers	Header	er
	RequestHeader	er
mod_log_debug	LogMessage	... =~ /er/
mod_log_config	CustomLog	... =~ /er/
mod_proxy	<ProxyMatch>	er
	ProxyPassMatch	er
	ProxyRemoteMatch	er
mod_rewrite	RewriteCond	er
	RewriteRule	er
mod_setenvif	BrowserMatch	er
	BrowserMatchNoCase	er
	SetEnvIf	er
	SetEnvIfExpr	... =~ /er/
	SetEnvIfNoCase	er
mod_substitute	Substitute	s///
mod_version	<IfVersion>	~ er

Independentemente do formato usado pela diretiva, a sintaxe da expressão regular em si é sempre a mesma. Você pode abusar dos metacaracteres modernos e avançados do Perl, pois o Apache usa a biblioteca PCRE (Perl Compatible Regular Expressions).

Em seus testes, lembre-se de que algumas dessas diretivas funcionam somente no arquivo de configuração central `httpd.conf`, enquanto outras também podem ser colocadas em arquivos `.htaccess` dentro de seu site. Na dúvida, consulte o site indicado no final deste tópico, que traz links para a documentação do Apache.

Há três diretivas para casar arquivos do site e aplicar um conjunto de regras sobre eles:

- <Files> para casar nomes de arquivos.
- <Directory> para casar nomes de pastas.
- <Location> para casar endereços (URL).

Normalmente essas diretivas só aceitam texto normal, mas cada uma possui uma versão especial para usar com expressões regulares: <FilesMatch>, <DirectoryMatch> e <LocationMatch>. O funcionamento da versão normal e da versão Match é o mesmo, o único diferencial é o suporte às expressões.

Destas três, a <FilesMatch> é a única que funciona dentro do .htaccess, então é uma boa candidata para iniciar nossos testes. Que tal um pouco de diversão? Vamos quebrar todas as imagens do site:

```
<FilesMatch "\.(gif|jpg|png)$">
    deny from all
</FilesMatch>
```

Em primeiro lugar, perceba que a expressão regular foi colocada entre aspas. Faça isso sempre, sem exceção, para evitar problemas com espaços em branco ou algum caractere maluco. Isso é bem importante, pois, se você cometer qualquer erro nessas diretivas, o seu site vai ficar fora do ar imediatamente, mostrando um feioso “Internal Server Error” para seus visitantes. Aspas. Sempre.

```
<FilesMatch \.(gif|jpg|png)$>      # n00b
<FilesMatch "\.(gif|jpg|png)$">    # ninja
```

Para fazer a sua expressão ignorar a diferença entre maiúsculas e minúsculas, coloque no início o metacaractere modernoso (?i). Assim, pegaremos também os arquivos com as extensões GIF, JPG e PNG, em maiúsculas.

```
<FilesMatch "(?i)\.(gif|jpg|png)$">
    deny from all
</FilesMatch>
```

É muito importante entender que em todas as diretivas com suporte às

expressões regulares o casamento é sempre parcial. Notou que no exemplo a expressão casa apenas a extensão do arquivo, e não o nome e caminho completo? Isso é um casamento parcial, e ele é sempre válido.

Mas esse casamento parcial confunde usuários acostumados à versão normal da diretiva, que, pelo contrário, só faz casamentos completos. Veja a diferença:

<code>&lt;Files "foto.jpg"&gt;</code>	<code>&lt;FilesMatch "foto\.jpg"&gt;</code>
<code>/foto.jpg</code>	<code>/foto.jpg</code>
<code>/imagens/foto.jpg</code>	<code>/imagens/foto.jpg</code>
<code>/foo/bar/baz/foto.jpg</code>	<code>/foo/bar/baz/foto.jpg</code>
	<code>/minhafoto.jpg</code>
	<code>/foto.jpg.zip</code>
	<code>/foto.jpg/foo.txt</code>
	<code>...</code>

Enquanto a diretiva `<Files>` casa o arquivo `foto.jpg`, em qualquer pasta, a `<FilesMatch>` casa o texto `foto.jpg` em qualquer lugar do path. Esta diferença de funcionamento também acontece com as outras diretivas que lidam com arquivos, pastas e redirecionamentos.

Meu conselho: para evitar surpresas e dores de cabeça, nunca use o casamento parcial. Sempre faça um casamento completo, usando as âncoras `^` e `$` ao redor de sua expressão, e os curingas `.*` e `.+` onde necessário. Faça disso uma rotina para todas as diretivas, nunca escreva uma expressão que não seja ancorada.

<code>&lt;FilesMatch "\.(gif jpg png)\$"&gt;</code>	<code># parcial</code>
<code>&lt;FilesMatch "^.+\. (gif jpg png)\$"&gt;</code>	<code># completo</code>

A diretiva `<DirectoryMatch>` tem um funcionamento similar, porém casa nomes de pastas em vez de arquivos, e não pode ser usada no `.htaccess`. Veja um exemplo:

```
# [httpd.conf]
# Barrar o acesso às pastas "build-<número>"
#
<DirectoryMatch "^.* /build-\d+/?$">
    deny from all
```

`</DirectoryMatch>`

Esta diretiva tinha um problema que durou até a versão 2.3.8 do Apache: se você usasse a âncora \$ no final, a expressão nunca casava. Confira a sua versão antes de ancorar. Ah, e lembre-se de que, nas requisições, o nome da pasta pode vir com ou sem a barra final, então termine suas expressões com `/?$` para pegar os dois casos.

Outra pegadinha aqui é na ordem de execução das regras. As diretivas `<DirectoryMatch>` são colocadas no final da fila e só são interpretadas depois que todas as diretivas normais `<Directory>` forem aplicadas. Essa exceção é exclusiva para essa diretiva, nas outras acontece o comportamento usual: `<Files>` e `<FilesMatch>` são aplicadas na mesma ordem em que aparecem na configuração.

A diretiva `<LocationMatch>` completa o trio e vale tanto para pastas quanto para arquivos. Porém o casamento é feito no endereço (URL) da requisição, e não na localização do arquivo/pasta no servidor. Assim você pode casar redirecionamentos e endereços virtuais, como por exemplo as páginas de um site WordPress, que ficam no banco de dados e não no sistema de arquivos.

```
# [httpd.conf]
# Bloqueia as páginas de arquivo do blog, exemplo:
# http://aurelio.net/blog/page/5/
#
<LocationMatch "^/blog/page/\d+/?.*$">
    deny from all
</LocationMatch>
```

A URL a ser casada já chega sem o `http://domínio`, você só precisa se preocupar em casar o path. Veja que neste exemplo eu coloquei um curinga `.*` no final da expressão. Sem ele, a expressão casaria apenas a pasta `/blog/page/5/`, mas não os arquivos e as pastas dentro dela. Por isso mais uma vez o lembrete: sempre faça suas expressões bem completas e precisas, para não ter surpresas.

Há uma diferença de funcionamento que merece ser citada. Na diretiva normal `<Location>`, as barras consecutivas do endereço, que são inúteis, são



eliminadas antes do casamento. Por exemplo, o endereço `/foo///bar//` vira `/foo/bar/`. Já na diretiva `<LocationMatch>` isso não acontece. Se você precisa casar endereços com barras consecutivas, use o quantificador mais em todas as barras de sua expressão: `/+foo/+bar/+`.

A diretiva `AliasMatch` serve para mapear URLs para arquivos e pastas locais, mesmo que estejam fora do `DocumentRoot`. Você usa uma expressão regular para casar a URL (sem `http://domínio`) e indica a localização dela no disco local. O legal é que você pode usar retrovisores, reaproveitando partes da URL para compor o caminho local.

Por exemplo, temos no servidor uma pasta chamada `/pacotes` com centenas de arquivos `.ZIP` separados em pastas com a primeira letra de seu nome. Assim, o pacote `foobar-1.0.zip` fica dentro da pasta `/pacotes/f/`. Mas no site é necessário que todos os pacotes fiquem debaixo de uma única URL

`/download`. Esta regra faz este mapeamento:

```
# [httpd.conf]
# De: http://example.com/download/foobar-1.0.zip
# Para: /pacotes/f/foobar-1.0.zip
#
AliasMatch "^/download/(.)(.+) $" /pacotes/$1/$1$2
```

O primeiro grupo casou a primeira letra do nome do arquivo zip, e o segundo grupo pegou da segunda letra em diante. Depois bastou recompor o caminho usando os retrovisores `$1` e `$2`. Novamente reforçando: use aspas, use expressões completas (`^...$`).

A diretiva `RedirectMatch` serve para mapear uma URL para outra, útil para informar o endereço novo quando algum arquivo de seu site for renomeado ou movido para outra pasta (ou domínio). O visitante é direcionado automaticamente para o endereço novo, em vez de receber um decepcionante erro 404 (não encontrado). O Google também atualizará sua base de dados e passará a apontar para o endereço novo.

```
# [/.htaccess]
# Todas as fotos do site foram renomeadas de .JPEG para .jpg
```

```
#
RedirectMatch permanent "^(.+)\.JPEG$" $1.jpg
# Os ícones foram renomeados e movidos para uma nova pasta:
# /img/icone-twitter.png -> /img/icon/twitter.png
# /img/icone-facebook.png -> /img/icon/facebook.png
#
RedirectMatch permanent "^/img/icone-(.+)$" /img/icon/$1
# Seu blog WordPress foi convertido em um site estático
# De: http://example.com/2012/01/31/nome-do-post/
# Para: http://example.com/blog/nome-do-post.html
#
RedirectMatch permanent "^/\d{4}/\d\d/\d\d/([^\/]+)/$" /blog/$1.html
```

E assim vai, não tem segredo. Imagine que você está fazendo uma substituição (s///) na URL (sem o http://domínio). Você cria a expressão regular completa, com calma, coloca os grupos nos lugares certos e usa os retrovisores para compor o endereço novo, que pode ser local ou remoto. Caso a mudança de endereço seja temporária, troque a palavra permanent por temp.

Estes exemplos de RedirectMatch mudam a URL no navegador do visitante. Para manter a mesma URL original, mas mesmo assim poder fazer substituições “por baixo dos panos” no servidor, você deve usar o módulo mod\_rewrite.

```
# [httpd.conf]
# Usar URLs amigáveis em vez de query strings feiasas
# De: http://example.com/produto/1234/info
# Para: http://example.com/produto.php?id=1234&action=info
#
RewriteEngine on
RewriteRule "^/(\w+)/(\d+)/(\w+)$" /$1.php?id=$2&action=$3
```

A diretiva RewriteEngine ativa o módulo e a RewriteRule reescreve o endereço. A sintaxe de uso é idêntica à do RedirectMatch, a diferença aqui é que o visitante não é redirecionado, ele permanecerá com a URL bonita no navegador /produto/1234/info. A mudança do endereço ocorre internamente no servidor, que chamará a URL nova, executando o arquivo PHP com a

query string.

Se você usa arquivos `.htaccess`, atenção a uma diferença importante: a URL é sempre relativa à pasta onde o `.htaccess` se encontra. Para ilustrar, veja este exemplo que mostra como implementar a mesma técnica em quatro arquivos diferentes:

De: `http://example.com/doc/atual/intro.html`

Para: `http://example.com/doc/2.1/intro.html`

Arquivo	Regra
<code>httpd.conf</code>	<code>RewriteRule ^/doc/atual/(.*)\$ /doc/2.1/\$1</code>
<code>/.htaccess</code>	<code>RewriteRule ^doc/atual/(.*)\$ doc/2.1/\$1</code>
<code>/doc/.htaccess</code>	<code>RewriteRule ^atual/(.*)\$ 2.1/\$1</code>
<code>/doc/atual/.htaccess</code>	<code>RewriteRule ^(.*)\$ ../2.1/\$1</code>

O objetivo é que a URL `/doc/atual` sempre aponte para a versão mais recente da documentação, neste caso, `/doc/2.1`. Perceba que enquanto no `httpd.conf` o caminho é sempre absoluto, no `.htaccess` ele é relativo à pasta corrente, e sem a barra `/` no início.

Na dúvida, ative os logs do `mod_rewrite` lá no `httpd.conf`, para ver exatamente o que acontece com suas regras. Lembre-se de desativar depois que estiver tudo certo, para não sobrecarregar o servidor.

```
# Apache 2.2
RewriteLog "/tmp/mod_rewrite.txt"
RewriteLogLevel 3
# Apache 2.4
LogLevel alert rewrite:trace3
# Depois faça: tail -f error_log | fgrep '[rewrite:]'
```

Você pode usar uma ou mais diretivas `RewriteCond` para fazer testes em variáveis do Apache e condicionar a execução da próxima `RewriteRule` ao sucesso destes testes: a URL só será reescrita se todos os testes forem bem-sucedidos.

```
# [/.htaccess]
```

```
# Mostra uma foto educativa para quem faz hotlink
RewriteEngine on
RewriteCond %{HTTP_REFERER} "!"^$"
RewriteCond %{HTTP_REFERER} "!"^http://(www\.)?aurelio\.net/.*$"
RewriteRule "\.(jpg|png)$" http://aurelio.net/hotlink.jpg [R,L]
```

As duas condições deste exemplo testam o conteúdo de HTTP\_REFERER com expressões regulares negadas (note a ! no início), para detectar requisições vindas de sites externos: o conteúdo não é vazio e não é o meu próprio site. Neste caso, a RewriteRule é executada, redirecionando (flag [R]) as imagens para um endereço fixo com minha foto educativa.

E tem também a festa dos retrovisores. Se a sua expressão no RewriteRule tiver grupos, você pode usar os retrovisores \$1, \$2 nas condições. E o contrário também funciona: os grupos das condições também podem ser acessados via %1, %2 etc. Muito útil para obter dados adicionais para compor a URL nova:

```
# [/.htaccess]
# Obter dados da query string original
# De: http://example.com/index.php?prod_id=123
# Para: http://example.com/produto.php?id=123
#
RewriteEngine on
RewriteCond %{QUERY_STRING} "^.*prod_id=(\d+).*$"
RewriteRule "^index\.php$" produto.php?id=%1
```

A diretiva <If>, introduzida no Apache 2.4, também faz o teste de condições e pode ser usada como alternativa ao mod\_rewrite. Além de suportar vários outros tipos de testes, expressões regulares podem ser casadas com o operador =~ e negadas com o !~. A expressão deve vir no formato do Perl, entre barras /.../ ou com outro delimitador m#...#. Veja como fica o exemplo anterior do hotlink:

```
# [/.htaccess]
# Mostra uma foto educativa para quem faz hotlink
<If "%{HTTP_REFERER} !~ /^$/ && \
    %{HTTP_REFERER} !~ m#^http://(www\.)?aurelio\.net/.*$#">
    RedirectMatch temp "\.(jpg|png)$" http://aurelio.net/hotlink.jpg
```

</If>

E para acabar o assunto em grande estilo, você sabia que é possível “fazer um sed” no conteúdo das páginas? Eu também não sabia :)

```
# [/.htaccess]
# Usar s/// com PCRE dentro do Apache, quem diria!
AddOutputFilterByType SUBSTITUTE text/html
Substitute "s|<(.*?)>.*?</\1>|<$1>sed dominou!</$1>|"
```

Mais informações são encontradas em:

<http://aurelio.net/regex/apache/>

## Find

Característica	Como fazer
Busca	Opção -regex
Substituição	-
Divisão	-
ER crua	'entre aspas simples'
Ignore M/m	Opção -iregex
Global	-

O find é um comando clássico do Unix, usado para encontrar arquivos. Ferramenta de uso rotineiro para administradores de sistemas, também é muito útil para usuários de linha de comando no Unix, Linux e Mac, onde já vem instalado por padrão. No Windows, ele pode ser usado dentro do ambiente Cygwin.

O comando find pode encontrar arquivos seguindo diversas regras, como data de criação ou modificação, tamanho, dono, tipo, permissões, entre outros. Mas o que nos interessa aqui é a procura pelo nome do arquivo.

```
find . -name 'index.html'      # Encontre o arquivo index.html
find . -name '*.html'         # Encontre todos os .html
find . -not -name '*.html'     # Encontre todos exceto .html
```

Este é o uso básico do find na procura pelo nome do arquivo. A opção -name recebe o nome do arquivo procurado e a opção -not inverte as regras, excluindo do resultado os arquivos mencionados. Aquele asterisco é o curinga do shell, que não tem nada a ver com expressões regulares, cuidado para não confundir!

Mas em vez de usar a opção -name, que só aceita texto e alguns curingas, vamos usar a opção -regex; essa, sim, já espera receber uma expressão regular. Veja como ficou a execução aqui na minha pasta de testes:

```
prompt$ find .  
.  
./blog  
./blog/index.html  
./index.html  
./site.css  
prompt$ find . -name 'index.html'  
./blog/index.html  
./index.html  
prompt$ find . -regex 'index\.html'  
prompt$
```

Epa! Simplesmente trocar o nome da opção não é suficiente, o comando parou de funcionar. O que acontece é que enquanto a opção -name aceita casamentos parciais, a opção -regex deve sempre casar o caminho completo do arquivo.

```
prompt$ find . -regex '.*index\.html'  
./blog/index.html  
./index.html
```

Agora sim, o curinga .\* casou o caminho e o comando funcionou. Imagine que sempre há um ^ no início e um \$ no final da expressão, mesmo que você não os coloque. Aliás, se você colocar, vai funcionar do mesmo jeito, então é uma boa prática sempre usá-los para ficar claro que esse casamento nunca é parcial.

```
prompt$ find . -regex '^.*index\.html$'  
./blog/index.html  
./index.html
```

Outra dica importante é sobre as aspas. O find não faz nenhum pré-processamento na expressão regular, então todas podem ser consideradas cruas. Porém, como seu uso é feito na linha de comando e o shell faz seu próprio processamento antes de encaminhar o comando ao find, é preciso colocar as expressões entre aspas simples para protegê-las.

A opção `-not` funciona em conjunto com a `-regex`, então você também pode fazer a pesquisa inversa e listar todos os arquivos, exceto aqueles que casarem com a expressão.

```
prompt$ find . -not -regex '^.*index\.html$'
.
./blog
./site.css
prompt$ find . -not -regex '^.*index\.html$' -type f
./site.css
```

O primeiro comando listou as pastas “.” e “./blog” no resultado. No segundo comando, a opção `-type` força que somente sejam listados arquivos, excluindo assim as pastas e os links simbólicos. A posição das opções é importante, sempre coloque o `-not` imediatamente antes do `-regex` quando for negar uma expressão.

Para fazer sua expressão ignorar a diferença entre letras maiúsculas e minúsculas, basta adicionar a letra `i` ao nome da opção, ficando `-iregex`.

```
prompt$ find . -iregex '^.*INDEX\.HTML$'
./blog/index.html
./index.html
```

Até este ponto, tudo funciona igual tanto para o find do BSD/Mac quanto para o find do GNU/Linux. Pode usar sem medo. Mas agora precisamos falar sobre a diferença dos metacaracteres, então, daqui para frente, as coisas começam a ficar um pouco nebulosas.

No find do BSD, há dois tipos de expressões: básicas e estendidas, ou, melhor dizendo, antigas e modernas. O padrão é o tipo antigo, bem limitado, em que não existem os metacaracteres `?`, `+` e `|` e as chaves e grupos precisam ser escapados. Com a opção `-E` você pode usar a sintaxe moderna e tem

disponíveis todos os metacaracteres. Prefira sempre a moderna.

Comando	Metacaracteres (BSD/Mac)									
find	.	^	\$	*			[ ]	\{ \}	\( \)	\1
find -E	.	^	\$	*	+	?		[ ]	{ }	( ) \1

No GNU find é mais complicado, pois há cinco tipos diferentes de expressões: emacs, posix-awk, posix-basic, posix-egrep e posix-extended. O padrão é o tipo emacs, mas você pode escolher qualquer um dos outros tipos usando a opção -regextype. Eu tive a paciência de testar todos os tipos e montar esta tabelinha para você consultar.

-regextype	Metacaracteres (GNU/Linux)									
emacs	.	^	\$	*	+	?		[ ]		\( \)
posix-basic	.	^	\$	*	\+	\?	\	[ ]	\{ \}	\( \)
posix-extended	.	^	\$	*	+	?		[ ]	{ }	( ) \1
posix-awk	.	^	\$	*	+	?		[ ]	{ }	( ) \1
posix-egrep	.	^	\$	*	+	?		[ ]	{ }	( ) \1

Apesar de os três últimos tipos usarem os mesmos metacaracteres, há pequenas diferenças em seu funcionamento, que são muito específicas e nem vale a pena mencionar. Na prática, pode considerá-los iguais. Infelizmente, o tipo padrão emacs é o mais bizarro de todos: precisa escapar o grupo, mas não os outros, e não tem chaves. Evite-o a todo custo.

Para preservar sua sanidade, acostume-se a sempre usar a opção -E no BSD e a opção -regextype posix-egrep no GNU. Assim você usará os mesmos metacaracteres em ambos, numa sintaxe moderna e completa. Aqui vão os exemplos para reforçar:

u GNU/Linux

```
prompt$ find . -regextype posix-egrep -regex '^(.*(css|html))$'
./blog/index.html
./index.html
```



```
./site.css
```

## u BSD/Mac

```
prompt$ find -E . -regex '^(.*(css|html))$'
./blog/index.html
./index.html
./site.css
```

Preste atenção a dois detalhes. No BSD, a opção `-E` deve ser o primeiro argumento do comando, antes mesmo da pasta de início. Se você inverter a ordem, vai dar erro:

```
prompt$ find . -E -regex '^(.*(css|html))$'
find: -E: unknown option
```

No GNU, a opção `-regextype` deve ser a primeira logo após a pasta de início. Se você colocá-la depois de outra opção, vai dar erro:

```
prompt$ find . -regex '^(.*(css|html))$' -regextype posix-egrep
find: warning: you have specified the -regextype option after a non-
```

Para finalizar, uma nota sobre compatibilidade: a opção `-regextype` foi adicionada ao GNU find na versão 4.2.24, em julho de 2005.

Mais informações são encontradas em:

<http://aurelio.net/regex/find/>

## Grep

Característica	Como fazer
Busca	Basta informar a ER
Substituição	-
Divisão	-
ER crua	'entre aspas simples'
Ignore M/m	Opção <code>-i</code>
Global	-

O grep é um comando do Unix que serve para pesquisar textos em arquivos,

usando expressões regulares. Trata-se de um comando clássico, surgido na década de 1970 (é mais velho que você!), e ainda hoje é um dos comandos mais utilizados por administradores de sistemas e usuários de linha de comando do Unix, Linux e Mac. Também pode ser instalado no Windows, isoladamente ou dentro do ambiente Cygwin.

Seu funcionamento é bem simples: lê o conteúdo do arquivo informado linha a linha e, para cada uma delas, tenta casar a expressão regular. Se obteve sucesso, mostra a linha toda na saída. Note que é “a linha toda” e não “o trecho casado”, essa diferença é bem importante. Veja um exemplo:

```
prompt$ grep '^r' /etc/passwd
root:x:0:0:root:/root:/bin/bash
rpm:x:37:37::/var/lib/rpm:/sbin/nologin
rpc:x:32:32:Portmapper RPC user:/:/sbin/nologin
```

Procuramos por linhas que começam com a letra r no arquivo /etc/passwd. O trecho casado foi somente a primeira letra da linha, mas o grep sempre retorna a linha toda.

Por sua rapidez e facilidade de uso, o grep é uma ferramenta excelente para testar rapidamente uma expressão regular. Você pode utilizá-lo para “grehpar” linhas de um ou mais arquivos, ou da entrada padrão (STDIN).

```
grep '^r' /etc/passwd          # um arquivo
grep '^r' *.txt                # arquivos TXT
cat /etc/passwd | grep '^r'    # STDIN
```

O grep não faz nenhum pré-processamento na expressão regular, então todas podem ser consideradas cruas. Porém, como seu uso é feito na linha de comando e o shell faz seu próprio processamento antes de encaminhar o comando ao grep, é preciso colocar as expressões entre aspas simples para protegê-las.

```
grep ^r /etc/passwd          # errado
grep '^r' /etc/passwd        # certo
```

Para ignorar a diferença entre letras maiúsculas e minúsculas, basta utilizar a opção -i. Para inverter a expressão, ignorando todas as linhas que casam com ela, use a opção -v. Outras opções úteis são a -w, para forçar o

casamento de uma palavra inteira (e não parcial), e a -x, para casar uma linha inteira.

```
prompt$ cat numeros.txt
um
dois
três
prompt$ cat numeros.txt | grep DOIS
prompt$ cat numeros.txt | grep -i DOIS
dois
prompt$ cat numeros.txt | grep -iv DOIS
um
três
```

O grep possui três identidades: grep, egrep e fgrep. São todos o mesmo programa, a diferença está na maneira que vão interpretar o padrão de pesquisa informado.

- O grep procura uma expressão regular básica usando aquela sintaxe antiga em que vários caracteres precisam ser escapados para ser considerados metacaracteres.
- O egrep é o “extended” que usa a sintaxe moderna das expressões regulares. Com ele, não são necessários escapes nos metacaracteres.
- O fgrep é o “fast” ou “fixed string” que não sabe nada de expressões regulares e somente procura textos normais.

Comando	Metacaracteres
grep	<code>^ \$ . * [ \? \+ \  \ ( \ ) \{ \}</code>
egrep	<code>^ \$ . * [ ? +   ( ) { }</code>
fgrep	nenhum

Sempre que possível, use o egrep para não ter de ficar escapando os metacaracteres. Quando precisar pesquisar textos normais, use o fgrep.

O GNU grep possui a opção -P, que aumenta ainda mais o poder da ferramenta, permitindo que se utilizem as expressões regulares avançadas do

Perl.

O GNU grep também possui a opção -o, que mostra apenas o trecho de texto casado pela expressão, em vez do comportamento padrão de sempre mostrar a linha inteira. Veja a diferença:

```
prompt$ grep 'root.*0' /etc/passwd
root:x:0:0:root:/root:/bin/bash
prompt$ grep -o 'root.*0' /etc/passwd
root:x:0:0
```

Mais informações são encontradas em:

<http://aurelio.net/regex/grep/>

## Nginx

O nginx é um servidor HTTP lançado em 2004. Famoso por ser muito rápido e eficiente no uso de memória, ele está entre os três servidores web mais utilizados no mundo, juntamente com o Apache e o Microsoft IIS. Sua configuração é feita usando diretivas, e algumas delas aceitam expressões regulares:

Independentemente do formato usado pela diretiva, a sintaxe da expressão regular em si é sempre a mesma. Você pode abusar dos metacaracteres modernos e avançados do Perl, pois o Nginx usa a biblioteca PCRE (Perl Compatible Regular Expressions).

Módulo	Diretiva	Formato	Tipo do match
Core	location	location ~ ER { ... }	normal
		location ~* ER { ... }	ignorecase
	server_name	server_name ~ ER;	ignorecase
FastCGI	fastcgi_split_path_info	fastcgi_split_path_info ER;	normal
Gzip	gzip_disable	gzip_disable "ER";	ignorecase
Map	map	map ... { ...; ~ER 1; }	normal
		map ... { ...; ~*ER 1; }	ignorecase

Proxy	proxy_cookie_domain	proxy_cookie_domain ~ER texto;	ignorecase
	proxy_cookie_path	proxy_cookie_path ~ER texto;	normal
		proxy_cookie_path ~*ER texto;	ignorecase
	proxy_redirect	proxy_redirect ~ER texto;	normal
		proxy_redirect ~*ER texto;	ignorecase
Referer	valid_referers	valid_referers ~ER;	ignorecase
Rewrite	if	if (\$var ~ ER) { ... }	normal
		if (\$var ~* ER) { ... }	ignorecase
		if (\$var !~ ER) { ... }	negado
		if (\$var !~* ER) { ... }	negado e ignorecase
	rewrite	rewrite ER texto;	normal
SSI	if expr	<!--# if expr="\$var = /ER/" -->	normal
		<!--# if expr="\$var != /ER/" -->	negado
Upstream match		match nome { campo ~ ER; }	normal
		match nome { campo !~ ER; }	negado

Uma das diretivas mais utilizadas na configuração do Nginx é a `location`, que serve para definir um conjunto de regras para endereços (URLs) específicos do seu site. Estes endereços, é claro, podem ser informados usando expressões regulares. Que tal começarmos a brincadeira quebrando todas as imagens do site?

```
location ~ "\.(gif|png|jpg)$" {
    deny all;
}
```

O operador `~` é usado para indicar que o próximo argumento será uma expressão regular, e não um texto normal. Com esta regra, será negado o acesso a todos os arquivos do site que tiverem a extensão gif, png ou jpg.

Para pegar também as imagens com as extensões GIF, PNG e JPG, em

maiúsculas, será preciso fazer um casamento ignorando a diferença entre letras minúsculas e maiúsculas. O operador `~*` faz este papel no Nginx.

```
location ~* "\.(gif|png|jpg)$" {  
    deny all;  
}
```

Outra maneira de fazer isso é utilizar o operador normal `~` e adicionar o modificador `(?i)` no início da expressão, pois ele é suportado pela biblioteca PCRE. É uma questão de gosto pessoal, uns preferem informar o modificador junto com a expressão, outros preferem usar o operador externo.

```
location ~ "(?i)\.(gif|png|jpg)$" {  
    deny all;  
}
```

Uma exceção ocorre em sistemas operacionais “case-insensitive”, como Windows, Mac OS X e Cygwin. Nestes sistemas, tanto faz maiúsculas ou minúsculas; você pode se referir a um arquivo como FOTO.JPG ou foto.jpg. Quando está em um sistema assim, o Nginx replica este comportamento nas expressões regulares; é como se o modificador `(?i)` estivesse sempre presente. Para forçar um casamento normal, use o modificador `(?-i)` no início da expressão.

```
# Casamento "case-sensitive" em sistemas "case-insensitive"  
location ~ "(?-i)\.(gif|png|jpg)$" {  
    deny all;  
}
```

É importante entender que, em todas as diretivas com suporte às expressões regulares, o casamento parcial é sempre válido. Note que nos exemplos a expressão casou apenas a extensão do arquivo, não se importando com o seu nome e caminho completo. Isso é um casamento parcial.

Se você quiser um casamento completo, que valide todo o endereço do início ao fim, use as âncoras `^` e `$` em sua expressão. Lembre-se, porém, de que a URL a ser casada já chega sem o prefixo `http://domínio`, então você só precisa se preocupar em casar o path.

```
# URL: http://aurelio.net/imagens/foo.png
```

```
# Path: /imagens/foo.png
location ~ "^/imagens/.\+\. (gif|png|jpg)$" {
    deny all;
}
```

Percebeu que em todos os exemplos a expressão regular foi colocada entre aspas? Faça isso sempre para evitar problemas com caracteres especiais para o Nginx, como o ; ou o {. Isso é bem importante, pois, se você cometer qualquer erro nestas diretivas, o servidor não iniciará e acusará erro de sintaxe na configuração. Aspas. Sempre.

```
location ~ ^/blog/\d{4}/      # Dá erro
location ~ "^/blog/\d{4}/"    # Ok
```

A diretiva `if` é útil para testar o conteúdo de variáveis. Se o teste for bem-sucedido, as regras que estão dentro do bloco serão executadas. As expressões regulares podem ser casadas com o operador `~` e negadas com o `!~`. Estes operadores também apresentam versões alternativas, que ignoram a diferença entre maiúsculas e minúsculas; basta adicionar um asterisco no final: `~*` e `!~*`.

```
# Bloqueia o site para o IE6 e anteriores :P
if ($http_user_agent ~ "MSIE [4-6]\.") {
    deny all;
}
# Rejeita requisições que não sejam GET ou POST
if ($request_method !~ "GET|POST") {
    return 405;
}
```

A diretiva `rewrite` é usada para editar a URL original, redirecionando a requisição para uma pasta ou arquivo diferente ou para uma nova URL. Muito útil quando você faz mudanças estruturais no site, para garantir que as URLs antigas não quebrem e sejam encaminhadas para a nova localização. O uso é bem simples; basta informar a expressão e o texto substituto.

```
# Todas as fotos do site foram renomeadas de .JPEG para .jpg
#
rewrite "^(.+)\.JPEG$" $1.jpg;
# Os ícones foram renomeados e movidos para uma nova pasta:
```

```
# /img/icone-twitter.png -> /img/icon/twitter.png
# /img/icone-facebook.png -> /img/icon/facebook.png
#
rewrite "^/img/icone-(.+) $" /img/icon/$1;
# Seu blog WordPress foi convertido em um site estático
# De: http://example.com/2015/01/31/nome-do-post/
# Para: http://example.com/blog/nome-do-post.html
#
rewrite "^/\d{4}/\d\d/\d\d/([^\s]+)/ $" /blog/$1.html permanent;
# Usar URLs amigáveis em vez de query strings feiasas
# De: http://example.com/produto/1234/info
# Para: http://example.com/produto.php?id=1234&action=info
#
rewrite "^(/(\w+)/(\d+)/(\w+)) $" /$1.php?id=$2&action=$3;
# O site mudou para um novo domínio
#
rewrite "^(/.*) $" http://site-novo.com$1 permanent;
```

E assim vai, não tem segredo. Imagine que você está fazendo uma substituição (s///) na URL original (sem o http://domínio). Você cria a expressão regular completa, com calma, coloca os grupos nos lugares certos e usa os retrovisores para compor o endereço novo, que pode ser local ou remoto.

Se no final da linha você colocar as flags `redirect` ou `permanent`, o redirecionamento será feito de forma explícita, ou seja, será criada uma nova requisição para o novo endereço, alterando a URL no navegador do usuário. Isso é útil quando você quer que o usuário perceba a mudança. Sem estas flags, o redirecionamento é apenas interno ao Nginx e não altera a URL do navegador.

```
rewrite "^(.+)\.JPEG$" $1.jpg permanent; # Muda URL no navegador
rewrite "^(.+)\.JPEG$" $1.jpg;           # Não muda
```

Continuam valendo as dicas de colocar a expressão entre aspas para evitar erros de sintaxe e usar o modificador `(?i)` no início da expressão caso precise fazer um casamento ignorando a diferença entre letras maiúsculas e minúsculas.



Não faça casamentos parciais no rewrite, sempre use expressões regulares completas, com as âncoras ^ e \$. Isso porque a nova URL sempre deve ser completa, seja ela um path local ou um novo domínio. Por exemplo, ao fazer a regra para trocar a extensão dos arquivos JPEG para jpg, deve-se levar em conta o path e o nome do arquivo, e não somente sua extensão.

```
rewrite    "\.JPEG$"    .jpg;      # Parcial, não funciona
rewrite "^(.+)\.JPEG$" $1.jpg;    # Completo, funciona
```

Sempre que você usa grupos nas expressões regulares, o conteúdo casado por eles é guardado em variáveis especiais, como \$1, \$2, \$3... Até aí tudo bem, acabamos de ver vários exemplos disso no rewrite. O legal é que você também pode usar estas variáveis nas outras diretivas que estão no mesmo contexto da expressão.

```
# Mapeia a URL /download para a pasta /dados/documentos/
location ~ "^/download/(.*)$" {
    autoindex on;
    alias /dados/documentos/$1;
}
# Cada domínio está numa subpasta em /sites
server {
    server_name ~ "^(www\..)?(.+)";
    location / {
        root /sites/$2;
    }
}
```

Você também pode dar nomes aos grupos da expressão regular, utilizando a sintaxe (?<nome>...). Neste caso, o nome do grupo será usado para criar a sua variável especial correspondente. Assim, fica fácil saber qual parte da expressão está sendo usada nas outras diretivas que referenciam os grupos. Veja os exemplos anteriores, agora com grupos nomeados:

```
# Mapeia a URL /download para a pasta /dados/documentos/
location ~ "^/download/(?<arquivo>.*)" {
    autoindex on;
    alias /dados/documentos/$arquivo;
}
```

```
# Cada domínio está numa subpasta em /sites
server {
    server_name ~ "^(www\.)?(?<dominio>.+)$";
    location / {
        root /sites/$dominio;
    }
}
```

Mais informações são encontradas em:

<http://aurelio.net/regex/nginx/>



## Capítulo 8

# Linguagens de programação

E como não podemos viver só editando trabalhos escolares e relatórios técnicos, nada como uma codificada em um final de semana chuvoso. E, para nossa alegria, lá estão as ERs de novo.

Várias linguagens de programação possuem suporte às expressões regulares, seja nativo, como módulo importável, como biblioteca carregável, como objeto instanciável, como... Ou seja, opções há várias.

Como cada linguagem tem sua maneira específica de receber e tratar ERs, vamos dar uma geral nas mais usadas, com dicas de uso e pegadinhas que podem assustar quem está começando.

Para facilitar a consulta posterior, cada linguagem tem logo no começo sua tabelinha, em que estão resumidas todas as informações que geralmente nos importam para lidar com ERs. Estas informações são:

Característica	Descrição
Busca	Como buscar, casar um texto
Substituição	Como substituir um texto por outro
Divisão	Como dividir um texto em partes
ER crua	Como especificar uma ER crua

Ignore M/m	Como ignorar diferença maiúsculas/minúsculas
Global	Como fazer uma substituição global

## Awk

Característica	Como fazer
Busca	Operador ~, Função match
Substituição	Funções sub, gensub
Divisão	Função split
ER crua	/entre barras/
Ignore M/m	Variável IGNORECASE
Global	Função gsub, modificador g

Awk é uma linguagem antiga (1977) que combina processamento de texto com estruturas de uma linguagem genérica, possuindo condicionais, operações aritméticas e afins.

Além do Awk clássico do Unix, também existe o GNU Awk, conhecido como gawk, que traz algumas funcionalidades novas e um melhor suporte às expressões regulares. O gawk é amplamente utilizado em sistemas Linux.

Grande parte dos exemplos deste tópico funcionará em ambas as versões. Caso contrário, será feita uma menção sobre quais são as características exclusivas do GNU Awk.

As expressões regulares são parte integrante da linguagem. Basta colocar uma expressão entre barras que o Awk saberá que aquilo não é uma string, e sim uma expressão regular com metacaracteres. Se sua expressão possuir uma barra /, lembre-se de escapá-la \ para evitar problemas.

O operador til ~ é usado para fazer comparações de strings com expressões regulares. Para uma comparação inversa, em que o teste retornará sucesso se a expressão não casar com a string, use o operador !~.

```
if ("Awk" ~ /^A/) {
```

```

    print "Casou"
}
if ("Awk" !~ /^X/) {
    print "Não casou"
}

```

Para fazer testes ignorando a diferença entre maiúsculas e minúsculas, antes mude para 1 o valor da variável IGNORECASE, que funciona como uma chave que liga e desliga esse comportamento. Mude o valor da variável para zero quando quiser retornar à comparação normal.

```

IGNORECASE = 1
if ("Awk" ~ /^a/) {
    print "Casou"
}

```

O Awk do Unix não reconhece essa variável, então a única maneira de simular esse comportamento é usando a criatividade: converta a string para minúsculas ao fazer o teste e use somente minúsculas em sua expressão.

```

if (tolower("Awk") ~ /^a/) {
    print "Casou"
}

```

Outra maneira de fazer testes é utilizar uma expressão regular para casar linhas específicas de um arquivo. Quando o Awk está processando um arquivo de texto, ele o lê linha a linha. A linha da vez é guardada na variável especial \$0.

Ao colocar uma expressão regular diretamente como um teste, ela será testada na linha atual (como se fosse \$0 ~ /er/) e os comandos do bloco seguinte só serão executados nas linhas que casarem com o padrão.

```

/[0-9]/ {
    print "Linha com números:", $0
}

```

Dessa maneira, fica fácil testar expressões regulares na linha de comando, seja com o conteúdo de um arquivo, seja com um texto vindo da entrada padrão (STDIN).

```

prompt$ echo Awk | awk '/^A/ { print "Casou" }'

```

Casou

Outra maneira de casar um texto é utilizar a função `match`. A vantagem é que, a cada vez que é usada, a função define as variáveis `RSTART` e `RLENGTH`, que guardam o ponto de início (índice) e o tamanho do trecho casado pela expressão.

No GNU Awk, é possível informar um array como terceiro parâmetro para a função `match`. Nesse caso, a primeira posição (índice zero) desse array será preenchida com o trecho casado pela expressão, e as posições seguintes guardarão o conteúdo de cada grupo.

```
match("Awk", /(.) (.)(.)/, resultado)
print resultado[0]    # Awk
print resultado[1]    # A
print resultado[2]    # w
print resultado[3]    # k
print RSTART          # 1
print RLENGTH         # 3
```

A substituição é feita tradicionalmente pela função `sub`, que troca apenas a primeira ocorrência do padrão. Sua irmã `gsub` encarrega-se de fazer a substituição de todas as ocorrências (global). O detalhe é que o texto alterado é gravado na própria variável que continha o texto original, em vez de ser retornado pela função. Com isso, não é possível usar strings diretamente, sendo sempre necessário o uso de uma variável.

```
texto = "Awk";
sub(/[A-Za-z]/, ".", texto)
print texto          # .wk
texto = "Awk";
gsub(/[A-Za-z]/, ".", texto)
print texto          # ...
```

Se a variável com o texto não for informada à função, é utilizada a variável especial `$0`, que contém a linha atual do arquivo processado (ou entrada padrão). Assim o uso dessas funções torna-se mais ágil.

```
prompt$ echo Awk | awk 'sub(/[A-Za-z]/, ".")'
.wk
```

```
prompt$ echo Awk | awk 'gsub(/[A-Za-z]/, ".")'
...
prompt$ echo Awk | awk 'sub(/.*/, "&&")'
AwkAwkAwk
```

Note que ao usar o caractere & no segundo argumento da função, ele é expandido para o trecho casado pela expressão. Mas este é o único caractere considerado especial no texto substituto, pois as funções sub e gsub não têm suporte aos retrovisores.

O GNU Awk criou uma função nova chamada gensub, que, além do suporte aos retrovisores, também traz outras novidades que visam a contornar as limitações das outras funções. Uma de suas vantagens é que o texto alterado é retornado pela função em vez de ser gravado em uma variável. Há também um terceiro argumento numérico que indica qual das ocorrências deve ser substituída. Se esse argumento for a letra g, todas as ocorrências serão substituídas.

```
print gensub(/\w/, ".", "1", "Awk")    # .wk
print gensub(/\w/, ".", "2", "Awk")    # A.k
print gensub(/\w/, ".", "3", "Awk")    # Aw.
print gensub(/\w/, ".", "g", "Awk")    # ...
```

Os retrovisores são indicados por \1, \2 e amigos. Mas como são parte de uma string (entre aspas), devem ser escapados para funcionar: \\1, \\2, ... O retrovisor \0 guarda todo o trecho casado pela expressão.

```
print gensub(/(.)(.)(.)/, "\\3\\2\\1", "g", "Awk")    # kwA
print gensub(/.*/ , "--\\0--" , "g", "Awk")    # --Awk--
```

A mesma dica de uso da variável IGNORECASE também vale para a gensub ignorar a diferença entre maiúsculas e minúsculas. Ligue e desligue essa variável com 1 e 0 conforme precisar dela.

```
IGNORECASE = 0
print gensub(/[a-z]/, ".", "g", "Awk")    # A..
IGNORECASE = 1
print gensub(/[a-z]/, ".", "g", "Awk")    # ...
```

A acentuação não é problema, desde que seu sistema esteja configurado

corretamente para o português. Confira o valor das variáveis de ambiente \$LANG e \$LC\_ALL. Use as classes POSIX para casar os caracteres acentuados. No gawk, você também pode usar o barra-letra \w.

```
print gensub(/[a-z]/, ".", "g", "adábuká") # ..á...á
print gensub(/[:,alpha:]/, ".", "g", "adábuká") # .....
print gensub(/\\w/, ".", "g", "adábuká") # .....
```

Use a função split para fazer a divisão de uma string. Seu segundo argumento é o array onde a string dividida será guardada e o terceiro é a expressão regular. Note que o índice inicial é 1 e não 0.

```
split("A w k", resultado, /[ \t]+/)
print resultado[1] # A
print resultado[2] # w
print resultado[3] # k
```

Usuários avançados de Awk gostarão de saber que as variáveis especiais RS (separador de registros) e FS (separador de campos) também podem ser definidas com uma expressão regular. Apenas se lembre de que, como sua definição é feita por meio de uma string, é preciso escapar as contrabarras \.

```
prompt$ echo "A -- w -- k" | awk -F '[- ]+' '{ print $2 }'
w
```

Por fim, se você estiver utilizando o GNU Awk e quiser testar a compatibilidade de seus scripts com o Awk original do Unix ou outras versões, veja as opções --posix, --traditional e --re-interval.

Mais informações são encontradas em:

<http://aurelio.net/regex/awk/>

## C

Característica	Como fazer
Busca	Função regexec
Substituição	-
Divisão	-
ER crua	-



Ignore M/m	Flag REG_ICASE
Global	-

Das interfaces de expressões regulares disponíveis para a linguagem C, a compatível com o padrão POSIX é a mais padrão por seguir o padrão. Sacou?

A rotina é a seguinte: primeiro você tenta compilar a ER (regcomp). Caso algo esteja errado na expressão, mostre a mensagem de erro para o usuário (regerror). Compilou? Então tente casar a ER com o texto (regexexec). No final de tudo, não se esqueça de limpar a bagunça (regfree). Um exemplo bem simples, sem checagem de erro nem faxina:

```
/* casa.c */
#include <stdio.h>
#include <stdlib.h>
#include <regex.h>
int main(int argc, char **argv) {
    /* Aloca e compila */
    regex_t er;
    regcomp(&er, argv[1], REG_EXTENDED|REG_NOSUB);
    /* Casou ou nao? */
    if ((regexexec(&er, argv[2], 0, NULL, 0)) == 0)
        printf("Casou\n");
    else
        printf("Não casou\n");
    exit(0);
}
```

A estrutura regex\_t é o buffer usado para guardar o padrão de pesquisa. A flag REG\_EXTENDED indica que a ER está no formato moderno, o mesmo que você aprendeu aqui. A flag REG\_NOSUB indica que apenas um teste será feito (casou ou não?), sem coletar informações adicionais. Executando este exemplo:

```
prompt$ ./casa ana banana
Casou
prompt$ ./casa maria banana
Não casou
```

Essa foi fácil. Já para saber onde casou e o que casou, fica mais trabalhoso. Vamos lá. Agora queremos obter as informações sobre o casamento, então não usamos mais o REG\_NOSUB na compilação e passamos a estrutura regmatch\_t para a regexec.

```
/* casa2.c */
#include <stdio.h>
#include <stdlib.h>
#include <regex.h>
int main(int argc, char **argv)
{
    int start, error;
    /* Aloca e compila */
    regex_t er;
    regmatch_t match;
    regcomp(&er, argv[1], REG_EXTENDED);
    start = 0;
    /* Tenta casar a ER no início da linha.
     * Se achou, entra no loop, mostra os dados e tenta novamente
     * desta vez indicando que não está mais no início da linha
     * REG_NOTBOL - Not Beginning Of Line
     */
    error = regexec(&er, argv[2], 1, &match, 0);
    while (error == 0) {
        printf("Texto de pesquisa: %s\n", argv[2]+start);
        printf("Casou de %d a %d\n", match.rm_so, match.rm_eo);
        start += match.rm_eo; /* Move o "cursor" adiante */
        error = regexec(&er, argv[2]+start, 1, &match, REG_NOTBOL);
    }
    exit(0);
}
```

Na leitura deste código, perceba que, para casar mais de uma vez na mesma linha (global), é preciso fazer “na mão” um loop que apaga o trecho já casado e tenta casar no que restou. Em rm\_so e rm\_eo ficam os índices de início de fim do trecho casado. Acompanhe:

```
prompt$ ./casa2 [0-9]+ 11aa222bbb3333cccc
Texto de pesquisa: 11aa222bbb3333cccc
```

Casou de 0 a 2

Texto de pesquisa: aa222bbb3333cccc

Casou de 2 a 5

Texto de pesquisa: bbb3333cccc

Casou de 3 a 7

Mais informações são encontradas em:

<http://aurelio.net/regex/c/>

## HTML5

Característica	Como fazer
Busca	Atributo pattern na tag input
Substituição	-
Divisão	-
ER crua	É o padrão
Ignore M/m	-
Global	-

O HTML5 é a versão atual da linguagem HTML, que veio para substituir o HTML4.01 e o XHTML 1.0. Ele corrige falhas das versões anteriores e traz novidades empolgantes que facilitam o desenvolvimento de aplicativos Web. Dessas novidades, a que realmente nos interessa é a possibilidade de usar expressões regulares para validar formulários diretamente no HTML, sem JavaScript.

No seu site ou aplicativo Web, se você pede para o usuário digitar o CPF, é preciso verificar se o número foi digitado corretamente. Como o formato é padronizado (nnn.nnn.nnn-nn), basta escrever uma expressão regular e tentar casá-la com o texto digitado. Se não casar, está errado. Atualmente, a maneira mais prática de se fazer isso é com JavaScript. Por exemplo:

```
<script>
function valida_cadastro() {
    var er_cpf = /^\\d{3}\\.\\d{3}\\.\\d{3}-\\d{2}$;/;
```

```

        var campo_cpf = document.cadastro.cpf.value;
        if (!campo_cpf.match(er_cpf)) {
            alert("Erro: O CPF é inválido");
            return false;
        }
        return true;
    }
</script>
...
<form name="cadastro" onsubmit="return valida_cadastro();">
    CPF:
    <input type="text" name="cpf">
    <input type="submit">
</form>

```

Funciona, mas é muito trabalhoso para fazer uma simples validação. Quanto mais campos o formulário tiver (Nome, Idade, E-mail, Data de nascimento), maior será seu script para poder testar todos. Outro problema é que a maneira de informar ao usuário sobre o erro (um `alert()` no exemplo) não é padronizada, cada site faz de um jeito. Porém, apesar de todos esses esforços, se o usuário desligou o JavaScript, então nada funcionará.

O HTML5 criou uma nova maneira de fazer essa validação, que é colocar a expressão regular diretamente no campo em questão, usando o novo atributo `pattern`. Veja como fica o exemplo anterior:

```

<form name="cadastro">
    CPF:
    <input type="text" name="cpf" pattern="\d{3}\.\d{3}\.\d{3}-\d{2}"
    <input type="submit">
</form>

```

E pronto! Sem JavaScript, sem evento `onsubmit`, sem DOM, sem `alert()`, sem complicação. É o navegador que verificará se o texto digitado casa com a expressão e, se necessário, avisará o usuário que há algo errado, de maneira padronizada. Use o atributo `title` para definir a mensagem que será mostrada ao usuário em caso de erro.

```

<input type="text" name="cpf"

```

```
pattern="\d{3}\.\d{3}\.\d{3}-\d{2}"  
title="Digite o CPF no formato nnn.nnn.nnn-nn">
```

Um leitor comum poderia parar por aqui e estaria satisfeito. Mas nós precisamos saber mais detalhes sobre o comportamento das expressões regulares, certo? Então vamos.

- **Metacaracteres:** Os metacaracteres usados pelas expressões no HTML5, como é de se esperar, são exatamente os mesmos do JavaScript. Não há segredo, basta escrever sua expressão normalmente, como você faria no JavaScript, porém sem as barras ao redor.
- **Casamento parcial:** Um detalhe importante é que sempre se tenta o casamento na string completa, não valendo casamentos parciais. Por exemplo, a expressão `\d+` vai casar “123”, mas não “CPM 22”. Na prática, considera-se que a expressão sempre possui âncoras no início e no fim: `^\d+$`, você as colocando ou não. Para permitir casamentos parciais, use o curinga `.*` ao redor: `.*\d+.*`.
- **Maiúsculas e minúsculas:** Não há como você usar modificadores em sua expressão, como `/[a-z]/i`, para ignorar a diferença entre maiúsculas e minúsculas. A expressão é sempre considerada normal, com todos os modificadores desligados: `global`, `multiline` e `ignoreCase`. Então, na prática, se precisar de maiúsculas e minúsculas, use `\w` ou `[A-Za-z]`.

O DOM também foi melhorado, para, além da simples verificação, você também poder interagir com os campos. E se você quiser fazer algo especial quando o usuário digitar um texto incorreto?

Há uma propriedade nova para o elemento `input` chamada `validity.patternMismatch`. É uma chave booleana que, quando ligada, indica que a expressão regular do atributo `pattern` não casou. Veja como acessar essa chave no JavaScript, em nosso exemplo do CPF:

```
if (document.cadastro.cpf.validity.patternMismatch) {  
    alert("O CPF está incorreto");  
} else {  
    alert("O CPF está correto");  
}
```

Há um método novo para o elemento form chamado `checkValidity()`. Com ele, você pode forçar a verificação dos campos do formulário a qualquer momento. Se algum campo estiver com o conteúdo incorreto, onde a expressão regular de seu atributo `pattern` não casar, será disparado um evento `invalid`. Você pode então usar o atributo `oninvalid` desse campo para definir uma função personalizada para cuidar do erro. Exemplo:

```
<input type="text" name="cpf"
      pattern="\d{3}\.\d{3}\.\d{3}-\d{2}"
      title="Digite o CPF no formato nnn.nnn.nnn-nn"
      oninvalid="return cpf_incorreto();">
```

Há um seletor novo no CSS chamado `:invalid`, para você mudar a aparência de campos que estão com o valor incorreto. Em nosso caso, será muito útil para destacar os campos em que a expressão regular não casou.

```
<style>
  /* Campos normais são brancos, campos com erro são vermelhos */
  input[type="text"] { background-color: white; color: black; }
  input:invalid      { background-color: #fcc ; color: red; }
</style>
```

Você pode usar todas essas funcionalidades sem problemas no Google Chrome, Firefox e Opera. O Safari está quase completo, só falta travar o envio do formulário quando a expressão não casa. Já no Internet Explorer 9, nada disso funciona, infelizmente.

Em navegadores mais antigos ou limitados, que não suportam o atributo `pattern` nativamente, você pode usar o `webforms2`, um arquivo JavaScript que implementa as novidades do HTML5 de maneira independente. Seu endereço é <https://github.com/westonruter/webforms2>.

Montei uma página especial para que você possa testar o funcionamento das expressões regulares no HTML5, o endereço está no parágrafo seguinte. Acesse e confira como o seu navegador atual se comporta quando algo errado é digitado nos campos.

Mais informações são encontradas em:

<http://aurelio.net/regex/html5/>

# Java

Característica	Como fazer
Busca	Método matches
Substituição	Método replaceFirst
Divisão	Método split
ER crua	-
Ignore M/m	(?i), Pattern.CASE_INSENSITIVE
Global	Método replaceAll

O suporte oficial a ERs no Java apareceu somente na versão 1.4 do J2SE, que trouxe o pacote `java.util.regex` e suas classes `Pattern` e `Matcher`. Além disso, a classe `java.lang.String` também foi melhorada, suportando ERs implicitamente nos métodos `matches`, `replaceFirst`, `replaceAll` e `split`.

```
// usandoString.java
class usandoString {
    public static void main(String argv[]) {
        System.out.println("acalento".matches(".*lento"));
        System.out.println("acalento".replaceAll(".*a(.*)", "$1"));
        System.out.println("acalento".split("[lt]")[0]);
    }
} // Retorna: true, lento, aca
```

Atenção a dois detalhes importantes. O primeiro é que o método `matches` sempre tenta aplicar a ER na linha toda, não fazendo casamento parcial. É como se a expressão sempre viesse com `^` e `$` ao redor. Se quiser casar somente uma parte do texto, use `.*` no início e no fim da ER.

O segundo detalhe é que os retrovisores são referenciados como `$1`, `$2` etc. Se quiser inserir um `$` literal, é preciso escapá-lo. Duplamente: `\\$`. Java não traz o conceito de ER crua. A seguir um exemplo similar ao anterior, porém agora utilizando o pacote `regex`.

```
// usandoRegex.java
```

```
import java.util.regex.*;
class usandoRegex {
public static void main(String argv[]) {
    String texto = "0A1B2C3D4E5F";
    Pattern er = Pattern.compile("(?i)([a-z])");
    Matcher result = er.matcher(texto);
    System.out.println(result.matches());
    System.out.println(result.replaceAll("."));
    System.out.println(er.split(texto)[3]);
}
} // Retorna: false, 0.1.2.3.4.5., 3
```

O detalhe chato de lembrar é que os métodos `matches` e `replaceAll` fazem parte da classe `Matcher`, enquanto o `split` é da classe `Pattern`. Não se preocupe, por mais que memorize, você vai confundir :)

Detalhes, detalhes, detalhes:

No geral, o Java utiliza as mesmas expressões do Perl, com direito aos modificadores (`?idmsux`), vários barra-letras e metacaracteres modernos.

Uma das novidades são os quantificadores possessivos (`.+*`, `.++` etc.), que, além de gulosos, não cedem nenhum bit sequer para que a expressão case. Ou casa tudo ou nada feito.

Outra novidade são as listas com interseção, usando o operador `&&`. O resultado são os caracteres casados pelos dois componentes. Por exemplo, para casar as consoantes, faça `[a-z&&[^aeiou]]`. Traduzindo: case as letras de a a z e que não sejam as vogais.

A classe de caracteres POSIX traz uma sintaxe estranha (`\p{Alnum}`) e não inclui acentuação, mesmo que o sistema esteja configurado para o português.

Mais informações são encontradas em:

<http://aurelio.net/regex/java/>

## JavaScript / ActionScript

Característica	Como fazer



Busca	Métodos <code>String.search</code> , <code>String.match</code> , <code>RegExp.test</code> , <code>RegExp.exec</code>
Substituição	Método <code>String.replace</code>
Divisão	Método <code>String.split</code>
ER crua	/entre barras/
Ignore M/m	Modificador <code>i</code>
Global	Modificador <code>g</code>

JavaScript é a linguagem de programação das páginas de Internet. O suporte às expressões regulares foi incluído no JavaScript versão 1.2 de 1997, estando hoje presente na maioria dos navegadores, incluindo todas as versões do Firefox, Google Chrome e Safari. No Internet Explorer, surgiu na versão 4 (Windows 98).

ActionScript é a linguagem de programação utilizada pelo Adobe Flash (antigo Macromedia Flash). Desde sua versão 3 de 2006, também traz consigo suporte às expressões regulares.

Tanto JavaScript quanto ActionScript implementam o mesmo padrão ECMA-262 para as expressões, então sua aplicação é idêntica. Tudo o que você aprender neste tópico poderá ser usado nas duas linguagens.

O objeto que toma conta das expressões é o `RegExp`. Para criar uma instância nova, já informando a expressão regular desejada, basta colocá-la entre barras, como em `/[a-z]/`. Logo após a segunda barra, é possível colocar os modificadores de maiúsculas e minúsculas, e casamento global: `i` e `g`.

```
var er = /[a-z]/;    // minúsculas
var er = /[a-z]/i;   // minúsculas e maiúsculas
var er = /[a-z]/g;   // minúsculas, casamento global
var er = /[a-z]/ig;  // minúsculas e maiúsculas, global
```

Você também pode usar strings para instanciar um objeto `RegExp`. É útil para compor a expressão usando variáveis ou um texto vindo do usuário. Só lembre-se de que como é uma string, você precisará escapar as contrabarras: o barra-letra `\w`, por exemplo, deve ser informado como `\\w`. Veja como ficam os exemplos anteriores, usando string:

```
var er = new RegExp('[a-z]');  
var er = new RegExp('[a-z]', 'i');  
var er = new RegExp('[a-z]', 'g');  
var er = new RegExp('[a-z]', 'ig');
```

Há um pouco de confusão com relação aos métodos que usam expressões regulares, pois alguns estão no objeto `RegExp`, enquanto outros estão no `String`, e há uma certa duplicação de funcionalidade, veja:

- `RegExp.test()` – Testa se casou ou não (true/false)
- `RegExp.exec()` – Retorna array com o trecho casado ou null
- `String.search()` – Testa se casou e retorna o index ou -1
- `String.match()` – Retorna array com o trecho casado ou null
- `String.replace()` – Faz substituições, retorna string
- `String.split()` – Faz divisões, retorna array

Para simplificar a vida, eu recomendo usar somente os métodos do objeto `String` e esquecer que existem os outros. Assim você usará sempre a versão mais poderosa e não precisará se preocupar com a diferença de sintaxe.

Para apenas testar se uma expressão casa ou não com determinado texto, use o método `String.search()`. Ele retorna -1 quando a expressão não casar, então o teste fica assim:

```
var minha_expressao = /^java/i;  
if ("JavaScript".search(minha_expressao) != -1) {  
    alert("Casou");  
} else {  
    alert("Não casou");  
}
```

Você também pode usar as expressões entre barras diretamente, sem precisar instanciar antes o objeto `RegExp` em uma variável. Usarei esta notação nos exemplos seguintes para ficarem mais sucintos. Veja como fica o exemplo anterior:

```
if ("JavaScript".search(/^java/i) != -1) {  
    alert("Casou");  
}
```

```

} else {
    alert("Não casou");
}

```

Para testar a expressão e ao mesmo tempo obter informações sobre o casamento, use o método `String.match()`, que retorna um array ou `null`. Guarde o resultado em uma variável para poder acessá-lo depois.

```

var resultado = "JavaScript".match(/^java/i);
if (resultado) {
    console.log(resultado.length);    // 1
    console.log(resultado.index);    // 0
    console.log(resultado.input);    // "JavaScript"
    console.log(resultado[0]);        // "Java"
} else {
    console.log("não casou");
}

```

O array resultante traz em sua posição zero (`resultado[0]`) o trecho de texto casado pela expressão. Além do tradicional atributo `length` com o tamanho do array, há dois atributos adicionais: `index` com a posição inicial do trecho casado dentro da string original e `input` com a própria string original. Se sua expressão contém grupos, o array também trará o conteúdo casado de cada grupo.

```

var resultado = "31/12/1999".match(/^(..)\/(..)\/(....)$/);
if (resultado) {
    console.log(resultado.length);    // 4
    console.log(resultado.index);    // 0
    console.log(resultado.input);    // "31/12/1999"
    console.log(resultado[0]);        // "31/12/1999"
    console.log(resultado[1]);        // "31"
    console.log(resultado[2]);        // "12"
    console.log(resultado[3]);        // "1999"
}

```

Perceba que as barras literais da data precisaram ser escapadas como `\` para evitar que se confundam com as barras que delimitam a expressão em si. Se for um casamento global (modificador `g` na expressão), então tudo muda: o resultado será um array normal, sem atributos adicionais, que será povoado

com todas as ocorrências encontradas, e o conteúdo dos grupos é descartado. Útil para encontrar e guardar de uma só vez todas as ocorrências.

```
var resultado = "um dois tres quatro".match(/\w+/g);
if (resultado) {
    console.log(resultado.length);    // 4
    console.log(resultado[0]);        // "um"
    console.log(resultado[1]);        // "dois"
    console.log(resultado[2]);        // "tres"
    console.log(resultado[3]);        // "quatro"
}
```

Para lidar com strings multilinha, use o modificador `m` no final da expressão. Com ele, as âncoras `^` e `$` casam cada uma das linhas da string.

```
"1\n2\n3\n4".match(/^d/g);    // ["1"]
"1\n2\n3\n4".match(/^d/gm);   // ["1", "2", "3", "4"]
```

Você pode casar a quebra de linha diretamente, usando o barra-letra `\n` em sua expressão.

```
"1\n2\n3\n4".match(/^d\n/g);    // ["1\n"]
"1\n2\n3\n4".match(/^d\n/gm);   // ["1\n", "2\n", "3\n"]
```

Não há o modificador `s`, comum em outras linguagens, que faz o metacaractere ponto também casar o `\n`. É possível improvisar usando `[\S\s]`, que casa qualquer caractere, inclusive o `\n`.

```
"1\n2\n3\n4".match(/^1.*4$/);    // null
"1\n2\n3\n4".match(/^1[\S\s]*4$/); // ["1\n2\n3\n4"]
```

Para fazer substituições, utilize o método `String.replace()`, que por padrão substitui apenas a primeira ocorrência encontrada. Se precisar de uma substituição global, ou ignorar maiúsculas e minúsculas, ou casamento multilinha, adicione os modificadores no final da expressão.

```
"JavaScript".replace(/[a-z]/ , '.'); // J.vaScript
"JavaScript".replace(/[a-z]/g , '.'); // J...S.....
"JavaScript".replace(/[a-z]/gi , '.'); // .....
"1\n2\n3\n4".replace(/^d/g , '.');    // .\n2\n3\n4
"1\n2\n3\n4".replace(/^d/gm , '.');   // .\n.\n.\n.
"1\n2\n3\n4".replace(/\n/g , '.');    // 1.2.3.4
```

```
"1\n2\n3\n4".replace(/./g, '.'); // .\n.\n.\n.
"1\n2\n3\n4".replace(/\S\s/g, '.'); // .....
```

Os retrovisores são referenciados com um cifrão na frente. Então em vez de \1 use \$1. Há também o retrovisor especial \$& que guarda todo o trecho de texto casado pela expressão.

```
"12:34".replace(/(..):(..)/, '$1h $2min'); // 12h 34min
"JavaScript".replace(/.*/, '--$&--'); // --JavaScript--
```

Para substituições realmente estilosas, você pode usar uma função no lugar do texto substituto. Esta função receberá um número variável de argumentos, dependendo do número de grupos de sua expressão, e deve retornar uma string.

```
function data_por_extenso(texto_casado, grupo1, grupo2, grupo3) {
    var dia = grupo1;
    var mes = grupo2;
    var ano = grupo3;
    var meses = {
        '01': 'Jan', '02': 'Fev', '03': 'Mar',
        '04': 'Abr', '05': 'Mai', '06': 'Jun',
        '07': 'Jul', '08': 'Ago', '09': 'Set',
        '10': 'Out', '11': 'Nov', '12': 'Dez'
    };
    return dia + " de " + meses[mes] + " de " + ano;
}
var texto = "Hoje é dia 31/12/1999.";
var regex = /(\d\d)\.(\d\d)\.(\d\d\d\d)/;
var resultado = texto.replace(regex, data_por_extenso);
// Hoje é dia 31 de Dez de 1999.
```

Acentuação é um problema. Não há suporte às classes POSIX como [:alpha:] e [:lower:]. O que temos é o barra-letra \w, que casa letras e números. Porém, não há como confiar nas informações sobre localização informadas pelo navegador, então para casar acentos é preciso usar os remendos.

```
"Jáva".replace(/\w/g, '.'); // .á..
"Jáva".replace(/[\wÀ-Ü]/g, '.'); // ....
```

A divisão é feita com o método `String.split()`, que retorna um array com o texto dividido. Se um segundo argumento numérico for informado, o número de itens do array fica limitado a esse número. O texto excedente será descartado.

```
"um dois três".split(/\s+/);      // ["um", "dois", "três"]  
"um dois três".split(/\s+/, 2);  // ["um", "dois"]
```

Mais informações são encontradas em:

<http://aurelio.net/regex/javascript/>

## Lua

Característica	Como fazer
Busca	Funções <code>string.match</code> , <code>string.gmatch</code> , <code>string.find</code>
Substituição	Função <code>string.gsub</code>
Divisão	-
ER crua	-
Ignore M/m	-
Global	É o padrão

Lua é uma linguagem de programação genuinamente brasileira, criada na PUC do Rio de Janeiro em 1993. Prezando pela simplicidade e pelo tamanho compacto, é uma excelente opção para quem precisa embutir uma linguagem interpretada em seu programa. O jogo World of Warcraft, por exemplo, usa Lua para permitir ao usuário criar scripts e melhorias (add-ons).

Porém, o objetivo do tamanho diminuto, que é uma de suas maiores vantagens, impediu que os criadores adicionassem um suporte completo às expressões regulares. A solução encontrada foi implementar uma versão reduzida das expressões, que eles chamaram de “patterns”. Faltam alguns metacaracteres, outros são mais limitados, mas também há novidades que abrem outras possibilidades.

Mas deixemos os detalhes para depois. Primeiro, vamos aprender como

usar expressões regulares em Lua. Para testar uma expressão em um texto qualquer, basta usar a função `string.match`.

```
if string.match("Lua", "^L")
then
    print "Casou"
end
```

Além de testar a expressão, essa função ainda retorna o trecho de texto casado. Se a expressão não casar, o retorno será nulo (`nil`).

```
print(string.match("Lua", "^." ))    -- L
print(string.match("Lua", "^.." ))   -- Lu
print(string.match("Lua", "^..." )) -- Lua
print(string.match("Lua", "^X" ))    -- nil
```

Essa função também aceita receber um terceiro argumento numérico, que indica em qual posição iniciar a pesquisa. Pode ser útil se houver um trecho inicial que você quer simplesmente ignorar. É como se ele não existisse. Lembre-se de que os índices em Lua sempre iniciam em 1 e não 0.

```
print(string.match("Lua 1993", ".*", 5))  -- 1993
print(string.match("Lua 1993", ".*", -3))  -- 993
```

O primeiro exemplo mostra o curinga `.*` aplicado no texto “Lua 1993”, a partir da quinta posição, ou seja, os quatro primeiros caracteres foram descartados. Também é possível informar um número negativo, caso você queria iniciar a contagem a partir do final do texto. O segundo exemplo mostra que o `-3` fez o curinga ser aplicado somente nos três últimos caracteres do texto.

Mostrando ser bastante versátil, essa mesma função ainda traz outra funcionalidade: a captura do conteúdo de todos os grupos definidos em sua expressão. Esse retorno pode ser guardado em variáveis ou em uma tabela.

```
-- Método 1: Guardar os grupos em variáveis
--
g1, g2, g3 = string.match("Lua", "(.)(.)(.)")
print(g1)    -- L
print(g2)    -- u
print(g3)    -- a
```

```
-- Método 2: Guardar os grupos em uma tabela
--
grupos = { string.match("Lua", "(.)(.)(.)") }
print(grupos[1])    -- L
print(grupos[2])    -- u
print(grupos[3])    -- a
```

Perceba que a mesma função retorna o trecho casado ou o conteúdo dos grupos. Mas é um ou outro, não ambos. Funciona assim: se você não usar nenhum grupo em sua expressão, ela retorna todo o trecho casado. Mas se você usou um ou mais grupos, ela retorna somente o conteúdo desses grupos.

```
print(string.match("Lua", "..."))      -- Lua
print(string.match("Lua", "..(.)"))    -- a
print(string.match("Lua", ".(.)(.))")) -- u  a
print(string.match("Lua", "(.)(.)(.)")) -- L  u  a
```

No primeiro exemplo, não há grupos, então todo o trecho casado é retornado. No segundo exemplo, há um grupo no último ponto, então somente seu conteúdo é retornado e o restante (“Lu”) é descartado por não estar contido em grupo algum. De maneira similar, o terceiro exemplo retorna somente o conteúdo dos dois grupos. No último exemplo, como cada ponto tem seu próprio grupo, todos os caracteres foram retornados.

Mas se você quiser o trecho casado e os grupos ao mesmo tempo, basta uma pequena modificação: coloque um grupo adicional que englobe toda a expressão. Ele vai ser o primeiro item do retorno, contendo todo o trecho casado. O conteúdo dos outros grupos seguirá depois. Veja como ficam os mesmos exemplos anteriores, usando esta técnica:

```
print(string.match("Lua", "(..(..))")) -- Lua  a
print(string.match("Lua", "(.(.)(.))")) -- Lua  u  a
print(string.match("Lua", "((.)(.)(.))")) -- Lua  L  u  a
```

Há uma variação dessa função chamada `string.gmatch`. A diferença é que, em vez de retornar o texto encontrado, ela retorna uma função que pode ser usada diretamente em um loop. Isso facilita muito o processo de percorrer o resultado do casamento. Por exemplo, para percorrer cada palavra de um texto qualquer, ignorando espaços e pontuação, você pode fazer:



```

texto = "Lua? Linguagem legal, leve, limpa :)"
for palavra in string.gmatch(texto, "[A-Za-z]+")
do
    print(palavra)
end
-- Resultado:
-- Lua
-- Linguagem
-- legal
-- leve
-- limpa

```

Outra função utilizada para o teste de expressões é a `string.find`. Em vez de retornar o trecho de texto casado, ela retorna sua posição, com os números que indicam seu início e seu fim. Você pode usar essa informação para fatiar seu texto conforme necessário.

```

inicio, fim = string.find("Lua 1993", "[0-9]+")
print(inicio)    -- 5
print(fim)       -- 8

```

A substituição de textos é feita com a função `string.gsub` (não confunda com `string.sub`). É similar ao que vimos até aqui, porém o terceiro argumento é o texto substituto.

```

print(string.gsub("Lua", "[A-Z]", "."))    -- .ua    1
print(string.gsub("Lua", "[a-z]", "."))    -- L..    2

```

Note que a função retorna, além do texto casado, o número de substituições que foram feitas. Se você não pretende usar esse número, basta ignorá-lo. Nos exemplos seguintes, vou omiti-lo para facilitar o aprendizado.

Diferentemente de outras linguagens, os retrovisores são identificados por um %, ficando %1, %2, %3 e assim por diante. Isso vale tanto para a expressão quanto para o texto substituto.

```

print(string.gsub("1993", "(9)%1", ".."))    -- 1..3
print(string.gsub("Lua", "(.)(.)(.)", "%3%2%1"))    -- auL

```

Uma característica poderosa da `string.gsub` é que, em vez de informar diretamente o texto substituto no terceiro argumento, você também pode

passar uma função ou uma tabela.

```
-- Exemplos de substituição usando uma função
-- Usando string.upper
print(string.gsub("Lua", "[a-z]", string.upper))    -- LUA
-- Usando função definida pelo usuário
tira1 = function (n) return n-1 end
print(string.gsub("Lua 1993", "[0-9]", tira1))      -- Lua 0882
```

O primeiro exemplo usou a `string.upper` para tornar maiúsculas as letras que a expressão `[a-z]` casou. No segundo exemplo foi criada uma função que recebe um número `n` e retorna `n-1`, ou seja, subtrai uma unidade. Essa função foi chamada quatro vezes, subtraindo cada um dos dígitos de 1993. Se alterarmos a expressão para `[0-9]+`, todo o número será passado de uma vez e o resultado será 1992.

Mais uma vez é preciso atentar à diferença de comportamento caso sua expressão possua grupos. Se você não usar grupos, a função receberá um único argumento, que é o trecho casado. Caso contrário, cada grupo será um argumento para a função.

Independentemente do número de argumentos recebidos, o retorno da função deve ser sempre uma única string, que substituirá todo o trecho casado. Se você retornar `false` ou `nil`, o texto original não será alterado.

```
-- Exemplo de substituição usando uma tabela
texto = "Meu nome é NOME e nasci em MES/ANO."
tabela = { NOME = "Aurelio", ANO = "1977" }
print(string.gsub(texto, "[A-Z][A-Z]+", tabela))
-- Resultado: Meu nome é Aurelio e nasci em MES/1977.
```

Ao usar uma tabela, o trecho casado (ou o conteúdo do primeiro grupo) é pesquisado. Caso seja encontrado na tabela, seu valor correspondente será usado como texto substituto. Veja no exemplo que a chave `MES` não existe na tabela, então não foi substituída.

Agora que você já sabe casar e substituir textos usando Lua, é o momento de conhecer os detalhes que a fazem ser diferente das outras linguagens de programação com suporte às expressões regulares.



Metacaractere	Lua	Detalhes
.	.	Igual
[]	[]	Igual
[^]	[^]	Igual
?	?	Igual
*	*	Igual
+	+	Igual
{, }		Não tem
^	^	Igual
\$	\$	Igual
\b		Não tem
\	%	O % é usado para escapar
		Não tem
()	()	Grupos não são quantificáveis
\1	%1	É %1 para casar e substituir
. *	. *	Igual
??		Não tem
*?	-	Igual
+?		Não tem
{ }?		Não tem

A maioria dos metacaracteres básicos funciona de modo similar. Porém, não há chaves. Se você precisar fazer algo como `[0-9]{3,5}`, terá que colocar as repetições na mão: `[0-9][0-9][0-9][0-9]?[0-9]?`. Também não há o metacaractere ou, então, algo simples como `mac|linux|windows` não é válido, você terá de fazer três pesquisas separadas.

Os grupos são limitados à captura de textos, pois não é possível quantificá-

los. Por exemplo: (Lua)\*, (Lua)+ e (Lua)? são inválidos. Você só pode usar esses quantificadores após caracteres normais, listas e classes de caracteres. Exemplos: Luas?, [0-9]+ e %d+.

Também não há as versões não gulosas dos quantificadores. A única exceção é o asterisco, cujo similar não guloso é o hífen. Desse modo, .\* é o curinga guloso e .- é seu complemento não tão faminto.

O escape é o %, então, para casar um asterisco literal, use %\*. De maneira similar, %[, %., %+, %? e %( são mais alguns exemplos de metacaracteres escapados. Para representar um % literal, use %%. Lembre-se de que sua expressão também é uma string que requer os escapes normais com a contrabarra caso você precise usar aspas ou contrabarras literais: \", \\.

Lua também possui alguns atalhos práticos, chamados classes de caracteres. Eles simplificam o uso de listas comuns como [A-Za-z] e [0-9].

Classe	Descrição	Classe	Descrição
%a	Letras	%A	Não letras
%c	Caracteres de controle	%C	Não caracteres de controle
%d	Dígitos	%D	Não dígitos
%l	Minúsculas	%L	Não minúsculas
%p	Pontuação	%P	Não pontuação
%s	Caracteres brancos	%S	Não caracteres brancos
%u	Maiúsculas	%U	Não maiúsculas
%w	Letras e números	%W	Não letras e números
%x	Números hexadecimais	%X	Não números hexadecimais
%z	O caractere zero \000	%Z	Não zero

Há similaridade com as classes POSIX e os barra-letas, então você já entende o funcionamento desses metacaracteres. Usar as letras em maiúsculas é o mesmo que fazer uma classe negada, invertendo seu significado. Assim, em vez de fazer [^A-Za-z] ou [^%a], basta fazer %A.

Usando essas classes, você pode fazer `%d%d/%d%d/%d%d%d%d` para casar datas e `%d%d%d%.%d%d%d%.%d%d%d%-%d%d` para casar um número de CPF. Note que, como não há chaves, é preciso colocar todos os dígitos um a um, também sendo preciso escapar o ponto e o hífen.

Uma vantagem de usar essas classes é ganhar de brinde o suporte à acentuação, contanto que seu sistema esteja corretamente configurado para o português. Assim, `%a`, `%l`, `%u` e `%w` também casarão as letras acentuadas. Use a função `os.setlocale` caso seja necessário.

```
print(string.gsub("Lúa", "%a", "."))    -- .ú.
os.setlocale("pt_BR")
print(string.gsub("Lúa", "%a", "."))    -- ...
```

Unicode, porém, não é suportado. Se você está em um sistema UTF-8, prepare-se para algumas surpresas não desejadas relacionadas ao tamanho das strings. Veja nos exemplos seguintes como o texto `Lúa` é tratado como se possuísse quatro caracteres.

```
os.setlocale("pt_BR.UTF-8")
print(string.match("Lúa", "^...$"))    -- nil
print(string.match("Lúa", "^....$"))   -- Lúa
print(string.gsub("Lúa", "%a", "."))   -- ....
```

Mais informações são encontradas em:

<http://aurelio.net/regex/lua/>

## .NET

Característica	Como fazer
Busca	Métodos <code>IsMatch</code> , <code>Match</code> , <code>Matches</code>
Substituição	Método <code>Replace</code>
Divisão	Método <code>Split</code>
ER crua	É o padrão
Ignore M/m	Modificadores <code>(?i)</code> , <code>RegexOptions.IgnoreCase</code>
Global	É o padrão

---

As expressões regulares são cidadãs de primeira classe no framework .NET, pois fazem parte da biblioteca padrão. Assim, programas em várias linguagens podem usar as expressões diretamente, sem precisar de instalação ou configuração adicional.

Veremos exemplos de uso do framework nas linguagens C# e Visual Basic. Você perceberá que as diferenças são mínimas, apenas nos detalhes da sintaxe de cada linguagem. Mas classes, métodos, propriedades e comportamento das expressões regulares são exatamente os mesmos, tudo sendo unificado. Se você programa em alguma outra linguagem do framework .NET, basta converter os exemplos para sua sintaxe.

As classes das nossas queridas expressões estão no módulo `System.Text.RegularExpressions`. Para não precisar digitar todo esse longo caminho cada vez que for utilizá-lo, importe todo o módulo já no início do seu programa.

#### u C#

```
using System.Text.RegularExpressions;
```

#### u Visual Basic

```
Imports System.Text.RegularExpressions
```

Os exemplos seguintes mostrarão apenas os comandos relativos ao uso das expressões, mas tenha em mente que seu programa também deve ter toda a estrutura padrão que a linguagem requer. Para facilitar, segue o esqueleto básico para você poder fazer seus testes:

#### u C#

```
using System;
using System.Text.RegularExpressions;
class Testando
{
    static void Main()
    {
        //
```

```

        // Seu código vai aqui
    //
}
}

```

## u Visual Basic

```

Imports System.Text.RegularExpressions
Module Testando
    Sub Main()
        '
        ' Seu código vai aqui
        '
    End Sub
End Module

```

Então, basta colocar os exemplos ali no meio e pronto. Para começar, veremos como testar se uma expressão casou ou não em determinado texto (ou variável). Basta usar o método `Regex.IsMatch`.

## u C#

```

if (Regex.IsMatch("CSharp", @"^[A-Z]"))
    Console.WriteLine("Casou");
else
    Console.WriteLine("Não casou");

```

## u Visual Basic

```

If Regex.IsMatch("VisualBasic", "^[A-Z]")
    Console.WriteLine("Casou")
Else
    Console.WriteLine("Não casou")
End If

```

No C#, é preciso usar a `@` na frente da string com a expressão regular para torná-la crua, evitando dores de cabeça com o escape da contrabarra. Senão, um simples `\w` terá de ser escrito como `\\w`. Acostume-se sempre a usar a `@` antes de suas expressões, mesmo que não haja nenhuma contrabarra. No Visual Basic, não se preocupe, pois as strings são sempre cruas.

Se você quiser que sua expressão ignore a diferença entre maiúsculas e minúsculas, basta adicionar um terceiro argumento. A classe `RegexOptions` guarda as opções disponíveis para modificar o comportamento de sua expressão. A opção que usaremos chama-se `IgnoreCase`.

u **C#**

```
if (Regex.IsMatch("CSharp", @"^[a-z]", RegexOptions.IgnoreCase))
    Console.WriteLine("Casou");
else
    Console.WriteLine("Não casou");
```

u **Visual Basic**

```
If Regex.IsMatch("VisualBasic", "^[a-z]", RegexOptions.IgnoreCase)
    Console.WriteLine("Casou")
Else
    Console.WriteLine("Não casou")
End If
```

Há um caminho mais curto, que é simplesmente adicionar o modificador (`?i`) no início da expressão. O efeito é o mesmo do exemplo anterior, com a vantagem de ter digitado menos.

u **C#**

```
if (Regex.IsMatch("CSharp", @"(?i)^[a-z]"))
    Console.WriteLine("Casou");
else
    Console.WriteLine("Não casou");
```

u **Visual Basic**

```
If Regex.IsMatch("VisualBasic", "(?i)^[a-z]")
    Console.WriteLine("Casou")
Else
    Console.WriteLine("Não casou")
End If
```

Outra maneira de utilizar uma expressão regular no framework .NET é criar um objeto do tipo `Regex` e, depois, acessar seus métodos diretamente. Com isso, a expressão será interpretada (compilada) uma única vez, rendendo um



ganho de performance durante a execução do programa.

u **C#**

```
Regex ER = new Regex(@"^[A-Z]");  
if (ER.IsMatch("CSharp"))  
    Console.WriteLine("Casou");  
else  
    Console.WriteLine("Não casou");
```

u **Visual Basic**

```
Dim ER as Regex = new Regex("^[A-Z]")  
If ER.IsMatch("VisualBasic")  
    Console.WriteLine("Casou")  
Else  
    Console.WriteLine("Não casou")  
End If
```

Perceba como agora o método `IsMatch` foi chamado diretamente da variável `ER`. Salvo em casos específicos, é uma boa prática guardar suas expressões em variáveis e usar tais variáveis adiante, em vez de chamar os métodos estáticos (`Regex.*`).

As possíveis opções que você pode dar à expressão são colocadas já na sua criação, como um segundo elemento. Por exemplo, para tornar a expressão anterior indiferente a maiúsculas e minúsculas, basta fazer:

u **C#**

```
Regex ER = new Regex(@"^[A-Z]", RegexOptions.IgnoreCase);
```

u **Visual Basic**

```
Dim ER as Regex = new Regex("^[A-Z]", RegexOptions.IgnoreCase)
```

O uso do modificador `(?i)` também continua valendo, então outra maneira de fazer isso seria:

u **C#**

```
Regex ER = new Regex(@"(?i)^[A-Z]");
```

## u Visual Basic

```
Dim ER as Regex = new Regex("(?i)^[A-Z]")
```

Mas apenas testar se a expressão casou ou não, como fizemos até aqui, é muito limitado. Geralmente é preciso obter informações sobre o que casou e onde casou. Entram em cena o método Match e suas propriedades.

## u C#

```
// Compila a expressão
Regex ER = new Regex(@"^[A-Z]{2}");
// Faz o casamento
Match Resultado = ER.Match("CSharp");
// Casou?
if (Resultado.Success)
{
    Console.WriteLine("Texto: " + Resultado.Value);    // CS
    Console.WriteLine("Início: " + Resultado.Index);   // 0
    Console.WriteLine("Tamanho: " + Resultado.Length); // 2
}
```

## u Visual Basic

```
' Compila a expressão
Dim ER as Regex = new Regex("^[A-Z][a-z]+")
' Faz o casamento
Dim Resultado as Match = ER.Match("VisualBasic")
' Casou?
If Resultado.Success
    Console.WriteLine("Texto: " & Resultado.Value)    ' Visual
    Console.WriteLine("Início: " & Resultado.Index)   ' 0
    Console.WriteLine("Tamanho: " & Resultado.Length) ' 6
End If
```

Ao usar o método Match, é retornado um objeto do tipo Match, que traz informações sobre o casamento. Em primeiro lugar, use a propriedade Success para saber se a expressão casou ou não no texto. Dentro de value está guardado o trecho de texto casado pela expressão, enquanto Index e Length indicam a posição de início e o tamanho desse trecho, respectivamente.

Se sua expressão contém grupos, basta utilizar a propriedade `Groups`, que faz parte do objeto `Match`. Informe o número do grupo desejado ou zero para obter todo o trecho de texto casado pela expressão. Note que o conteúdo do grupo zero é o mesmo que o da propriedade `value` do exemplo anterior.

#### u C#

```
// Compila e casa a expressão
Regex ER = new Regex(@"^(..)(..)(..)");
Match Resultado = ER.Match("CSharp");
// Mostra o conteúdo dos grupos
if (Resultado.Success)
{
    Console.WriteLine(Resultado.Groups[0]); // CSharp
    Console.WriteLine(Resultado.Groups[1]); // CS
    Console.WriteLine(Resultado.Groups[2]); // ha
    Console.WriteLine(Resultado.Groups[3]); // rp
}
```

#### u Visual Basic

```
' Compila e casa a expressão
Dim ER as Regex = new Regex("^(.)(.)(.)(.)")
Dim Resultado as Match = ER.Match("VisualBasic")
' Mostra o conteúdo dos grupos
If Resultado.Success
    Console.WriteLine(Resultado.Groups(0)) ' Visu
    Console.WriteLine(Resultado.Groups(1)) ' V
    Console.WriteLine(Resultado.Groups(2)) ' i
    Console.WriteLine(Resultado.Groups(3)) ' s
    Console.WriteLine(Resultado.Groups(4)) ' u
End If
```

Para fazer substituições, use o método `Replace`. Basta passar o texto e a expressão, e o retorno será o texto modificado. Todas as ocorrências são substituídas (global). Se precisar ignorar a diferença entre maiúsculas e minúsculas, use o modificador `(?i)` no início da expressão.

#### u C#

```
Regex.Replace("CSharp", @"[a-z]", ".");           // CS....
Regex.Replace("CSharp", @"(?i)[a-z]", ".");       // .....
```

## u Visual Basic

```
Regex.Replace("VisualBasic", "[a-z]", ".")        ' V.....B....
Regex.Replace("VisualBasic", "(?i)[a-z]", ".")    ' .....
```

Novamente, também é possível compilar a expressão e, depois, acessar o método Replace diretamente pela variável. A grande vantagem é que, fazendo dessa maneira, você pode utilizar os argumentos opcionais disponíveis no método. O terceiro argumento indica quantas substituições devem ser feitas. O valor padrão é -1, que significa trocar todas as ocorrências. Se colocar o valor zero, nenhuma substituição será feita. O quarto argumento indica a posição inicial em que as substituições devem ser feitas, deixando intocado tudo o que vier antes.

## u C#

```
Regex ER = new Regex(@"\w");
Console.WriteLine(ER.Replace("CSharp", "."));           // .....
Console.WriteLine(ER.Replace("CSharp", ".", -1));      // .....
Console.WriteLine(ER.Replace("CSharp", ".", 0));       // CSharp
Console.WriteLine(ER.Replace("CSharp", ".", 2));       // ..harp
Console.WriteLine(ER.Replace("CSharp", ".", 4));       // ....rp
Console.WriteLine(ER.Replace("CSharp", ".", 2, 3));    // CSh..p
```

## u Visual Basic

```
Dim ER as Regex = new Regex("\w")
Console.WriteLine(ER.Replace("VisualBasic", "."))      ' .....
Console.WriteLine(ER.Replace("VisualBasic", ".", -1))  ' .....
Console.WriteLine(ER.Replace("VisualBasic", ".", 0))   ' VisualBa
Console.WriteLine(ER.Replace("VisualBasic", ".", 2))   ' ..sualBa
Console.WriteLine(ER.Replace("VisualBasic", ".", 6))   ' .....Ba
Console.WriteLine(ER.Replace("VisualBasic", ".", 2, 4)) ' Visu..Ba
```

Na acentuação, uma grata surpresa: simplesmente funciona. As expressões contam com um bom suporte ao Unicode, então o \w também inclui os caracteres acentuados. Se por algum motivo você precisar do \w sem

acentuação ([a-zA-Z0-9\_]), use a opção `RegexOptions.ECMAScript`, que faz as expressões se comportarem como se estivessem no ambiente JavaScript/ActionScript.

#### u C#

```
Regex ER1 = new Regex(@"\w");
Regex ER2 = new Regex(@"\w", RegexOptions.ECMAScript);
Console.WriteLine(ER1.Replace("Çshárp", ".")); // .....
Console.WriteLine(ER2.Replace("Çshárp", ".")); // Ç..á..
```

#### u Visual Basic

```
Dim ER1 as Regex = new Regex("\w")
Dim ER2 as Regex = new Regex("\w", RegexOptions.ECMAScript)
Console.WriteLine(ER1.Replace("VisuálBásíc", ".")) ' .....
Console.WriteLine(ER2.Replace("VisuálBásíc", ".")) ' ....á....í.
```

Os retrovisores são indicados com o cifrão (\$1, \$2 etc.) e você também pode usar \$0 ou \$& para referenciar todo o trecho de texto casado pela expressão.

#### u C#

```
Regex ER = new Regex(@"^(..)(..)(..)*");
Console.WriteLine(ER.Replace("CSharp", "$1-$2-$3")); // CS-ha-rp
Console.WriteLine(ER.Replace("CSharp", "-$0-")); // -CSharp-
Console.WriteLine(ER.Replace("CSharp", "-$&-")); // -CSharp-
```

#### u Visual Basic

```
Dim ER as Regex = new Regex("^(..)(..)(..)*")
Console.WriteLine(ER.Replace("VisualBasic", "$1-$2-$3")) ' Vi-su-a
Console.WriteLine(ER.Replace("VisualBasic", "-$0-")) ' -Visual
Console.WriteLine(ER.Replace("VisualBasic", "-$&-")) ' -Visual
```

A divisão é feita com o método `split`, que retorna um array de strings que contém os pedaços resultantes após o corte. Você pode, ainda, informar um número para limitar o tamanho desse array. Neste caso, o último item do array trará o texto restante, que ficou sem corte.

#### u C#

```
Regex ER = new Regex(@"[/.]");  
string[] Array1 = ER.Split("31/12/99");           // {"31", "12", "99"}  
string[] Array2 = ER.Split("31/12/99", 2);        // {"31", "12/99"}
```

## u Visual Basic

```
Dim ER as Regex = new Regex("[/.]")  
Dim Array1 as String() = ER.Split("31/12/99")      ' {"31", "12", "9"}  
Dim Array2 as String() = ER.Split("31/12/99", 2)    ' {"31", "12/99"}
```

Mais informações são encontradas em:

<http://aurelio.net/regex/dotnet/>

## Perl

Característica	Como fazer
Busca	Comando m//, operador =~
Substituição	Comando s///
Divisão	Comando split
ER crua	'entre aspas simples'
Ignore M/m	Modificadores i, (?i)
Global	Modificador g

A linguagem Perl ditou o rumo da evolução das expressões regulares, sendo a primeira a introduzir os metacaracteres modernos e outras novidades avançadas. Seu exemplo foi seguido e hoje a maioria das linguagens modernas tem um suporte completo aos metacaracteres da nova geração.

As expressões fazem parte do coração da linguagem, sendo cidadãos de primeira classe no mundo Perl. Para definir uma expressão nova, basta colocá-la entre barras.

Um detalhe importante é que, apesar de as barras delimitarem uma expressão, variáveis como \$foo são interpretadas ali dentro. Para que sua expressão seja realmente crua, sem nenhum pré-processamento, coloque-a

entre aspas simples. Apesar disso, os exemplos seguintes terão barras por sua utilização ser mais comum entre os programadores.

Para casar uma expressão, utilize o operador `=~`. Para inverter a lógica do teste, use o operador `!~`, que retorna sucesso se a expressão não casar com o texto.

```
if ("Perl" =~ /^Pe/) {
    print "Casou";
}
if ("Perl" !~ /^Ja/) {
    print "Não casou";
}
print "Sim" if "Perl" =~ /^Pe/;
print "Não" if "Perl" !~ /^Ja/;
```

Se sua expressão possuir o caractere `/`, ele deverá ser escapado `\` para evitar confusão com as barras delimitadoras. Outra alternativa é usar outro caractere como delimitador em vez das barras, como `%` ou `@`. Basta colocar a letra `m` no início da expressão.

```
if ("Perl" =~ m/^Pe/) {
    print "Casou";
}
```

O Perl possui uma variável mágica chamada `$_`, que é chamada de variável-padrão. É utilizada quando um comando deveria receber um argumento, mas o programador não o especificou. Então, a linguagem automaticamente usa o que estiver dentro de `$_`. Se você está acostumado a usar essa variável especial, pode simplificar o comando, testando diretamente a expressão.

```
$_ = "Perl";
if (/^Pe/) {
    print "Casou";
}
print "Casou" if /^Pe/;
```

Para testar uma expressão ignorando a diferença entre maiúsculas e minúsculas, adicione o modificador `i` logo após a segunda barra. Outra alternativa é utilizar o metacaractere moderno `(?i)`.

```
print "Casou" if "Perl" =~ /^pe/i;
print "Casou" if "Perl" =~ /^(?i)pe/;
```

Cada vez que você testa uma expressão regular, o trecho casado por ela é automaticamente guardado dentro da variável especial \$&. Se você usou grupos, o conteúdo deles é guardado nas variáveis \$1, \$2, \$3 e assim por diante.

```
"Perl" =~ /(.)()(.)()/;
print $&;    # Perl
print $1;    # P
print $2;    # e
print $3;    # r
print $4;    # l
```

Se você preferir guardar o conteúdo de todos os grupos em um array específico, use a atribuição. Mas note que o primeiro item do array (item zero) guardará o conteúdo do primeiro grupo (número um).

```
@resultado = ("Perl" =~ /(.)()(.)()/);
print $resultado[0];    # P
print $resultado[1];    # e
print $resultado[2];    # r
print $resultado[3];    # l
```

A substituição de textos é feita pelo comando s, usando a sintaxe herdada do Unix: s/expressão/texto/. O comando deve ser usado em conjunto com o operador =~ e o resultado é guardado diretamente na própria variável que contém o texto.

```
$texto = "Perl";
$texto =~ s/[a-z]/./;    # P.rl
```

Apenas a primeira ocorrência é substituída. Para fazer uma substituição global, use o modificador g logo após a última barra do comando s. Você também pode usar o modificador i para fazer com que a expressão considere como iguais as letras maiúsculas e minúsculas.

```
$texto = "Perl";
$texto =~ s/[a-z]/./g;    # P...
$texto = "Perl";
```



```
$texto =~ s/[a-z]/./gi;    # ....
```

As variáveis especiais \$1, \$2, \$3 e outras que guardam o conteúdo dos grupos podem ser usadas no texto substituto, funcionando como retrovisores. Outra variável especial e útil na substituição é a \$&, que guarda todo o trecho casado pela expressão.

```
$texto = "Perl";
$texto =~ s/(..)/$1$1/;    # PePe
$texto = "Perl";
$texto =~ s/.*/--$&--/;    # --Perl--
```

Um modificador especial que também pode ser usado na substituição é o e, que executa comandos. Depois de feita a substituição, o texto resultante é executado por um eval{...} e o resultado desse comando é, então, retornado como o texto substituto, sendo útil para fazer algum tipo de processamento durante a substituição.

```
$texto = "Perl";
$texto =~ s/.*/reverse $&/e;    # lreP
```

O suporte à localização foi incluído a partir do Perl versão 5. Se seu sistema estiver corretamente configurado para o português, o barra-letra \w e as classes POSIX irão casar caracteres acentuados também.

```
$texto = "Pârl";
$texto =~ s/\w/./g;    # .â..
use locale;
$texto = "Pârl";
$texto =~ s/\w/./g;    # ....
```

Se a localização do sistema não estiver configurada para o português, você poderá forçar isso dentro de seu programa com o comando setlocale.

```
use POSIX qw(locale_h);
setlocale(LC_CTYPE, "pt_BR.ISO8859-1");
```

O suporte ao Unicode iniciou-se no Perl versão 5.6, então, se você manipula textos em UTF-8, certifique-se de que sua versão da linguagem está atualizada. Caso enfrente problemas com acentos, deixe clara qual a codificação de seu programa.

```
use utf8;
```

O comando `split` divide um texto, retornando um array com os pedaços resultantes dos cortes. Se você informar um terceiro argumento numérico, ele indicará o tamanho máximo do array. Nesse caso, o último item concentrará o restante do texto não dividido.

```
@resultado = split /\s+/, "P e r l";
print @resultado[0]; # P
print @resultado[1]; # e
print @resultado[2]; # r
print @resultado[3]; # l
@resultado = split /\s+/, "P e r l", 3;
print @resultado[0]; # P
print @resultado[1]; # e
print @resultado[2]; # r l
```

Mais informações são encontradas em:

<http://aurelio.net/regex/perl/>

## PHP (PCRE)

Característica	Como fazer
Busca	Função <code>preg_match</code>
Substituição	Função <code>preg_replace</code>
Divisão	Função <code>preg_split</code>
ER crua	'entre aspas simples'
Ignore M/m	Modificadores <code>i</code> , <code>(?i)</code>
Global	É o padrão

A linguagem PHP usa a excelente biblioteca PCRE para trazer o suporte às expressões regulares. Os metacaracteres são poderosos, compatíveis com os da linguagem Perl, e a sua execução é muito eficiente. Todas as funções relacionadas às expressões possuem o mesmo prefixo `preg_`.

Função	Descrição
--------	-----------

<code>preg_filter</code>	Substituição condicional
<code>preg_grep</code>	Pesca itens de um array
<code>preg_last_error</code>	Código de erro da última execução
<code>preg_match_all</code>	Casa globalmente e retorna o resultado
<code>preg_match</code>	Casa e retorna o resultado
<code>preg_quote</code>	Escapa metacaracteres
<code>preg_replace_callback</code>	Substituição usando função
<code>preg_replace</code>	Substituição
<code>preg_split</code>	Divisão

Em primeiro lugar, para não ter problemas com os escapes da linguagem, sempre coloque sua expressão entre aspas simples. Além disso, você também deve colocar a expressão dentro dos delimitadores que o PCRE requer, que geralmente são as barras: `'/[a-z]/'`. Se houver barras no meio de sua expressão, você precisará escapá-las `\` ou então deverá usar outros símbolos como delimitadores.

```
if (preg_match('/^PH/', "PHP PCRE")) {
    print "Casou";
}
if (preg_match('%^PH%', "PHP PCRE")) {
    print "Casou";
}
```

Para casar uma expressão ignorando a diferença entre maiúsculas e minúsculas, basta adicionar o modificador `i` após a segunda barra ou utilizar o metacaractere modernoso `(?i)` no início da expressão.

```
if (preg_match('/^ph/i', "PHP PCRE")) {
    print "Casou";
}
if (preg_match('/(?i)^ph/', "PHP PCRE")) {
    print "Casou";
}
```

Mas além de testar expressões, o `preg_match` também serve para guardar informações sobre o casamento. Informe um terceiro argumento que ali será guardado um array com o trecho casado e o conteúdo de todos os grupos marcados.

```
preg_match('/(...) (...).../', "PHP PCRE", $resultado);
print $resultado[0];    // PHP PCRE
print $resultado[1];    // PHP
print $resultado[2];    // PC
print $resultado[3];    // RE
```

A substituição é feita com a função `preg_replace`, que é global por padrão, ou seja, todas as ocorrências são substituídas. Um quarto argumento numérico pode ser informado para limitar o número de substituições a ser feitas. O modificador `i` pode ser usado para ignorar a diferença entre maiúsculas e minúsculas.

```
print preg_replace('/[a-z]/i', ".", "PHP PCRE");           // ... ...
print preg_replace('/[a-z]/i', ".", "PHP PCRE", 4);        // ... .CR
```

Os retrovisores são indicados como `$1`, `$2` e assim por diante. O retrovisor especial `$0` representa o trecho casado pela expressão regular.

```
print preg_replace('/(.*?) (.*?)/', '$1 ($2)', "PHP PCRE"); // PHP (
print preg_replace('/....$/ '      , '($0)'      , "PHP PCRE"); // PHP (
```

A acentuação pode ser casada com o barra-letra `\w` ou com as classes POSIX, contanto que seu sistema esteja corretamente configurado para o português. Se seu texto estiver codificado em UTF-8, coloque o modificador `u` após a segunda barra da expressão, para que o PHP saiba como lidar com ele.

```
print preg_replace('/[[:alpha:]]/u', '.', "pêcêérreê");    // .....
print preg_replace('/\w/u',          '.', "pêcêérreê");    // .....
```

A divisão é feita com a função `preg_split`. Se precisar que a expressão ignore a diferença entre maiúsculas e minúsculas, coloque o modificador `i` após a segunda barra. Você também pode passar um terceiro argumento numérico, que indicará o tamanho do array resultante. Neste caso, o último item será o texto restante não dividido.

```

print_r(preg_split('/\s+/', "PHP PC RE"));
//
// Array
// (
//     [0] => PHP
//     [1] => PC
//     [2] => RE
// )
print_r(preg_split('/\s+/', "PHP PC RE", 2));
//
// Array
// (
//     [0] => PHP
//     [1] => PC RE
// )

```

Antes do PHP versão 5.3.0, de 2009, existia outro conjunto de funções que lidavam com expressões regulares, chamadas funções POSIX. Elas foram aposentadas da linguagem. Se você precisa dar manutenção em códigos antigos, veja mais informações no próximo tópico: PHP (POSIX). Caso contrário, pode ignorá-lo.

Mais informações são encontradas em:

<http://aurelio.net/regex/php/>

## PHP (POSIX)

Característica	Como fazer
Busca	Função ereg
Substituição	Função ereg_replace
Divisão	Função split
ER crua	'entre aspas simples'
Ignore M/m	Funções eregi, eregi_replace, spliti
Global	É o padrão

As funções POSIX eram a maneira tradicional de usar expressões regulares no PHP, mas foram aposentadas (deprecated) em 2009, na versão 5.3.0 da linguagem. Desde então, as funções PCRE, que são bem mais poderosas e eficientes, ocuparam seu lugar.

O conselho é atualizar todas as suas funções POSIX para os equivalentes PCRE, assim que possível. Veja quais são as funções equivalentes:

POSIX	PCRE
ereg()	preg_match()
eregi()	preg_match()
ereg_replace()	preg_replace()
eregi_replace()	preg_replace()
split()	preg_split()
spliti()	preg_split()
sql_regcase()	—

Mas se você precisa lidar com código legado e está preso às funções antigas, veremos a seguir os detalhes para tornar sua vida mais tranquila.

Em primeiro lugar, para especificar uma expressão crua e não ter problemas com escapes, sempre a coloque entre aspas simples. A função que serve para testar uma expressão é a `ereg`.

```
if (ereg('^PH', "PHP")) {  
    print "Casou";  
}
```

Se passado um terceiro argumento para a função, nele será criado um array que conterá o resultado do casamento, com o trecho casado na primeira posição, seguido do conteúdo de todos os grupos marcados.

```
ereg('(.)(.)(.)', "PHP", $resultado);  
print $resultado[0];    // PHP  
print $resultado[1];    // P  
print $resultado[2];    // H
```

```
print $resultado[3];    // P
```

A maneira de o PHP ignorar a diferença entre maiúsculas e minúsculas é, no mínimo, curiosa. Em vez de um modificador ou argumento adicional, é o nome da função que muda. Basta adicionar a letra i no final, ficando eregi.

```
if (eregi('^ph', "PHP")) {  
    print "Casou";  
}
```

A função que faz substituições é a ereg\_replace. Adicionando a letra i em seu nome, ela ignora a diferença entre maiúsculas e minúsculas: eregi\_replace. Não há como especificar o número de substituições, pois sempre todas as ocorrências são trocadas.

```
print ereg_replace( '[A-Z]', ".", "PHP");    // ...  
print eregi_replace( '[a-z]', ".", "PHP");    // ...
```

Ao usar retrovisores, lembre-se de sempre colocá-los entre aspas simples também, para evitar problemas com os escapes. Use o \0 para referenciar todo o trecho casado pela expressão.

```
print ereg_replace( '^ (P) .*', '\1-\1', "PHP");    // P-P  
print ereg_replace( '^ (P) .*', '\0-\0', "PHP");    // PHP-PHP
```

Acentuação é um pouco complicado e talvez seja preciso experimentação até chegar ao resultado desejado. Não há suporte ao barra-letra \w, então use as classes POSIX. Primeiro, tente a substituição normal com a classe alpha.

```
print ereg_replace( '[A-Za-z]', ".", "Pêagápê");    // .ê..á.ê  
print ereg_replace( '[:alpha:]', ".", "Pêagápê");    // .....
```

Se não funcionar, defina a localização “na mão” para que o PHP saiba quais caracteres fazem parte do alfabeto. Confira a codificação de seu arquivo e a do texto que você está manipulando; essas codificações devem ser preferencialmente iguais. Informe a língua e a codificação com o setlocale. Veja a seguir alguns valores que você pode usar: pt\_BR, pt\_BR.IS08859-1 e pt\_BR.UTF-8.

```
setlocale(LC_ALL, 'pt_BR.UTF-8');
```

Outra alternativa é usar as funções mb (multibyte). A partir do PHP 4.2,

foram adicionadas as funções `mb_ereg`, `mb_eregi`, `mb_ereg_replace`, `mb_eregi_replace` e `mb_split`, que são similares às funções vistas neste tópico, porém elas possuem suporte ao Unicode.

Veja também `mb_internal_encoding` e `mb_regex_encoding`, que são funções para mostrar o valor atual da codificação e também modificá-lo.

```
mb_regex_encoding("UTF-8");  
print mb_ereg_replace('[[:alpha:]]', ".", "Pêagápê"); // .....
```

A divisão é feita com a função `split`, que pode receber um terceiro argumento numérico, limitando o tamanho do array resultante. Sua irmã `spliti` funciona de modo semelhante, porém ignora a diferença entre maiúsculas e minúsculas no casamento.

```
print_r(split('-', "P---H---P"));  
//  
// Array  
// (  
//    [0] => P  
//    [1] => H  
//    [2] => P  
// )  
print_r(split('-', "P---H---P", 2));  
//  
// Array  
// (  
//    [0] => P  
//    [1] => H---P  
// )
```

Mais informações são encontradas em:

<http://aurelio.net/regex/php/>

## PowerShell

PowerShell é a shell moderna dos sistemas Windows, lançada em 2006 com o Windows XP. Construída sobre a plataforma .NET, traz para a linha de comando uma visão orientada a objetos: diferentemente do Unix, o pipe não



passa textos de um comando para outro, mas sim objetos, que podem ser manipulados.

As expressões regulares são informadas por meio de operadores que veremos a seguir, mas por baixo dos panos são processadas pelo módulo oficial do .NET, o `System.Text.RegularExpressions`, que é bem poderoso e suporta a maioria dos metacaracteres modernos do Perl.

Para casar um texto, basta usar o operador `-Match`, seguido da expressão.

```
PS> 'PowerShell' -Match '[A-Z]'
True
PS> 'PowerShell' -Match '[0-9]'
False
PS>
```

O retorno será um valor booleano, `True` ou `False`, conforme o texto tenha casado ou não com a expressão informada. Você poderá usar este retorno em um comando como o `if`, para decidir o que fazer. No exemplo seguinte, é mostrada uma mensagem na tela informando se a expressão casou ou falhou.

```
PS> if ('PowerShell' -Match '[A-Z]') { 'Casou' } else { 'Falhou' }
Casou
PS> if ('PowerShell' -Match '[0-9]') { 'Casou' } else { 'Falhou' }
Falhou
PS>
```

Outro caso de uso do operador `-Match` é dentro de comandos que testam condições para filtrar resultados, como o `Where-Object`. Por exemplo, filtrar a lista de serviços do Windows (`Get-Service`) para mostrar somente aqueles cujo nome (`$_Name`) termina em “host” ou “server”.

```
PS> Get-Service | Where-Object {$_Name -Match '(host|server)$'}
Status      Name           DisplayName
-----
Stopped     Eaphost        Protocolo de Autenticação Extensível
Stopped     fdPHost        Host de Provedor da Descoberta de F...
Running     LanmanServer   Server
Stopped     msiserver      Windows Installer
Stopped     PerfHost       Host de DLL de Contador de Desempenho
Stopped     smphost        SMP de Espaços de Armazenamento da ...
```

Stopped	upnphost	Host de dispositivo UPnP
Running	WdiServiceHost	Host do Serviço de Diagnóstico
Stopped	WdiSystemHost	Host do Sistema de Diagnósticos

Embora o uso do operador `-Match` dentro de outros comandos seja mais útil e divertido, nos exemplos seguintes usarei sua forma mais básica para poder demonstrar seus detalhes.

Sistemas Windows são “case-insensitive”, ou seja, tanto faz usar letras maiúsculas ou minúsculas, elas são consideradas iguais. Assim, WIN.TXT, win.txt e Win.Txt são maneiras válidas de se referir a um mesmo arquivo. Este comportamento se repete nas ferramentas do sistema, então lembre-se disso ao usar o `-Match`.

```
PS> 'PowerShell' -Match '^power'
True
PS> 'PowerShell' -Match '^POWER'
True
PS> 'PowerShell' -Match '^[a-z]'
True
PS>
```

Isso nem sempre é desejado, então há uma versão alternativa deste operador, chamada `-cMatch`, cuja única diferença é desligar esse comportamento. Use-o sempre que quiser fazer um casamento mais exato, fazendo a distinção entre letras maiúsculas e minúsculas.

```
PS> 'PowerShell' -cMatch '^power'
False
PS> 'PowerShell' -cMatch '^POWER'
False
PS> 'PowerShell' -cMatch '^Power'
True
PS>
```

Outra maneira de escolher como casar maiúsculas/minúsculas é colocar dentro da própria expressão, já no início, o modificador `(?i)` para casar maiúsculas e minúsculas, ou o modificador `(?-i)` para casar somente as letras especificadas na expressão. Estes modificadores têm prioridade, então tanto faz usá-los com `-Match` ou `-cMatch`.

```
PS> 'PowerShell' -cMatch '(?i)^POWER'  
True  
PS> 'PowerShell' -Match '(?-i)^POWER'  
False  
PS>
```

Às vezes, você precisa do contrário: saber quando a expressão *não* casa com o texto. Para esse tipo de casamento negado, há o operador `-NotMatch`. Você também pode usar a negação lógica padrão do PowerShell, com `-Not` ou `!` e o uso de parênteses ao redor no casamento normal.

```
PS> 'PowerShell' -Match '^Win'  
False  
PS> 'PowerShell' -NotMatch '^Win'  
True  
PS> -Not ('PowerShell' -Match '^Win')  
True  
PS> !('PowerShell' -Match '^Win')  
True  
PS>
```

Felizmente, a acentuação não é um problema. Tudo funciona normalmente, conforme esperado. A equivalência entre letras acentuadas maiúsculas e minúsculas é respeitada e o metacaractere `\w` também casa letras acentuadas. Por exemplo, a expressão `\w+` pode ser usada para casar as palavras de um texto em português.

```
PS> 'PówêrShéll' -Match '^PÓWÊR'  
True  
PS> 'PówêrShéll' -Match '^\\w+$'  
True  
PS>
```

Percebeu que todas as expressões regulares dos exemplos foram colocadas entre aspas ‘simples’? Assim elas ficam protegidas. Não use aspas “duplas”, pois senão o PowerShell tentará expandir variáveis e outras estruturas ali dentro, bagunçando sua expressão.

Toda vez que você faz um casamento com expressões regulares no PowerShell, o trecho de texto casado é guardado em um dicionário

(HashTable) chamado `$matches`. Basta acessar a chave zero do dicionário para obter o texto casado.

```
PS> 'PowerShell' -Match '^.....'
True
PS> $matches[0]
Power
PS> $matches
Name      Value
-----
0         Power
PS>
```

Se sua expressão tiver um ou mais grupos, os trechos de texto casados por cada um desses grupos serão guardados em novas chaves numéricas dentro do dicionário `$matches`, sempre respeitando a contagem dos grupos: a chave um é o conteúdo do primeiro grupo, a chave dois é do segundo, e assim por diante.

```
PS> 'PowerShell' -Match '^(.)(.)(.)(.)(.)'
True
PS> $matches[0]
Power
PS> $matches[1]
P
PS> $matches[2]
o
PS> $matches
Name      Value
-----
5         r
4         e
3         w
2         o
1         P
0         Power
PS>
```

Para deixar o resultado mais amigável, você também pode usar grupos nomeados na expressão, assim as chaves do dicionário `$matches` usarão estes

nomes em vez de números. Você pode inclusive usar nomes com acentuação, que funciona normalmente.

```
PS> 'PowerShell' -Match '^(?<primeira>.....)(?<última>.....)$'
True
PS> $matches['última']
Shell
PS> $matches.última
Shell
PS> $matches
Name      Value
----      -
última    Shell
primeira  Power
0         PowerShell
PS>
```

Uma pegadinha importante: o dicionário `$matches` é atualizado somente quando o casamento é bem-sucedido. Quando o casamento falhar, o dicionário não será resetado, ele permanecerá com seu valor anterior. Veja no próximo exemplo como, mesmo após três casamentos falhos, o `$matches` continua com seu conteúdo original, do primeiro casamento.

```
PS> 'PowerShell' -Match '^Power'
True
PS> $matches
Name  Value
----  -
0     Power
PS> 'PowerShell' -Match '^DOS'
False
PS> 'PowerShell' -Match '^Windows'
False
PS> 'PowerShell' -Match '^Bill'
False
PS> $matches
Name  Value
----  -
0     Power
PS>
```

Por causa disso, não confie cegamente no conteúdo deste dicionário. Antes de acessá-lo, sempre confira se o casamento foi de fato bem-sucedido, por exemplo, com o comando `if`.

```
PS> if ('PowerShell' -Match '^Power') { 'Casei: ' + $matches[0] }
Casei: Power
PS> if ('PowerShell' -Match '^Bill' ) { 'Casei: ' + $matches[0] }
PS>
```

Uma limitação do operador `-Match` é que não há como fazer um casamento global, ou seja, encontrar todas as ocorrências possíveis dentro de um texto. Ele sempre casa somente a primeira ocorrência que encontrar. Por exemplo, ao tentar casar todas as palavras de um texto com a expressão `\w+`, somente a primeira palavra será guardada em `$matches`.

```
PS> $texto = 'Sabe o PowerShell? É legal!'
PS> $regex = '\w+'
PS> $texto -Match $regex
True
PS> $matches
Name      Value
----      -
0         Sabe
PS>
```

Para fazer uma pesquisa global, é preciso recorrer a algo mais elaborado: use o comando `Select-String` com a opção `-AllMatches` para fazer o casamento e em seguida extraia o atributo `Matches.Value` de cada resultado. O resultado final será um array de strings com todas as ocorrências.

```
PS> $texto = 'Sabe o PowerShell? É legal!'
PS> $regex = '\w+'
PS> $texto | Select-String -AllMatches $regex | foreach { $_.Matches
Sabe
0
PowerShell
É
legal
PS>
```

Para ficar mais conveniente o uso, você pode salvar estes comandos em um filtro personalizado. Podemos batizá-lo de `Get-Matches` para seguir o estilo Verbo-Substantivo, que é o padrão de nomenclatura dos comandos no PowerShell. Basta digitar o seguinte na linha de comando:

```
filter Get-Matches($regex) {  
    $_ | Select-String -AllMatches $regex | foreach { $_.Matches.Val  
}
```

E pronto. Agora o comando `Get-Matches` está disponível para uso. Ele espera receber o texto via entrada padrão e aceita como argumento a expressão regular a ser casada. O resultado será um array com todas as ocorrências encontradas, ou nulo se a expressão não casar.

```
PS> 'Sabe o PowerShell? É legal!' | Get-Matches('\w+')  
Sabe  
o  
PowerShell  
É  
legal  
PS>
```

O mesmo comando `Select-String` recém-utilizado no filtro também serve para fazer um “grep”, ou seja, extrair de um arquivo todas as linhas que casam com a expressão regular informada. Por exemplo, obter todas as linhas do Hino Nacional que terminam em “orte”, seguido de pontuação:

```
PS> Get-Content hino-nacional.txt | Select-String 'orte[,.!]$'  
Conseguimos conquistar com braço forte,  
Desafia o nosso peito a própria morte!  
Mas, se ergues da justiça a clava forte,  
Nem teme, quem te adora, a própria morte.  
PS>
```

Você já deve conhecer o comando `switch`, que pode fazer vários testes diferentes em um mesmo texto. Mas você sabia que ele também aceita expressões regulares ao usar a opção `-Regex`? Veja um exemplo que guarda em variáveis distintas o dia, o mês e o ano de datas nos formatos brasileiro (dd/mm/aaaa) e internacional (aaaa-mm-dd). Perceba o uso do dicionário

\$matches para obter o conteúdo de cada grupo da expressão.

```
switch -Regex ($data) {  
    '^(..)/(..)/(....)$'    # Formato dd/mm/aaaa  
    {  
        $ano = $matches[3]  
        $mes = $matches[2]  
        $dia = $matches[1]  
    }  
    '^(..)-(..)-(..)$'    # Formato aaaa-mm-dd  
    {  
        $ano = $matches[1]  
        $mes = $matches[2]  
        $dia = $matches[3]  
    }  
}
```

Para fazer substituições, use o operador -Replace, informando a expressão regular e o texto substituto. O comportamento é sempre global, ou seja, troca todas as ocorrências encontradas. Não há como limitar o número de substituições a serem feitas.

```
PS> 'PowerShell' -Replace '^Power', 'Windows'  
WindowsShell  
PS> 'PowerShell' -Replace '[a-z]', '.'  
.....  
PS>
```

Assim como no operador -Match, o modo de operação padrão do -Replace é ser “case-insensitive”. Para desligar este comportamento e manter a distinção entre letras maiúsculas e minúsculas, use a sua versão alternativa -cReplace ou insira o modificador (?-i) no início de sua expressão.

```
PS> 'PowerShell' -cReplace '[a-z]', '.'  
P....S....  
PS> 'PowerShell' -Replace ' (?-i)[a-z]', '.'  
P....S....  
PS>
```

É possível encadear várias substituições em um mesmo comando, que elas serão aplicadas em sequência: o texto original é modificado pela primeira



substituição, o resultado então é modificado pela segunda substituição, e assim sucessivamente.

```
PS> 'Power' -Replace '^', 'PP'
PPPower
PS> 'Power' -Replace '^', 'PP' -Replace 'P', 'P-'
P-P-P-ower
PS> 'Power' -Replace '^', 'PP' -Replace 'P', 'P-' -Replace '$', '!'
P-P-P-ower!
PS>
```

Os retrovisores são referenciados como \$1, \$2, e assim por diante. Há também o retrovisor especial \$0 que guarda todo o trecho de texto casado pela expressão. Se precisar grudar algum número logo após o retrovisor, use a notação com chaves: \${1}, \${2} etc. Lembre-se de usar aspas ‘simples’ ao redor do texto substituto, senão o PowerShell interpretará os retrovisores como variáveis normais.

```
PS> 'PowerShell' -cReplace '^(P.*)(S.*)', '$1 + $2 = $0'
Power + Shell = PowerShell
PS> '1 2 3 4' -Replace '(\d)', '${1}00'
100 200 300 400
PS>
```

Se precisar inserir caracteres especiais no texto substituto, como Tabs e quebras de linha, use `t e `n, respectivamente. Porém estes escapes só funcionam dentro de aspas “duplas”. Isso irá requerer um cuidado especial quando também houver retrovisores: os seus cifrões deverão ser escapados com `\$.

```
PS> '1 2 3 4' -Replace '\s+', "`t"
1      2      3      4
PS> 'PowerShell' -cReplace '^(P.*)(S.*)', "`$1`n`$2"
Power
Shell
PS>
```

Para dividir um texto usando expressões regulares, use o operador -split, que retorna um array de strings. Um segundo argumento opcional pode ser informado para limitar o tamanho do array. Neste caso, o último item do

array trará o texto restante, que ficou sem corte.

```
PS> 'um;dois,três;quatro,cinco' -Split '[,;]'
um
dois
três
quatro
cinco
PS> 'um;dois,três;quatro,cinco' -Split '[,;]', 3
um
dois
três;quatro,cinco
PS>
```

Assim como os outros operadores já vistos, o `-Split` é “case-insensitive” por padrão. Já sabe né? Use `-csplit` ou o modificador `(?-i)` para desligar este comportamento e manter a distinção entre letras maiúsculas e minúsculas.

Mais informações são encontradas em:

<http://aurelio.net/regex/powershell/>

## Python

Característica	Como fazer
Busca	Funções <code>re.search</code> , <code>re.findall</code> , <code>re.finditer</code>
Substituição	Função <code>re.sub</code>
Divisão	Função <code>re.split</code>
ER crua	<code>r'raw string'</code>
Ignore M/m	Modificadores <code>(?i)</code> , <code>re.I</code>
Global	É o padrão

Python possui um dos mais completos suportes às expressões regulares, com objetos e métodos já prontos para obter diversas informações sobre os casamentos. O primeiro passo é carregar o módulo `re`, responsável pelo tratamento das expressões:

```
import re
```

Antes de começar, uma dica muito importante: sempre coloque suas expressões dentro de “raw strings” (r'...') para torná-las cruas, evitando assim os infundáveis problemas com escapes. Acostume-se a sempre usar esta notação, mesmo quando sua expressão for simples e não possuir nenhuma contrabarra.

Para testar se uma expressão casou ou não em determinado texto (ou variável), use a função `search`.

```
if re.search(r'^Py', 'Python'):
    print 'Casou'
else:
    print 'Não casou'
```

Além de somente testar se casou ou não, esta função também retorna um objeto com informações sobre o casamento. Guarde o resultado em uma variável para poder acessá-lo depois. É prática comum na comunidade Python chamar esta variável de `m`, uma abreviação para `match`.

```
m = re.search(r'^Py', 'Python'):
if m:
    print 'Casou'
else:
    print 'Não casou'
```

Com o resultado guardado, agora podemos usar seus métodos para obter informações úteis, como o trecho casado e os índices.

```
m = re.search(r'^Py', 'Python')
if m:
    print m.group()    # Py
    print m.start()    # 0
    print m.end()      # 2
    print m.span()     # (0, 2)
```

O método `group` traz o trecho de texto casado pela expressão. Com os métodos `start` e `end`, você obtém a posição de início e fim do trecho casado na string original. O método `span` é similar, porém já traz ambas as posições dentro de uma tupla. Lembre-se de que em Python os índices iniciam em

zero.

Se sua expressão contém grupos, além das informações do casamento como um todo (considerado grupo zero), você também pode obter informações sobre cada um dos grupos, informando seu número. Há também o método `groups` que retorna uma tupla com o conteúdo casado de todos os grupos.

```
m = re.search(r'(..)/(..)/(....)', '31/12/1999')
if m:
    print m.group(0)    # 31/12/1999
    print m.group(1)    # 31
    print m.group(2)    # 12
    print m.group(3)    # 1999
    print m.span(0)     # (0, 10)
    print m.span(1)     # (0, 2)
    print m.span(2)     # (3, 5)
    print m.span(3)     # (6, 10)
    print m.groups()    # ('31', '12', '1999')
```

E se a expressão casar mais de uma vez no texto? Para encontrar todas as ocorrências, use a função `findall`. Ela retorna uma lista com todos os trechos de texto casados pela expressão, ou uma lista vazia, se não casar.

```
texto = "Corri 3km em 15 minutos, ouvindo CPM 22."
print re.findall(r'\d+', texto)    # ['3', '15', '22']
print re.findall(r'XXX', texto)    # []
```

Se houver grupos na expressão, o retorno da função será uma lista de tuplas. Cada tupla representa um casamento, trazendo o conteúdo de todos os seus grupos.

```
texto = "Acordei às 08:00, comi 12:30, dormi às 23:59."
print re.findall(r'(\d\d):(\d\d)', texto)
# Resultado:
# [('08', '00'), ('12', '30'), ('23', '59')]
```

Uma maneira mais sofisticada de se lidar com múltiplas ocorrências é fazer um loop nos casamentos, usando a função `finditer`. Você pode inspecionar cada ocorrência, usando aqueles métodos bacanas que já vimos anteriormente, como `group` e `span`.

```

texto = "Acordei às 08:00, comi 12:30, dormi às 23:59."
for m in re.finditer(r'(\d\d):(\d\d)', texto):
    hora = m.group(1)
    min = m.group(2)
    print "%s horas, %s minutos." % (hora, min)
# Resultado:
# 08 horas, 00 minutos.
# 12 horas, 30 minutos.
# 23 horas, 59 minutos.

```

Um método útil de se utilizar dentro de loops é o `expand`, que funciona de maneira similar à substituição, expandindo os retrovisores (`\1`, `\2`, ...). Não se esqueça de usar “raw string”! Veja como o exemplo anterior fica mais simples:

```

texto = "Acordei às 08:00, comi 12:30, dormi às 23:59."
for m in re.finditer(r'(\d\d):(\d\d)', texto):
    print m.expand(r'\1 horas, \2 minutos.')

```

A substituição é feita pela função `sub`, que troca todas as ocorrências encontradas. Ela aceita um quarto argumento opcional para limitar o número de substituições a serem feitas:

```

print re.sub(r'\w', '.', 'Python')      # .....
print re.sub(r'\w', '.', 'Python', 2)   # ..thon

```

Os retrovisores são referenciados normalmente, usando a contrabarra. Por isso, lembre-se de também colocar o texto substituto dentro de uma “raw string”, para evitar problemas de escape.

```

print re.sub(r'(Py).*', r'\1\1', 'Python')  # PyPy

```

Uma sintaxe alternativa para os retrovisores é `\g<1>`, `\g<2>`, útil quando você precisar usar um número literal logo após o retrovisor.

```

print re.sub(r'(Py).*', r'\13', 'Python')   # erro
print re.sub(r'(Py).*', r'\g<1>3', 'Python') # Py3

```

Para substituições realmente estilosas, você pode usar uma função no lugar do texto substituto. Esta função receberá uma instância de `MatchObject` para cada ocorrência e deve retornar uma string.

```

def data_por_extenso(m):

```

```

dia = m.group(1)
mes = m.group(2)
ano = m.group(3)
meses = {
    '01': 'Jan', '02': 'Fev', '03': 'Mar',
    '04': 'Abr', '05': 'Mai', '06': 'Jun',
    '07': 'Jul', '08': 'Ago', '09': 'Set',
    '10': 'Out', '11': 'Nov', '12': 'Dez'
}
return dia + " de " + meses[mes] + " de " + ano
texto = "Hoje é dia 31/12/1999."
regex = r'(\d\d)/(\d\d)/(\d\d\d\d)'
print re.sub(regex, data_por_extenso, texto)
# Hoje é dia 31 de Dez de 1999.

```

Para dividir um texto usando expressões regulares, use a função `split`, que retorna uma lista de strings. Um terceiro argumento opcional pode ser informado para limitar o número de vezes que o texto vai ser dividido. Neste caso, o último item da lista trará o texto restante, que ficou sem corte.

```

print re.split(r'[/.]', '31/12/99')      # ['31', '12', '99']
print re.split(r'[/.]', '31/12/99', 1)   # ['31', '12/99']

```

Se você for utilizar a mesma expressão mais de uma vez, é possível compilá-la para garantir uma execução mais rápida. O objeto retornado da compilação tem os mesmos métodos do módulo `re`, então você pode casar, substituir e dividir textos usando expressões regulares compiladas.

```

# Primeiro compile as expressões
er_hora = re.compile(r'(\d\d):(\d\d)')
er_separador = re.compile(r'[/.:]')
# Agora pode usá-las diretamente para pesquisar
if er_hora.search('23:59'):
    print 'Casou'
# Para substituir
print er_hora.sub(r'\1h\2min', '23:59') # 23h59min
# E para dividir
print er_separador.split('1.23')         # ['1', '23']
print er_separador.split('23:59')       # ['23', '59']
print er_separador.split('31/12/99')    # ['31', '12', '99']

```

# Flags

Para ignorar a diferença entre maiúsculas e minúsculas, você precisa adicionar uma flag à expressão. Há duas maneiras de se fazer isso: usar o grupo moderno `(?i)` no início da expressão, ou passar as constantes `re.I` ou `re.IGNORECASE` às funções.

```
if re.search(r'(?i)^py', 'Python'):
    print 'Casou'
if re.search(r'^py', 'Python', re.IGNORECASE):
    print 'Casou'
```

Não recomendo o uso de constantes pois ele é específico do Python e nem todas as funções o suportam. Prefira sempre a primeira forma `(?i)`, pois, além de manter tudo dentro da própria expressão, esse formato é padrão e funciona em outras linguagens.

Sempre coloque este grupo especial bem no início da expressão para evitar problemas, e se você precisar usar mais de uma flag ao mesmo tempo, basta juntar as letras dentro do mesmo grupo: `(?iux)`.

É com flags que resolvemos problemas de acentuação também. O Python não tem classes POSIX (como `[:alpha:]` e amigos), mas possui o barra-letra `\w`, que casará letras acentuadas se os planetas estiverem bem alinhados :)

Tudo depende da configuração de seu sistema, da versão do Python e da codificação do texto original. Há duas flags que você pode usar: `(?L)` para levar em conta a localização de seu sistema, e `(?u)` para levar em conta a tabela Unicode.

Sugiro testar as combinações e ver qual funciona para o seu ambiente. Para o futuro, é melhor sempre manipular o texto como Unicode e usar a flag `(?u)`. Estes são os resultados em meu sistema atual (Python 2.7.1, `LANG=pt_BR.UTF-8`):

```
print re.sub(r'\w', '.', 'Páiton')      # .á....
print re.sub(r'(?L)\w', '.', 'Páiton')  # .á....
print re.sub(r'(?u)\w', '.', 'Páiton')   # ..?....
print re.sub(r'\w', '.', u'Páiton')      # .á....
```

```
print re.sub(r'(?L)\w', '.', u'Páiton') # .á....
print re.sub(r'(?u)\w', '.', u'Páiton') # ..... OK
```

Há duas flags especiais para lidar com strings de múltiplas linhas. Use `(?s)` para fazer o metacaractere ponto casar o `\n`, o que normalmente não acontece. Assim, o seu curinga `.` vai casar a string toda. Use `(?m)` para fazer as âncoras `^` e `$` casarem cada uma das linhas da string.

```
# Flag (?s) para o . casar o \n
re.sub(r'.*', '.', '1\n2\n3') # .\n.\n.
re.sub(r'(?s).*', '.', '1\n2\n3') # .
# Flag (?m) para usar ^ e $ em todas as linhas
re.sub(r'^2', '.', '1\n2\n3') # 1\n2\n3
re.sub(r'(?m)^2', '.', '1\n2\n3') # 1\n.\n3
```

E caso sua expressão fique realmente grande e complexa, use a flag `(?x)`, que permite que você organize sua expressão em várias linhas, com alinhamento e comentários. Os espaços em branco e Tabs são ignorados e o caractere `#` é usado para iniciar um comentário. Para inserir espaços literais, você deve escapá-los ou usar `\s`.

```
# Números de telefone no formato internacional
texto = '''
+554798765432
+55 47 98765432
+55 47 9876-5432
+55 11 98765-4321
'''

# Expressão para casar números de telefone
# Nota: São duas flags no início: 'x' e 'm'
regex = r'''(?xm)
^          # Início da linha
\s*       # espaços opcionais
\+55      # Código do Brasil: +55
\s?      # espaço opcional
[0-9]{2}  # Código de área (DDD)
\s?      # espaço opcional
9?       # Dígito 9 adicional para São Paulo
[0-9]{4}  # Quatro primeiros dígitos
-?       # Separador opcional
```



```

    [0-9]{4}    # Últimos quatro dígitos
    $          # Fim da linha
'''
print re.sub(regex, 'casou', texto)
# Resultado:
# casou
# casou
# casou
# casou

```

## Grupos nomeados

Em Python você também pode dar nomes aos grupos de sua expressão. Assim, além dos números, você também pode referenciar esses grupos usando o seu nome.

O formato usado para dar um nome ao grupo é feito que dói: (?P<nome>...). Por exemplo, a versão simplificada da expressão para casar uma data no formato brasileiro (...)/(...)/(....), colocando nomes nos grupos, fica assim:

```
(?P<dia>...)/(?P<mes>...)/(?P<ano>....)
```

Para usar o retrovisor em grupos normais, você usa \1 tanto dentro da expressão quanto na substituição. Já para grupos nomeados, você deve usar (?P=nome) dentro da expressão e \g<nome> na substituição.

```

# Converter datas de DD/MM/AAAA para AAAA-MM-DD
data = '31/12/1999'
# Substituição usando grupos normais
print re.sub(
    r'(..)/(..)/(....)',
    r'\3-\2-\1',
    data)
# Substituição usando grupos nomeados
print re.sub(
    r'(?P<dia>...)/(?P<mes>...)/(?P<ano>....)',
    r'\g<ano>-\g<mes>-\g<dia>',
    data)
# Resultado:

```

```
# 1999-12-31
# 1999-12-31
```

Os métodos de `MatchObject` aceitam o nome ou o número do grupo, tanto faz. Perceba que, enquanto o método `groups` retorna um tupla com todos os grupos, o método `groupdict` retorna um dicionário composto unicamente com os grupos nomeados.

```
m = re.search(r'(?P<dia>..)/(?P<mes>..)(....)', '31/12/1999')
if m:
    print m.group(0)      # 31/12/1999
    print m.group(1)      # 31
    print m.group(2)      # 12
    print m.group(3)      # 1999
    print m.group('dia')  # 31
    print m.group('mes')  # 12
    print m.groups()      # ('31', '12', '1999')
    print m.groupdict()   # {'dia': '31', 'mes': '12'}
```

Mais informações são encontradas em:

<http://aurelio.net/regex/python/>

## Ruby

Característica	Como fazer
Busca	Método <code>match</code> , operador <code>=~</code>
Substituição	Métodos <code>sub</code> , <code>gsub</code>
Divisão	Método <code>split</code>
ER crua	/entre barras/
Ignore M/m	Modificadores <code>i</code> e <code>(?i)</code>
Global	Método <code>gsub</code>

Em Ruby, as expressões regulares são parte integrante da linguagem, como um tipo de dado adicional. Para definir uma string, basta colocar um texto entre aspas. De maneira similar, para definir uma expressão regular, basta

colocá-la entre barras. Assim não será feita a interpolação de strings e os escapes serão respeitados.

Uma vez definida a expressão, seu casamento com um texto qualquer poderá ser feito com o método `match`, ou diretamente pelo operador `=~`.

```
if /^Ru/.match("Ruby") then
  puts "Casou"
end
if "Ruby" =~ /^Ru/ then
  puts "Casou"
end
```

Para ignorar a diferença entre maiúsculas e minúsculas, era comum definir a variável `$=`, mas essa prática atualmente é considerada obsoleta. O recomendado é colocar o modificador `i` logo após a segunda barra ou usar o metacaractere modernoso `(?i)` no início da expressão.

```
if "Ruby" =~ /^ru/i then
  puts "Casou"
end
if "Ruby" =~ /(?i)^ru/ then
  puts "Casou"
end
```

Além de testes simples se casou ou não, o método `match` pode ser usado para guardar informações sobre o casamento. Seu retorno é um array cujo primeiro item é o trecho casado pela expressão, seguido de itens que guardam o conteúdo de todos os grupos marcados.

```
resultado = /^(.)(.)(.)/.match("Ruby")
puts resultado[0]  # Rub
puts resultado[1]  # R
puts resultado[2]  # u
puts resultado[3]  # b
```

A substituição de textos é feita pelo método `sub`, presente no objeto `String`. Informe a expressão como primeiro argumento e lembre-se de que apenas a primeira ocorrência é trocada. Se desejar uma substituição global, em que todas as ocorrências são trocadas, use o método `gsub`. O uso dos

modificadores `i` e `(?i)` para maiúsculas e minúsculas continua valendo.

```
puts "Ruby".sub( /[a-z]/ , '.' )    # R.by
puts "Ruby".gsub( /[a-z]/ , '.' )    # R...
puts "Ruby".gsub( /[a-z]/i , '.' )    # ....
```

Ao utilizar retrovisores, coloque todo o texto substituto entre aspas simples, para evitar que os escapes sejam mal interpretados. O retrovisor especial `\0` guarda todo o trecho casado pela expressão.

```
puts "Ruby".sub( /(..).. / , '\1\1' )    # RuRu
puts "Ruby".sub( /.... / , '\0\0' )    # RubyRuby
```

Acentuação tem suas pegadinhas. O barra-letra `\w` não inclui letras acentuadas, nem adianta perder seu tempo. Já as classes POSIX incluem acentuação, porém somente à partir do Ruby versão 1.9. Para uma solução geral, que funciona em todas as versões da linguagem, use o remendo `[À-ü]` para casar letras acentuadas.

```
puts "Rubí".gsub( /\w/ , '.' )    # ...í
puts "Rubí".gsub( /\w/u , '.' )    # ...í
puts "Rubí".gsub( /[:alpha:]/ , '.' )    # ....
puts "Rubí".gsub( /\wÀ-ü/ , '.' )    # ....
```

Para dividir uma string, basta usar o método `split`, informando a expressão regular como argumento. Se você informar um segundo argumento numérico, ele será usado para limitar o tamanho do array resultante. Neste caso, o último item será o texto restante não dividido.

```
puts "R u b y".split( /\s+/ )
# R
# u
# b
# y
puts "R u b y".split( /\s+/, 3 )
# R
# u
# b y
```

Mais informações são encontradas em:

<http://aurelio.net/regex/ruby/>

# Sed

Característica	Como fazer
Busca	/endereço/
Substituição	Comando s///
Divisão	-
ER crua	/entre barras/
Ignore M/m	Modificador I
Global	Modificador g

O Sed é um editor de textos não interativo orientado à linha. Você programa as regras e ele as aplica em um texto, fazendo as alterações necessárias automaticamente. Originado em 1974 no Unix, hoje se encontra disponível na maioria dos sistemas modernos.

Atualmente, as duas versões mais usadas são a do Unix (utilizada nos BSDs e no Mac) e a da GNU (utilizada no Linux e no Windows/Cygwin). A versão GNU também é chamada de gsed e possui um suporte mais moderno e poderoso às expressões regulares.

O Sed sempre manipula linhas inteiras. Você passa um arquivo ou um texto pela entrada padrão (STDIN) e ele o lê linha a linha, aplicando as regras definidas pelo programador. Você pode casar linhas e aplicar comandos nelas.

Para casar linhas, basta colocar a expressão regular entre barras. Em seguida, insere-se o comando a ser executado, como o `d` (delete) ou o `p` (print). Se sua expressão possuir alguma barra `/`, lembre-se de escapá-la `\`.

```
/[0-9]/ d    # Deleta as linhas que possuem números  
/[0-9]/ p    # Duplica as linhas que possuem números
```

Para que o casamento seja feito ignorando-se a diferença entre maiúsculas e minúsculas, adicione o modificador `I` logo após a segunda barra. Note que é a letra `i` maiúscula.

```
/sed/I d    # Apaga linhas com sed, Sed, sEd, SED, sED, ...
```

O sed do Unix não entende esse modificador, então a única maneira de casar maiúsculas e minúsculas é sempre especificar todas elas.

```
/[Ss][Ee][Dd]/ d
```

Com o uso na linha de comando, é preciso ter o cuidado de colocar todo o comando dentro de aspas simples, para evitar problemas com a interpretação de caracteres especiais pelo shell.

```
prompt$ echo Sed | sed '/[Ss][Ee][Dd]/ p'
Sed
Sed
```

O comando de substituição é o s. Sua sintaxe é a mesma utilizada pelo ed, Vim e Perl: s/expressão/texto/. As barras separam a expressão regular do texto substituto. Ainda é possível colocar alguns modificadores após a última barra.

O comportamento padrão é trocar somente a primeira ocorrência, mas colocando-se um número após a última barra, ele indicará qual a ocorrência que será trocada. Se colocar o modificador g, então todas as ocorrências serão trocadas. O modificador I (exclusivo do gsed) ignora a diferença entre maiúsculas e minúsculas.

```
prompt$ echo Sed | sed 's/[A-Za-z]/./'      # .ed
prompt$ echo Sed | sed 's/[A-Za-z]/./2'     # S.d
prompt$ echo Sed | sed 's/[A-Za-z]/./3'     # Se.
prompt$ echo Sed | sed 's/[A-Za-z]/./g'     # ...
prompt$ echo Sed | sed 's/[a-z]/./gI'       # ...
```

Se você precisar colocar barras dentro do comando, lembre-se de escapá-las. Uma alternativa mais elegante é utilizar outro caractere como delimitador do comando em vez das barras, como % ou @.

```
prompt$ echo /etc/passwd | sed 's\/etc\/tmp/'
/tmp/passwd
prompt$ echo /etc/passwd | sed 's@/etc@/tmp@'
/tmp/passwd
prompt$ echo /etc/passwd | sed 's_/etc_/tmp_'
/tmp/passwd
```

Use retrovisores normalmente como \1, \2 e assim por diante. O caractere especial & representa todo o trecho casado pela expressão regular.

```
prompt$ echo Sed | sed 's/.*/&&/'
SedSedSed
prompt$ echo Sed | sed 's/\(.\)\(.\)\(.\)/\3\2\1/'
deS
```

O padrão do Sed é utilizar a notação antiga das expressões, em que é preciso escapar alguns caracteres para que sejam considerados metacaracteres, como os grupos \(.\) do exemplo anterior. Versões mais recentes trazem uma opção nova (-r no gsed, -E no BSD/Mac) para você poder usar os metacaracteres modernos, sem a necessidade de escapes. Veja a diferença:

```
sed 's/^\([0-9]\{1,3\}\)\.+/ \1/'
sed -r 's/^\([0-9]{1,3}\)\.+/ \1/'
```

A única desvantagem é que não vai funcionar em versões antigas do Sed. Mas se isso não é problema para você, use sempre essa opção, para que suas expressões fiquem mais simples e fáceis de ler. Segue uma tabelinha para não confundir mais:

Comando	Metacaracteres									
sed	.	^	\$	*	\+	\?	\	[ ]	\{ \}	\( \)
sed -r/-E	.	^	\$	*	+	?		[ ]	{ }	( )

A acentuação é possível casar com as classes POSIX. No gsed, também é possível usar o barra-letra \w. Apenas se certifique de que seu sistema está corretamente configurado para o português, conferindo as variáveis de ambiente \$LANG e \$LC\_ALL.

```
prompt$ echo séd | sed 's/[a-z]/./g'          # .é.
prompt$ echo séd | sed 's/[[:alpha:]]/./g'     # ...
prompt$ echo séd | sed 's/\w/./g'             # ...
```

Se você estiver utilizando o GNU sed e quiser testar a compatibilidade de seus scripts com outras versões do editor, use a opção --posix, que desliga todas as características adicionais.

Mais informações são encontradas em:

<http://aurelio.net/regex/sed/>

## Shell Script (Bash)

Característica	Como fazer
Busca	Operador =~
Substituição	-
Divisão	-
ER crua	É o padrão
Ignore M/m	Opção nocasematch
Global	-

Bash é o shell padrão da maioria dos sistemas Linux e do Mac. O suporte às expressões regulares foi incluído em sua versão 3.0 de 2004. Em versões anteriores, era preciso utilizar o grep ou outra ferramenta externa, mas agora o Bash conta com o operador =~ para casar expressões.

```
prompt$ [[ 'Bash' =~ B[a-z]{3} ]] && echo Casou  
Casou
```

Percebeu que não usei aspas ao redor da expressão? É necessário que seja sempre assim, pois, se colocar aspas, sua expressão será considerada um texto normal, composto apenas de caracteres literais, e o casamento falhará.

```
prompt$ [[ 'Bash' =~ "B[a-z]{3}" ]] && echo Casou || echo Falhou  
Falhou
```

Como não pode usar aspas, você deve tomar o cuidado de escapar todos os espaços em branco e caracteres especiais do shell, como a crase ` e o cifrão \$.

```
prompt$ [[ 'B ash' =~ B\[a-z\]{3} ]] && echo Casou || echo Falhou  
Casou
```

É um tédio ficar escapando caracteres. Uma alternativa é guardar sua expressão dentro de uma variável. Ao definir a variável você pode usar as aspas normalmente, evitando preocupar-se com escapes desnecessários.



```
prompt$ regex='B [a-z]{3}'
prompt$ [[ 'B ash' =~ $regex ]] && echo Casou || echo Falhou
Casou
```

Para casar uma expressão ignorando a diferença entre maiúsculas e minúsculas, ligue a opção `nocase`. Trata-se de uma chave que, enquanto estiver ligada, valerá para todos os testes efetuados. Use `shopt -s` para ligá-la e `shopt -u` para desligá-la.

```
prompt$ [[ 'Bash' =~ ^b ]] && echo Casou || echo Falhou
Falhou
prompt$ shopt -s nocase
prompt$ [[ 'Bash' =~ ^b ]] && echo Casou || echo Falhou
Casou
```

Toda vez que o operador `=~` é utilizado, o trecho casado pela expressão regular é guardado no array `$BASH_REMATCH`. Se a expressão não casar, o array ficará vazio.

```
prompt$ [[ 'Bash' =~ ^. ] ] ; echo $BASH_REMATCH
B
prompt$ [[ 'Bash' =~ ^... ] ] ; echo $BASH_REMATCH
Bas
prompt$ [[ 'Bash' =~ ^x ] ] ; echo $BASH_REMATCH
prompt$
```

Se sua expressão possuir um ou mais grupos, os trechos casados por cada um desses grupos serão guardados em novas posições dentro do array `$BASH_REMATCH`, sempre respeitando a contagem dos grupos: o índice um é o conteúdo do primeiro grupo, o dois, do segundo grupo, e assim por diante.

```
prompt$ [[ 'Bash' =~ (.) (.) (.) (.) ] ]
prompt$ echo ${BASH_REMATCH[0]}
Bash
prompt$ echo ${BASH_REMATCH[1]}
B
prompt$ echo ${BASH_REMATCH[2]}
a
prompt$ echo ${BASH_REMATCH[3]}
s
prompt$ echo ${BASH_REMATCH[4]}
```

h  
prompt\$

O suporte às expressões regulares no Bash ainda é limitado, não existindo comandos nativos para usá-las em substituições e divisões de textos. Essas tarefas ainda dependem de ferramentas externas do sistema.

Cuidado para não confundir expressões regulares com os curingas do shell (glob). Em ambos usamos símbolos como asterisco, interrogação e chaves, porém seu significado é diferente. Enquanto o glob `*.txt` representa todos os arquivos com a extensão `txt`, em expressões regulares o asterisco sozinho não diz nada, pois ele é um repetidor que precisa de algo antes, como em `\w*` ou `[0-9]*`. Veja esta tabelinha para não confundir mais:

### Equivalência entre glob e expressões regulares

Glob	Expressões regulares
<code>*</code>	<code>.*</code>
<code>?</code>	<code>.</code>
<code>{a,b}</code>	<code>(a b)</code>
<code>[abc]</code>	<code>[abc]</code>
<code>[^abc]</code>	<code>[^abc]</code>
<code>[0-9]</code>	<code>[0-9]</code>
<code>*.txt</code>	<code>.*\.txt</code>
<code>arquivo-?.txt</code>	<code>arquivo-.\.txt</code>
<code>arquivo.{txt html}</code>	<code>arquivo\.(txt html)</code>
<code>[Aa]rquivo.txt</code>	<code>[Aa]rquivo\.txt</code>

Mais informações são encontradas em:

<http://aurelio.net/regex/bash/>

## Tcl

--	--

Característica	Como fazer
Busca	Função regexp
Substituição	Função regsub
Divisão	Comando splitx
ER crua	{entre chaves}
Ignore M/m	Opção -nocase
Global	Opção -all

Na versão 8.1, o Tcl turbinou seu suporte às expressões regulares, modernizando os metacaracteres e adicionando novas opções aos comandos regexp e regsub. Se você está usando uma versão mais antiga, recomenda-se a atualização.

Regra sagrada: sempre coloque as expressões entre chaves. Assim, nenhum pré-processamento será feito e você não terá problemas com os escapes, pois sua expressão não será modificada pela linguagem. Para casar um texto, use o comando regexp.

```
if [regexp {^T} "Tcl"] {
    puts "Casou"
}
```

Se precisar casar um texto ignorando a diferença entre maiúsculas e minúsculas, há duas alternativas: use a opção -nocase ou coloque o metacaractere modernoso (?i) no início da expressão.

```
if [regexp -nocase {^t} "Tcl"] {
    puts "Casou"
}
if [regexp {(?i)^t} "Tcl"] {
    puts "Casou"
}
```

Ao utilizar grupos em sua expressão, use a opção -inline para que o comando retorne uma lista contendo todo o trecho casado, seguido pelo conteúdo de cada um dos grupos informados. Assim, fica fácil obter partes

específicas do texto casado.

```
set resultado [regexp -inline {(.) (.) (.)} "Tcl"]
lindex $resultado 0 ; # Tcl
lindex $resultado 1 ; # T
lindex $resultado 2 ; # c
lindex $resultado 3 ; # l
```

O comando para fazer substituições é o `regsub`. Seu comportamento padrão é trocar somente a primeira ocorrência encontrada. Para trocar todas as ocorrências de uma vez, use a opção `-all`.

```
puts [regsub {\w} "Tcl" "."] ; # .cl
puts [regsub -all {\w} "Tcl" "."] ; # ...
```

Ao utilizar retrovisores, uma boa dica é também colocar o texto substituto entre chaves, para evitar dores de cabeça com os escapes. O retrovisor `\0` e o caractere `&` podem ser usados para representar todo o trecho casado pela expressão.

```
puts [regsub {(T).*.} "Tcl" {\1\1\1}] ; # TTT
puts [regsub {(T).*.} "Tcl" {&&}] ; # TclTcl
```

A acentuação não é problema para o Tcl, desde que seu sistema esteja corretamente configurado para o português. Tanto as classes POSIX quanto o barra-letra `\w` respeitam a localização padrão.

```
puts [regsub -all {\w} "Têcêéle" "."] ; # .....
puts [regsub -all {[[:alpha:]]} "Têcêéle" "."] ; # .....
```

O comando `split` divide um texto, mas não sabe nada sobre expressões regulares. É preciso utilizar o comando `splitx`, presente no pacote `textutil` da `tcllib`. Quando cabível, uma alternativa é usar o `regex -inline` e inverter a lógica da divisão, informando uma expressão que case a parte desejada em vez do separador.

```
puts [regexp -inline -all {\d+} "31/12/99"] ; # 31 12 99
```

Mais informações são encontradas em:

<http://aurelio.net/regex/tcl/>

## VBscript

Característica	Como fazer
Busca	Método Test
Substituição	Método Replace
Divisão	Método Execute
ER crua	-
Ignore M/m	Propriedade IgnoreCase
Global	Propriedade Global

Em 1999, os usuários do Visual Basic Scripting Edition puderam experimentar o tão desejado recurso de expressões regulares, implementado na versão 5.0 do Microsoft Scripting Engines. O nome do objeto que trata das expressões é RegExp, e para o alívio dos programadores, a sintaxe e a funcionalidade das ERs no VBScript são similares ao Perl, então não precisa aprender uma linguagem nova.

Assim, se na sintaxe da ER não há novidade, podemos nos concentrar em como utilizar esse objeto. Dentro dele os métodos que temos são Test e Execute para busca e Replace para substituições. Para ignorar maiúsculas e minúsculas e fazer substituições globais, temos as duas propriedades booleanas IgnoreCase e Global, que devem ser definidas como true para entrarem em ação.

Como peculiaridade, a ER não pode ser passada diretamente ao método, ela precisa primeiro ser definida na propriedade Pattern. Vamos a um exemplo:

```
Dim er, texto
Set er = new regexp
texto = "visual"
er.Pattern = "^[A-Z]+$"
er.IgnoreCase = true
if er.Test(texto) then
    msgbox ("casou!")
end if
```

Então instanciamos o objeto er e definimos um texto qualquer a ser casado.

Depois definimos a ER, uma linha toda de maiúsculas, na propriedade Pattern e ligamos a opção de “ignorância”. Por fim, fazemos o Test da ER no texto. O Replace funciona de maneira similar, sem surpresas.

O método Execute funciona como um split ao contrário, em que você informa o padrão e não o separador. Ele casa a ER no texto e retorna todos os pedaços de texto em que a ER casou de maneira organizada, seguindo uma hierarquia. Ele retorna um objeto “collection” chamado Matches que, por sua vez, contém zero ou mais objetos Match. Cada um desses Match contém um pedaço casado, que, além do texto propriamente dito, também tem a posição de início do casamento no texto original e o tamanho desse trecho. Vamos ver um exemplo que separa os trechos por espaços em branco, com um diagrama de brinde no cabeçalho:

```
' - Hierarquia do Execute e seus objetos
'
' Execute
'   |
' Matches -> Count, Item
'   |
' Match  -> FirstIndex, Length, Value*
'
' * o Value é o padrão de acesso
'

Dim er, texto
Set er = new regexp
texto = "um dois três"
er.Pattern = "[^ ]+"
er.Global = true
Set z = er.Execute(texto) 'retorna três objetos Match
For i = 0 to (z.Count - 1) 'para cada Match,
    MsgBox z.Item(i)       'mostre o trecho casado
Next
```

Mais informações são encontradas em:

<http://aurelio.net/regex/vbscript/>



## Capítulo 9

# Bancos de dados

## MySQL

Característica	Como fazer
Busca	Operador REGEX
Substituição	-
ER crua	-
Ignore M/m	Palavra-chave BINARY
Global	-
Inverter	Operador NOT

No banco de dados MySQL é possível fazer pesquisas com expressões regulares. Basta utilizar o operador REGEXP, que tem seu funcionamento similar ao do operador LIKE. A diferença é que em vez de estar limitado ao uso dos curingas % e \_, você pode usar nossas estimadas expressões. Veja um exemplo:

```
SELECT 'MySQL' LIKE 'My%';  
SELECT 'MySQL' REGEXP '^My';
```

Esses comandos são similares, ambos procuram o texto “My” seguido de qualquer coisa. Assim como no LIKE, você também pode usar o operador NOT

para negar uma expressão, retornando apenas os registros que não casam com o padrão:

```
SELECT 'MySQL' NOT LIKE 'My%';  
SELECT 'MySQL' NOT REGEXP '^My';
```

Porém, uma diferença entre os operadores é que no REGEXP a pesquisa é sempre parcial, enquanto no LIKE é exata. Para obter todos os nomes que contêm Maria, com o LIKE é preciso fazer %Maria%. Já com o REGEX, basta informar somente Maria, que ele faz a pesquisa como se fosse .\*Maria.\*. Se quiser forçar uma pesquisa exata com o REGEXP, use as âncoras: ^Maria\$.

```
SELECT 'MySQL' LIKE 'My';      -- Não casa  
SELECT 'MySQL' REGEXP 'My';    -- Casa
```

O comportamento padrão do MySQL é sempre ignorar a diferença entre maiúsculas e minúsculas nas comparações de texto. Se você quiser forçar a diferenciação, a dica é colocar um BINARY antes da expressão.

```
SELECT 'MySQL' REGEXP '^my';    -- Casa  
SELECT 'MySQL' REGEXP BINARY '^my'; -- Não casa
```

Você pode ver em alguns exemplos o uso do operador RLIKE, que é um sinônimo para REGEXP, criado para fins de compatibilidade. Mas evite-o, pois é muito fácil confundi-lo com o operador LIKE.

Não há como especificar uma ER crua, então sempre se lembre de escapar todas as contrabarras de sua expressão. Por exemplo, se for procurar pelo texto mysql entre parênteses, faça \\(mysql\\).

Os metacaracteres usados para casar a borda de uma palavra são, no mínimo, bizarros: [[:<:]] e [[:>:]].

```
SELECT 'MySQL' REGEXP 'SQL';      -- Casa  
SELECT 'MySQL' REGEXP '[[:<:]]SQL'; -- Não casa  
SELECT 'MySQL' REGEXP 'SQL[[:>:]]'; -- Casa
```

Segue um exemplo completo de criação de tabela e obtenção dos dados usando expressões regulares:

```
mysql> CREATE TABLE dados (email TEXT);  
mysql> INSERT INTO dados VALUES('foo@bar.com');
```



```
mysql> SELECT * FROM dados WHERE email REGEXP '@[a-z.]+$';
+-----+
| email      |
+-----+
| foo@bar.com |
+-----+
mysql>
```

Mais informações são encontradas em:

<http://aurelio.net/regex/mysql/>

## Oracle

Característica	Como fazer
Busca	Operadores REGEXP_LIKE, REGEXP_INSTR, REGEXP_SUBSTR, REGEXP_COUNT
Substituição	Operador REGEXP_REPLACE
ER crua	-
Ignore M/m	Modificador i
Global	Valor 0 no quinto argumento de REGEXP_REPLACE
Inverter	Operador NOT

Lançado há 30 anos, o banco de dados Oracle permanece até hoje em constante evolução e é o preferido das empresas de grande porte. O suporte às expressões regulares, porém, veio somente em 2003, com o lançamento de sua versão 10g. Foram adicionados vários operadores novos, que veremos a seguir.

Nos exemplos seguintes, será usada uma tabela de testes chamada `dua1`. Ela é instalada por padrão junto com o Oracle, então você deve tê-la. Caso não, pode usar outra tabela qualquer, já que os exemplos não são consultas, mas, sim, meros testes em strings. Se ao executar o exemplo aparecer o número 1 na tela, isso significará que o teste deu certo, a expressão casou. Se o casamento falhar, será mostrada a mensagem “no rows selected”.

O operador criado para o casamento de expressões regulares foi o

REGEXP\_LIKE, que tem o funcionamento parecido com o tradicional LIKE, apesar de sua sintaxe de uso ser bem diferente.

```
SELECT 1 FROM dual WHERE 'Oracle' LIKE 'Ora%';  
SELECT 1 FROM dual WHERE REGEXP_LIKE('Oracle', '^Ora');
```

Esses comandos são similares, ambos procuram pelo texto “Ora” seguido de qualquer coisa. Perceba como o operador novo mais parece uma função com argumentos. Primeiro vem o texto a ser casado, ou o nome do campo de uma tabela, e, depois, a expressão regular.

Para inverter o sentido do casamento e obter apenas os registros que não casam com a expressão, use o operador NOT.

```
SELECT 1 FROM dual WHERE 'Oracle' NOT LIKE 'Ora%';  
SELECT 1 FROM dual WHERE NOT REGEXP_LIKE('Oracle', '^Ora');
```

Uma diferença importante entre os dois operadores é que no LIKE a pesquisa é sempre exata. Se você digitar somente parte da palavra, ele não irá encontrá-la. Para isso, é preciso utilizar o curinga %. Já no REGEXP\_LIKE, o casamento parcial é sempre válido.

```
SELECT 1 FROM dual WHERE 'Oracle' LIKE 'Ora';           -- Não casa  
SELECT 1 FROM dual WHERE REGEXP_LIKE('Oracle', 'Ora');  -- Casa
```

Diferentemente de outros bancos de dados, no Oracle o operador LIKE leva em conta a diferença entre letras maiúsculas e minúsculas: uma pesquisa por “oracle” não irá encontrar “Oracle” nem “ORACLE”. A partir da versão 10gR2, porém, esse comportamento pode ser mudado ao se alterar o valor das configurações NLS\_SORT e NLS\_COMP.

```
SELECT 1 FROM dual WHERE 'Oracle' LIKE 'oracle';      -- Não casa  
alter session set NLS_SORT=BINARY_CI;  
alter session set NLS_COMP=LINGUISTIC;  
SELECT 1 FROM dual WHERE 'Oracle' LIKE 'oracle';      -- Casa
```

De maneira similar, os operadores que suportam expressões regulares também são afetados pelas configurações do ambiente. Assim sendo, [a-z] pode ou não casar letras maiúsculas, dependendo da configuração. É uma confusão. Na prática, você não pode confiar no comportamento padrão.

Felizmente, o operador REGEXP\_LIKE aceita receber um terceiro argumento, que é o modificador da expressão. Para evitar problemas, é mais seguro informar em cada pesquisa se você quer que ela seja feita em modo normal (modificador c) ou ignorando a diferença entre maiúsculas e minúsculas (modificador i).

```
SELECT 1 FROM dual WHERE REGEXP_LIKE('Oracle', 'oracle');      --
SELECT 1 FROM dual WHERE REGEXP_LIKE('Oracle', 'oracle', 'i');  --
SELECT 1 FROM dual WHERE REGEXP_LIKE('Oracle', 'oracle', 'c');  --
SELECT 1 FROM dual WHERE REGEXP_LIKE('Oracle', 'Oracle', 'c');  --
```

Esse mesmo modificador pode ser usado em todos os outros operadores que veremos a seguir. Acostume-se a utilizá-lo sempre. Nos exemplos seguintes, quando ele não for utilizado, vou assumir que a configuração padrão respeita a diferença entre maiúsculas e minúsculas.

Para obter o trecho de texto casado pela expressão, use o operador REGEXP\_SUBSTR. Além dos dois argumentos principais, que são o texto e a expressão, você ainda pode informar a posição de início da busca (o padrão é 1 para sempre começar no início do texto) e o número da ocorrência (em caso de múltiplos casamentos). O modificador i deve ser colocado no final, como quinto argumento.

```
SELECT REGEXP_SUBSTR('Oracle 11g', '\d+') FROM dual;
SELECT REGEXP_SUBSTR('Oracle 11g', '[a-z]+') FROM dual;
SELECT REGEXP_SUBSTR('Oracle 11g', '[a-z]+', 1, 1, 'i') FROM dual;
SELECT REGEXP_SUBSTR('Oracle 11g', '[a-z]+', 1, 2, 'i') FROM dual;
```

Se em vez do texto casado você quiser saber apenas sua posição de início, use o operador REGEXP\_INSTR. Ele aceita os mesmos argumentos do REGEXP\_SUBSTR, com a diferença de que o modificador i aqui é o sexto argumento, e não o quinto. Veja como ficam os mesmos exemplos anteriores:

```
SELECT REGEXP_INSTR('Oracle 11g', '\d+') FROM dual;
SELECT REGEXP_INSTR('Oracle 11g', '[a-z]+') FROM dual;
SELECT REGEXP_INSTR('Oracle 11g', '[a-z]+', 1, 1, 0, 'i') FROM dual;
SELECT REGEXP_INSTR('Oracle 11g', '[a-z]+', 1, 2, 0, 'i') FROM dual;
```

Outro operador que retorna números é o REGEXP\_COUNT, que conta quantas vezes a expressão casa no texto. Esse operador, porém, só foi incluído na

versão 11g do Oracle. Use o quarto argumento se precisar passar um modificador.

```
SELECT REGEXP_COUNT('Oracle 11g', '\d') FROM dual;           -- 2
SELECT REGEXP_COUNT('Oracle 11g', '[a-z]') FROM dual;        -- 6
SELECT REGEXP_COUNT('Oracle 11g', '[a-z]', 1, 'i') FROM dual; -- 7
```

Finalmente, se você quiser substituir textos, também é possível: REGEXP\_REPLACE. Passe o texto (ou nome do campo), a expressão e o texto substituto. O quarto argumento indica a posição de início da busca (geralmente 1). O quinto argumento indica o número da ocorrência a ser substituída, ou use zero para substituir todas (global). Finalmente, no sexto argumento, vem o modificador da expressão.

```
SELECT REGEXP_REPLACE('Oracle', '[a-z]', '.') FROM dual;
SELECT REGEXP_REPLACE('Oracle', '[a-z]', '.', 1, 0) FROM dual;
SELECT REGEXP_REPLACE('Oracle', '[a-z]', '.', 1, 0, 'i') FROM dual;
SELECT REGEXP_REPLACE('Oracle', '[a-z]', '.', 1, 1, 'i') FROM dual;
SELECT REGEXP_REPLACE('Oracle', '[a-z]', '.', 1, 4, 'i') FROM dual;
```

Os retrovisores são utilizados normalmente, na notação \1, \2 etc. Você pode usá-los tanto na expressão quanto no texto substituto. Não há o retrovisor \0 para casar toda a expressão.

```
SELECT REGEXP_REPLACE('Oracle', '^(.)(.)*', '\1\2\1') FROM dual; --
```

Cuidado para não se perder nas contas, pois a posição do modificador i (ou c) varia conforme o operador utilizado. Que tal uma tabelinha para consultas rápidas?

Operador	Modificador i ou c
REGEXP_LIKE	3º argumento
REGEXP_COUNT	4º argumento
REGEXP_SUBSTR	5º argumento
REGEXP_INSTR	6º argumento
REGEXP_REPLACE	6º argumento

Segue um exemplo completo de criação de tabela e obtenção dos dados

usando expressões regulares:

```
SQL> CREATE TABLE dados (email varchar(50));
SQL> INSERT INTO dados(email) VALUES('foo@bar.com');
SQL> commit;
SQL> SELECT * FROM dados WHERE REGEXP_LIKE(email, '@[a-z.]+$', 'i');
EMAIL
-----
foo@bar.com
1 rows selected
```

Você pode aprofundar-se no uso das expressões dentro do banco de dados, indo além de simples queries. As expressões caem como luva na tarefa de validação de dados. Por exemplo, se sua tabela de clientes tem o campo cpf, você pode criar uma regra para validá-lo a cada inserção ou alteração, assegurando-se de que o número está no formato nnn.nnn.nnn-nn.

```
ALTER TABLE clientes
  ADD CONSTRAINT cpf_check
  CHECK (REGEXP_LIKE(cpf, '^\\d{3}\\.\\d{3}\\.\\d{3}-\\d{2}$'));
```

Mais informações são encontradas em:

<http://aurelio.net/regex/oracle/>

## PostgreSQL

Característica	Como fazer
Busca	Função substring, operador ~
Substituição	Função regexp_replace
ER crua	-
Ignore M/m	Modificador i, operador ~*
Global	Modificador g
Inverter	Operador !~

As expressões regulares também podem ser usadas no banco de dados PostgreSQL. Em vez de um operador como REGEXP, utiliza-se o til ~.

Compare a sintaxe de uso do LIKE com o operador til:

```
SELECT 'PostgreSQL' LIKE 'Post%';  
SELECT 'PostgreSQL' ~ '^Post';
```

Esses são comandos similares, então, se você já está acostumado ao operador LIKE, basta trocá-lo por um til e escrever sua expressão. Para negar uma expressão, não se utiliza o NOT, mas uma exclamação na frente do til:

```
SELECT 'PostgreSQL' NOT LIKE 'Post%';  
SELECT 'PostgreSQL' !~ '^Post';
```

No PostgreSQL, todas as pesquisas com expressões regulares são feitas levando-se em conta a diferença entre letras maiúsculas e minúsculas. Se você quiser um comportamento igual ao do LIKE, em que “M” e “m” são considerados iguais, coloque um asterisco após o til:

```
SELECT 'PostgreSQL' ~ '^post';      -- Não casa  
SELECT 'PostgreSQL' ~* '^post';     -- Casa
```

Diferentemente do LIKE, o til casa textos parciais, é como se todas as expressões viessem com um .\* antes e depois. Para fazer uma pesquisa exata, é preciso usar as âncoras ^ e \$.

```
SELECT 'PostgreSQL' LIKE 'Post';      -- Não casa  
SELECT 'PostgreSQL' ~ 'Post';         -- Casa (parcial)  
SELECT 'PostgreSQL' ~ '^PostgreSQL$'; -- Casa (exato)
```

Não há como especificar uma ER crua, então sempre se lembre de escapar todas as contrabarras de sua expressão. Por exemplo, se for procurar pelo texto postgresql entre parênteses, faça \\(postgresql\\).

Além do operador de pesquisa, também é possível manipular e alterar textos usando expressões regulares. Com a função substring você extrai partes de um texto. Se não houver nenhum grupo, todo o trecho casado pela expressão será retornado. Senão, será extraído somente o conteúdo do primeiro grupo. Se a expressão não casar nada, o retorno será NULL.

```
SELECT substring('PostgreSQL' from '...$');      -- SQL  
SELECT substring('PostgreSQL' from '(.)(..)$');  -- S  
SELECT substring('PostgreSQL' from '[0-9]$');    -- NULL
```

Mas o poder verdadeiro está na função `regexp_replace`, que faz substituições, facilitando muito sua tarefa de manipular e formatar os dados. Passe o texto (ou nome do campo), a expressão e o texto substituto. O quarto parâmetro é opcional, servindo para especificar os modificadores `g` e `i`, que indicam substituição global, e ignorar maiúsculas e minúsculas, respectivamente.

```
SELECT regexp_replace('PostgreSQL', '[A-Z]', '.');      -- .ostgr
SELECT regexp_replace('PostgreSQL', '[A-Z]', '.', 'g');  -- .ostgr
SELECT regexp_replace('PostgreSQL', '[A-Z]', '.', 'gi'); -- .....
```

Também é possível utilizar os retrovisores `\1`, `\2`, entre outros. O retrovisor especial `&` guarda todo o trecho casado pela expressão. Mas sempre se lembre de escapar as contrabarras no momento de utilizá-las.

```
SELECT regexp_replace('PostgreSQL', '.*(...)', '\\1' );  -- SQL
SELECT regexp_replace('PostgreSQL', '.*', '(\&)', );    -- (Postg
```

Segue um exemplo completo de criação de tabela e obtenção dos dados usando expressões regulares:

```
piazinho=# CREATE TABLE dados (email TEXT);
CREATE TABLE
piazinho=# INSERT INTO dados VALUES('foo@bar.com');
INSERT 0 1
piazinho=# SELECT * FROM dados WHERE email ~* '@[a-z.]+$';
      email
-----
foo@bar.com
(1 row)
piazinho=#
```

Mais informações são encontradas em:

<http://aurelio.net/regex/postgresql/>

## SQLite

Característica	Como fazer
Busca	Operador REGEX

Substituição	-
ER crua	É o padrão
Ignore M/m	-
Global	-
Inverter	Operador NOT

O SQLite é um banco de dados enxuto, leve e prático criado em 2000. Desde seu início, foi projetado para não ser complicado. Toda a base de dados reside em um único arquivo e, em vez de rodar como um aplicativo autônomo, ele pode ser embutido diretamente dentro de seu programa. O navegador Firefox, por exemplo, traz o SQLite consigo.

Muitas linguagens de programação permitem embutir o SQLite em seus programas, entre elas, a maioria das citadas neste livro. É importante saber isso, pois apesar de o SQLite trazer o operador REGEXP, ele vem vazio. Quem deve fazer o processamento e dizer se a expressão casou ou não é a linguagem hospedeira.

Desse modo, se você está programando em Python, o operador REGEXP vai usar as expressões regulares do Python. Se estiver no Tcl, vão ser as expressões do Tcl, e assim por diante. Em outras palavras, o SQLite não tem suas próprias expressões regulares, mas espera que cada linguagem ceda as suas quando for utilizá-lo.

Pode parecer frustrante em um primeiro momento, mas é uma grande vantagem você não precisar aprender uma nova sintaxe das expressões só para usar no banco de dados. Você pode continuar usando as mesmas expressões que já está acostumado.

A biblioteca SQLite de sua linguagem preferida já deve ter tudo funcionando, mas caso apareça uma mensagem dizendo que o operador REGEXP não existe, só está faltando ligá-lo. Foge do escopo deste livro abordar detalhadamente esse assunto, mas em resumo basta criar uma função simples para casar o texto e informá-la à função `sqlite_create_function`. Veja um exemplo em Python. Você pode fazer algo parecido em sua linguagem:



```

import sqlite3, re
# Abre a conexão com o banco de dados
conexao = sqlite3.connect(":memory:")
# Passo 1
# Criar a função que testa se a ER casa com o texto.
# A ordem dos argumentos é: expressão, texto.
# O retorno deve ser True ou False.
#
def regexp(er, texto):
    return re.search(er, texto) is not None
# Passo 2
# Informar ao SQLite qual função usar no operador REGEXP.
# Basta chamar a create_function e passar nossa função.
#
conexao.create_function("regexp", 2, regexp)
# Pronto. O operador REGEXP já está funcionando.
# Teste do operador
print conexao.execute("SELECT 'SQLite' REGEXP '^SQL'").fetchone()

```

Mas chega de preparativos, vamos aos exemplos de como utilizar as expressões, que são a parte divertida da brincadeira. Use o operador REGEXP do mesmo modo como usaria o operador LIKE. Não tem segredo!

```

SELECT 'SQLite' LIKE 'SQL%';
SELECT 'SQLite' REGEXP '^SQL';

```

Esses dois comandos são similares e ambos retornam OK. Viu como é simples? Basta trocar o operador e colocar sua expressão entre aspas, lembrando que os metacaracteres serão os mesmos da linguagem de programação que você estiver utilizando por baixo.

Como de costume, você também pode usar o operador NOT se quiser inverter o sentido da expressão, fazendo com que sejam retornados apenas os registros que não casarem.

```

SELECT 'SQLite' NOT LIKE 'SQL%';
SELECT 'SQLite' NOT REGEXP '^SQL';

```

Uma diferença importante entre os dois operadores é que no LIKE a pesquisa é sempre exata, então, se você digitar somente parte da palavra, ele

não irá encontrá-la. Por isso, é preciso sempre utilizar o curinga %. Já no REGEX, o casamento parcial é válido e você não precisa preocupar-se com isso.

```
SELECT 'SQLite' LIKE 'SQL';      -- Não casa
SELECT 'SQLite' REGEXP 'SQL';    -- Casa
```

O comportamento padrão do operador LIKE é sempre ignorar a diferença entre maiúsculas e minúsculas nas comparações de texto. Já o operador REGEXP é independente e seu comportamento nessa questão varia conforme a linguagem hospedeira ou a função que você for carregar para ele. Faça o teste para garantir, mas, em geral, expressões regulares diferenciam M de m.

```
SELECT 'A' LIKE 'a';            -- Casa
SELECT 'A' REGEXP 'a';          -- Depende
```

O tratamento de strings do SQLite é bem primário, o único escape que ele reconhece é duplicar as aspas para poder incluir aspas literais. Por exemplo, para incluir o texto “It's ok” deve-se usar “It''s ok”. Não há suporte aos tradicionais escapes com a contrabarra, como \'. Essa é uma ótima notícia para nós, pois, na prática, podemos considerar que as ERs são sempre cruas, evitando dores de cabeça com escapes indesejados.

Segue um exemplo completo de criação de tabela e obtenção dos dados usando expressões regulares:

```
CREATE TABLE dados (email TEXT);
INSERT INTO dados VALUES('foo@bar.com');
SELECT * FROM dados WHERE email REGEXP '@[a-z.]+$';
```

Mais informações são encontradas em:

<http://aurelio.net/regex/sqlite/>



## Capítulo 10

# Bibliotecas e programas relacionados

Todos os programas e as bibliotecas listados aqui são softwares de código aberto, ou seja, você tem acesso aos fontes e não precisa pagar para utilizá-los. E, além de tudo, são de excelente qualidade.

## Bibliotecas

Além do POSIX, outros módulos para usar ERs em C são o regex original de 1986 do Henry Spencer, atualizado (<http://www.codeproject.com/KB/string/spencerregexp.aspx>) e o regex da GNU (<http://www.gnu.org/software/gnulib>). Essas bibliotecas são plug'n'play, basta incluir o diretório com os fontes de seu programa, sem maiores traumas.

Para C++ tem a biblioteca Regexx (<http://sourceforge.net/projects/regexx/>) que é uma solução completa com execução, busca, substituição, divisão e fácil acesso aos textos casados.

Há uma biblioteca mais poderosa para C, chamada PCRE (<http://www.pcre.org>), que traz a funcionalidade avançada das expressões do Perl. Se você está iniciando um projeto e não sabe qual biblioteca utilizar, prefira essa. Ela é moderna, eficiente e utilizada por projetos grandes como Apache e PHP.

# Programas

O programa `txt2regex` (<http://aurelio.net/projects/txt2regex/>) é um assistente (“wizard”) que constrói ERs. Você apenas responde perguntas (em português inclusive) e a ER vai sendo construída. Por conhecer metacaracteres de diversos aplicativos, também é útil para tirar dúvidas de sintaxe.

O Visual REGEXP (<http://laurent.riesterer.free.fr/regexp>), que mostra graficamente, em cores distintas, cada parte da ER e o que ela casa em um determinado texto. Simples e muito útil, bom para iniciantes e ERs muito complicadas.

O Regex Coach (<http://www.weitz.de/regex-coach/>) é um programa avançado que, além da operação básica de mostrar o que casou, ainda pode simular um split e uma substituição, mostra o conteúdo dos grupos e ainda mostra um gráfico para lhe ajudar a entender como o robzinho “enxerga” a sua ER.

Há excelentes sites que testam e analisam suas ERs diretamente no navegador, sem precisar baixar e instalar nenhum programa. Entre os vários testadores, um que se destaca é o Regexpal (<http://regexpal.com>). Ele é tão eficiente que já vai casando a expressão enquanto você a digita, sem precisar apertar nenhum botão. Outra ferramenta excelente é o Regex101 (<https://regex101.com>) que além de casar expressões em tempo real, permite fazer substituições no texto. Experimente também o Debuggex (<https://www.debuggex.com>), que gera um gráfico com a estrutura de sua expressão, ficando mais fácil entendê-la.



## Apêndice A

# Onde obter mais informações

A Errata e as informações complementares deste livro estão no site do livro em [www.piazinho.com.br](http://www.piazinho.com.br).

No capítulo de cada linguagem de programação há dicas de como encontrar mais informações. A documentação que acompanha cada aplicativo também pode ajudar bastante. Seguem dicas de informações relacionadas às expressões regulares em geral.

## Livros

Para conhecer a fundo os detalhes do funcionamento das expressões, com testes de performance e análises de táticas de implementação, leia a bíblia das ERs: *Mastering Regular Expressions*, de Jeffrey E. F. Friedl, ISBN 0-596-00289-0 (<http://regex.info>).

Para saber como funcionam internamente e como são implementados os robozinhos, o assunto é “autômatos finitos determinísticos e não-determinísticos”, que consta em qualquer livro que explique como escrever compiladores.

## Fóruns

O grupo sed-br é sobre o Sed, mas é aberto para discussões gerais sobre ERs (em português). Para se inscrever, mande um e-mail para [sed-br-subscribe@yahoogrupos.com.br](mailto:sed-br-subscribe@yahoogrupos.com.br) ou visite o seu site em <http://br.groups.yahoo.com/group/sed-br>.

Fora este, temos outros grupos nacionais sobre linguagens e editores de texto em que dúvidas sobre como aplicar a ER neles podem ser sanadas:

[http://groups.google.com/group/dotnet\\_br](http://groups.google.com/group/dotnet_br)

<http://br.groups.yahoo.com/group/java-br/>

<http://br.groups.yahoo.com/group/javascript-br/>

<http://groups.google.com/group/lua-br>

<http://br.groups.yahoo.com/group/mysql-br/>

<http://br.groups.yahoo.com/group/php-pt/>

<http://br.groups.yahoo.com/group/postgres-pt/>

<http://br.groups.yahoo.com/group/python-brasil/>

<http://br.groups.yahoo.com/group/shell-script/>

[http://groups.yahoo.com/group/usuarios\\_oracle](http://groups.yahoo.com/group/usuarios_oracle)

<http://br.groups.yahoo.com/group/vbmaster/>

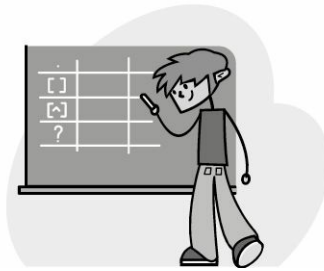
<http://br.groups.yahoo.com/group/vi-br>

## Internet

O portal nacional de expressões regulares fica em [www.aurelio.net/regex/](http://www.aurelio.net/regex/), com uma lista extensa de sites sobre o assunto, em português.

Em inglês, visite o tópico de expressões regulares na Wikipedia, que, além de informações teóricas e históricas, traz uma bela coleção de links para engrandecer seus conhecimentos ([http://en.wikipedia.org/wiki/Regular\\_expression](http://en.wikipedia.org/wiki/Regular_expression)).

Há um banco de dados de ERs em <http://regexlib.com> que se propõe a armazenar diversas expressões criadas pelos usuários. Quando estiver em dificuldades para criar sua expressão, procure neste site se não há algo já pronto para resolver seu problema.



## Apêndice B

# Tabelas

Você deve estar se perguntando: “Mas por que esse cara colocou essas tabelas aqui no final?”.

É bom encontrar rapidamente o que se procura, certo? Não é irritante ficar folheando algo procurando uma tabela perdida em meio a um texto? E quando tem, o índice de tabelas nunca nos dá descrições muito claras do seu conteúdo.

Então, por isso, elas estão todas aqui. Após a leitura, você vai jogar este livrinho em um canto e só vai vê-lo novamente quando precisar tirar alguma dúvida que geralmente está respondida em uma das tabelas. Nessa hora de aperto, basta ver as últimas páginas, simples como expressões regulares!

Também preparei uma versão especial em PDF, juntando todas estas tabelas em uma única página A4. Boa para imprimir, colar na parede do quarto, mandar para os amigos, usar como papel de parede no computador, guardar no smartphone... Acesse o site do livro [www.piazinho.com.br](http://www.piazinho.com.br) para fazer o download do PDF.

## Diferenças de metacaracteres entre aplicativos

--	--	--	--	--	--	--

Programa	Opcional	Mais	Chaves	Borda	Ou	Grupo
Apache	?	+	{,}	\b		()
Awk	?	+				()
Bash	?	+	{,}	\b		()
C	?	+	{,}			()
Ed	\?	\+	\{, \}	\b	\	\(\)
Egrep	?	+	{,}	\b		()
Emacs	?	+	\{, \}	\b	\	\(\)
Expect	?	+				()
Gawk	?	+	{,}	\< \>		()
Google Docs	?	+	{,}	\b		()
Grep	\?	\+	\{, \}	\b	\	\(\)
HTML 5	?	+	{,}	\b		()
Java	?	+	{,}	\b		()
JavaScript	?	+	{,}	\b		()
Lex	?	+	{,}			()
Lua	?	+				()
Mawk	?	+				()
MySQL	?	+	{,}	[[:<:]]		()
.NET	?	+	{,}	\b		()
OpenOffice	?	+	{,}	\< \>		()
Oracle	?	+	{,}			()
Perl	?	+	{,}	\b		()
PHP	?	+	{,}	\b		()
PostgreSQL	?	+	{,}	\y		()
Python	?	+	{,}	\b		()



Ruby	?	+	{,}	\b		()
Sed	\?	\+	\{, \}	\< \>	\	\(\)
Tcl	?	+	{,}	\< \>		()
VBscript	?	+	{,}	\b		()
Vim	\=	\+	\{, }	\< \>	\	\(\)

Nota: . \* [] [^] ^ \$ e \ são iguais pra todos.

## Resumão dos metacaracteres e seus detalhes

Metacaractere	Nome	Dicas
.	Ponto	Curinga de um caractere
[]	Lista	Dentro todos são normais, traço é intervalo ASCII, [:POSIX:] tem acentuação
[^]	Lista negada	Sempre casa algo, [^[:POSIX:]]
?	Opcional	Guloso, 0 ou 1, pode ter ou não
*	Asterisco	Guloso, 0 ou mais, repete em qualquer quantidade
+	Mais	Guloso, 1 ou mais, repete em qualquer quantidade, pelo menos uma vez
{, }	Chaves	Guloso, número exato, mínimo, máximo, ou uma faixa numérica
^	Circunflexo	Casa o começo da linha, especial no começo da ER
\$	Cifrão	Casa o fim da linha, especial no fim da ER
\b	Borda	Limita uma palavra (letras, números e sublinhado)
\	Escape	Escapa um meta, tira seu poder, escapa a si mesmo \\
	Ou	Indica alternativas, poder multiplicado pelo grupo
()	Grupo	Agrupa, é quantificável, pode conter outros grupos
\1	Retrovisor	Usado com o grupo, máximo 9, conta da esquerda para direita
.*	Curinga	Qualquer coisa, o tudo e o nada

??	Opcional	Não-guloso, 0 ou 1, casa o mínimo possível
*?	Asterisco	Não-guloso, 0 ou mais, casa o mínimo possível
+?	Mais	Não-guloso, 1 ou mais, casa o mínimo possível
{ }?	Chaves	Não-guloso, numérico, casa o mínimo possível

# POSIX, barra-letas e outros aliens

POSIX	Similar	Significa
[ :upper: ]	[ A-Z ]	Letras maiúsculas
[ :lower: ]	[ a-z ]	Letras minúsculas
[ :alpha: ]	[ A-Za-z ]	Maiúsculas e minúsculas
[ :alnum: ]	[ A-Za-z0-9 ]	Letras e números
[ :digit: ]	[ 0-9 ]	Números
[ :xdigit: ]	[ 0-9A-Fa-f ]	Números hexadecimais
[ :punct: ]	[ ., !?:... ]	Caracteres de pontuação
[ :blank: ]	[ \t ]	Espaço em branco e Tab
[ :space: ]	[ \t\n\r\f\v ]	Caracteres brancos
[ :cntrl: ]		Caracteres de controle
[ :graph: ]	[ ^ \t\n\r\f\v ]	Caracteres imprimíveis
[ :print: ]	[ ^\t\n\r\f\v ]	Imprimíveis e o espaço

b-l	Nome	Tradução
\a	Alert	Alerta (bipe)
\b	Backspace	Caractere Backspace
\e	Escape	Caractere Esc
\f	Form feed	Alimentação
\n	Newline	Linha nova

\r	Carriage return	Retorno de carro
\t	Htab	Tabulação horizontal
\v	Vtab	Tabulação vertical

b-l	Equivalente POSIX	Significa
\d	[[ :digit:]]	Dígito
\D	[^ :digit:]]	Não-dígito
\w	[[ :alnum:]]_]	Palavra
\W	[^ :alnum:]]_]	Não-palavra
\s	[[ :space:]]	Branco
\S	[^ :space:]]	Não-branco

b-l	Significado	Similar
\a	Alfabeto	[[ :alpha:]]
\A	Não alfabeto	[^ :alpha:]]
\h	Cabeça de palavra	[[ :alpha]]_]
\H	Não cabeça de palavra	[^ :alpha:]]_]
\l	Minúsculas	[[ :lower:]]
\L	Não minúsculas	[^ :lower:]]
\u	Maiúsculas	[[ :upper:]]
\U	Não maiúsculas	[^ :upper:]]
\o	Número octal	[0-7]
\O	Não número octal	[^0-7]
\B	Não-borda	
\A	Início do texto	
\Z	Fim do texto	
\l	Torna minúscula	

\L	Torna minúscula até \E	
\u	Torna maiúscula	
\U	Torna maiúscula até \E	
\Q	Escapa até \E	
\E	Fim da modificação	
\G	Fim do casamento anterior	

## Modernos, remendos e precedência

Metacaractere	Significado	Dica
(?#texto)	Texto é um comentário	
(?:ER)	Grupo fantasma, retrovisor não conta	
(?=ER)	Casa se ER casar adiante	==
(?!ER)	Casa se ER não casar adiante	!=
(?<=ER)	Casa se ER casar antes	<==
(?<!ER)	Casa se ER não casar antes	<!=
(?<nome>ER)	Nomeia parte de uma ER	
(?letra)	Letra é um modificador: imsxL	
(?{código})	Executa código Perl	
(?(cond)s n)	If-then-else	

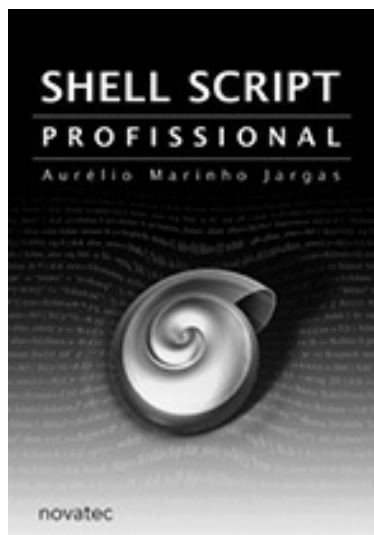
Classe POSIX	Remendo
[[:lower:]]	[a-zà-ü]
[[:upper:]]	[A-ZÀ-Ü]
[[:alpha:]]	[A-Za-zÀ-Ü]
[[:alnum:]]	[A-Za-zÀ-Ü0-9]

Tipo de meta	Exemplo	Precedência
--------------	---------	-------------



52	4	84	T	116	t	182	¶	214	Ö	246	ö
53	5	85	U	117	u	183	·	215	×	247	÷
54	6	86	V	118	v	184	,	216	Ø	248	ø
55	7	87	W	119	w	185	¹	217	Ù	249	ù
56	8	88	X	120	x	186	º	218	Ú	250	ú
57	9	89	Y	121	y	187	»	219	Û	251	û
58	:	90	Z	122	z	188	¼	220	Ü	252	ü
59	;	91	[	123	{	189	½	221	Ý	253	ý
60	<	92	\	124		190	¾	222	Þ	254	þ
61	=	93	]	125	}	191	¿	223	ß	255	ÿ
62	>	94	^	126	~	192	À	224	à		
63	?	95	_	161	ı	193	Á	225	á		

**CONHEÇA TAMBÉM DO MESMO AUTOR - SHELL SCRIPT  
PROFISSIONAL**



*“O melhor livro técnico que eu já li.”*

– **THIAGO NUNES VILELA**

*“Perfeito. O melhor livro que já comprei.”*

– **CAIO CESAR CECCON DE AZEVEDO**

*“Foi um dos melhores investimentos que já fiz.”*

– **RODRIGO AMORIM FERREIRA**

*“Ótimo livro. Descontraído e gostoso de ler.”*

– **ALCEU DE LIMA SAMPAIO**

*“Genial. Simples, direto e muito bem explicado.”*

– **BRUNO GONÇALVES TIKAMI**

*“Indispensável na mesa de qualquer administrador de sistemas.”*

– **CHRISTIANO ANDERSON**

*“Não só ensina o Shell de fato, como orienta e incentiva boas práticas de programação.”*

– **SAMIR BRAGA**

*“O que ganhei de produtividade com as dicas do livro valeu cada centavo investido na compra.”*

– **GUILHERME MAGALHÃES GALL**

*“Antes de ler esse livro, desenvolvia scripts sem tomar qualquer cuidado com o fonte. Hoje desenvolvo verdadeiros programas.”*

– **THIAGO SANTANA**

No site do livro há dezenas de depoimentos de leitores satisfeitos:

[www.shellscript.com.br](http://www.shellscript.com.br)