
Expressões Regulares
Julio Neves
Expressões Regulares

Expressão Regular é:

Curto e grosso:

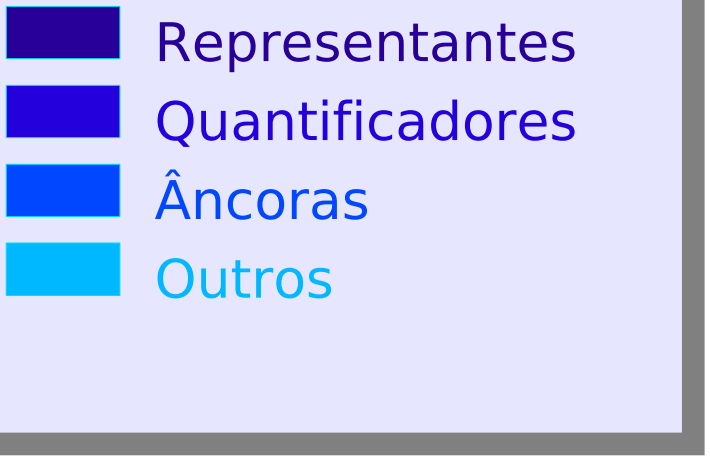
Um método formal de se especificar um padrão de texto

Mas também podemos dizer que é uma:

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

Metacaracteres

Meta	Mnemônico
.	ponto
[]	lista
[^]	lista negada
?	opcional
*	asterisco
+	mais
{}	chaves
^	circunflexo
$	cifrão
\b	borda
\	escape
|	ou
()	grupo
\n	retrovisor



Metacaracteres Representantes

Metacaracteres do tipo representante, aqueles cuja função é representar um ou mais caracteres.

São eles:

- Ponto `.`
- Lista `[...]`
- Lista Negada `[^...]`

Ponto: o necessitado

O ponto é um fácil e casa com qualquer um, 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. É um promíscuo :-)

Pode ser usado para procurar:

Palavras que não lembra se acentuou ou não
Palavras que que podem começar com maiúsculas ou não
Pobremas com o *portugueiz*
Horário com qualquer separador
Marcações (*tags*) HTML

Exemplos:	**Casa com:**
`coc.`	`coco, cocô, ...`
`.osta`	`costa, Costa, ...`
`ca.ado`	`casado, cazado, ...`
`12.45`	`12:45, 12 45, 12.45, ...`
`<.>`	`<B>, <i>, <p>, ...`

Lista: a exigente

A lista é muito mais seletiva que o ponto, pois não casa com qualquer um. A lista só casa com quem ela está afim.

Pode ser usado para procurar:

Palavras que não lembra se acentuou ou não
Palavras que que podem começar com maiúsculas ou não
Pobremas com o *portugueiz*
Horário com qualquer separador
Marcações (*tags*) HTML

Exemplos:	**Casa com:**
`coc[oô]`	coco, cocô
`[Cc]osta`	Costa, costa
`ca[sz]ado`	casado, cazado
`12[: .]45`	12:45, 12 45, 12.45
`<[BIP]>`	<B>, <I>, <P>

Lista: a exigente

Evite:

```
[01234567890]
[0-9][0-9]:[0-9][0-9]
[A-z]
```

Prefira:

```
[0-9]
[012][0-9]:[0-5][0-9]
[A-Za-z]
```

Classes de Caracteres POSIX

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
`[:digit:]`	`[0-9]`	números
`[:xdigit:]`	`[0-9A-Fa-f]`	números hexadecimais
`[:punct:]`	`[.,!?:...]`	sinais de pontuação
`[:blank:]`	`[ \t]`	espaço 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

↓ Isso que define uma lista ↓ **↓ idem ↓** **↓ idem ↓**

`[[:upper:][:lower:][:digit:]]` **=** `[[:alpha:][:digit:]]` **=** `[[:alnum:]]`

Para listar endereços *mac* do `ifconfig` (1ª Versão)

```
$ ifconfig | grep '[[:xdigit:]][[:xdigit:]]:[[:xdigit:]][[:xdigit:]]:[[:xdigit:]][[:xdigit:]]:[[:xdigit:]][[:xdigit:]]:[[:xdigit:]][[:xdigit:]]:[[:xdigit:]][[:xdigit:]]'

eth0  Link encap:Ethernet  Endereço de HW 1c:75:08:b0:75:76
wlan0 Link encap:Ethernet  Endereço de HW 88:9f:fa:61:9b:10

$ ifconfig | grep -o '[[:xdigit:]][[:xdigit:]]:[[:xdigit:]][[:xdigit:]]:[[:xdigit:]][[:xdigit:]]:[[:xdigit:]][[:xdigit:]]:[[:xdigit:]][[:xdigit:]]:[[:xdigit:]][[:xdigit:]]'

1c:75:08:b0:75:76
88:9f:fa:61:9b:10
```

Lista Negada: a Experiente

- 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, fora os componentes listados.
- Observe que a diferença em sua notação é que o primeiro caractere da lista é um circunflexo `(^)`, ele indica que esta 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.

Lista Negada: a Experiente

Exemplos

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 o espaço após a pontuação:

```
[:;,.!?][^ ]
```

Ou, ainda, explicitando melhor nosso objetivo:

```
[[:punct:]][^ ]
```

Metacaracteres Quantificadores

Os quantificadores 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, 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.**

São eles:

- opcional **?**
- asterisco *****
- mais **+**
- chaves **{ }**

O opcional ?

O ponto de perguntação deixa a entidade anterior opcional, ou seja, pode existir ou não. Um bom exemplo vale mais que mil palavras:

Expressão:	**Casa com:**
`chopes?`	`chope chopes`
`gr?ota`	`gota grota`
`r?ondas?`	`onda ronda ondas rondas`
`e?namorada`	`namorada enamorada`
`[cf]?alar?`	`ala cala fala calar falar`
`[tcmv]?ela`	`ela tela cela mela vela`
`</?[BIP]>`	`</B> </I> </P> <B> <I> <P>`

Asterisco: do zero ao tudo

O uso do asterisco permite que a entidade anterior apareça em qualquer quantidade, de zero a zilhões.

Expressão:	**Casa**:
`5*1`	`1, 51, 5551, 55555555555551, ...`
`cai*ndo`	`cando, caindo, caiiiindo, caiiiiiiiiiiindo, ...`
`l.*x`	`lx, lax, lemx, linux, lsertvxx, ...`
`ca[ru]*`	`ca, car, cau, caru, caruru, caurruurur, ...`

O asterisco é guloso e sempre tentará casar o **máximo** que conseguir.

```
$ grep 'f.*e' quebralingua.txt
Eu vi um velho com um fole velho nas costas.
Tanto fede o fole do velho,
quanto o velho do fole fede.
$ grep 'f[a-z]*e' quebralingua.txt
Eu vi um velho com um fole velho nas costas.
Tanto fede o fole do velho,
quanto o velho do fole fede.
```

Mais: o pelo meno um

Igualzinho ao 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.

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

Expressão	Casa
`5+1`	`51, 5551, 55555555555551, ...`
`cai+ndo`	`caindo, caiiiindo, caiiiiiiiiiiindo, ...`
`l.+x`	`lax, lemx, linux, lsertvxx, ...`
`ca[ru]+`	`car, cau, caru, caruru, caurruurur, ...`

O mais também é guloso e precisa de **expansão de Reg Exp** (`-E` no `grep`)

```
$ grep -E '1.+9' <<< "01234567890123456789012345678901"
01234567890123456789012345678901
```

Chaves: o controle {n,m}

As chaves são a solução para uma quantificação mais controlada, onde se pode especificar exatamente quantas repetições se quer da entidade anterior.

`{n,m}` significa de `n` até `m` vezes, assim algo como `3{1,4}` casa `3`, `33`, `333` e `3333`. 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:

`{0,1}`	zero ou 1	(igual ao opcional)
`{0,}`	zero ou mais	(igual ao asterisco)
`{1,}`	um ou mais	(igual ao mais)
`{3}`	exatamente	

Para listar endereços *mac* do `ifconfig` (2ª Versão)

```
$ ifconfig | grep -E '[[:xdigit:]]{2}:[[:xdigit:]]{2}:[[:xdigit:]]{2}:[[:xdigit:]]{2}:[[:xdigit:]]{2}:[[:xdigit:]]{2}'

eth0  Link encap:Ethernet  Endereço de HW 1c:75:08:b0:75:76
wlan0 Link encap:Ethernet  Endereço de HW 88:9f:fa:61:9b:10

$ ifconfig | grep -Eo '[[:xdigit:]]{2}:[[:xdigit:]]{2}:[[:xdigit:]]{2}:[[:xdigit:]]{2}:[[:xdigit:]]{2}:[[:xdigit:]]{2}'

1c:75:08:b0:75:76
88:9f:fa:61:9b:10
```

Metacaracteres tipo Âncora

Por que âncora? Porque eles não casam caracteres ou definem quantidades, ao invés 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.

São eles:

- circunflexo **^**
- cifrão **$**
- borda **\b**

Circunflexo: o início ^

O nosso amigo circunflexo (^) marca o começo de uma linha. Nada mais.

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

`^[0-9]` Casa com uma linha começando com qualquer algarismo

O inverso disso seria:

`^[^0-9]`

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.

Para demonstrar seu uso, vamos primeiramente entender como funciona o `grep` com a opção `-v`:

```
$ cat -n /etc/passwd | head -4
     1  root:x:0:0:root:/root:/bin/bash
     2  daemon:x:1:1:daemon:/usr/sbin:/bin/sh
     3  bin:x:2:2:bin:/bin:/bin/sh
     4  sys:x:3:3:sys:/dev:/bin/sh
$ cat -n /etc/passwd | head -4 | grep -v root
     2  daemon:x:1:1:daemon:/usr/sbin:/bin/sh
     3  bin:x:2:2:bin:/bin:/bin/sh
     4  sys:x:3:3:sys:/dev:/bin/sh
```

Cifrão: o fim $

`[0-9]$` Casa linhas que terminam com um número

```
$ ifconfig | grep -v '[0-9]$'
eth0      Link encap:Ethernet  Endereço de HW 1c:75:08:b0:75:76
          colisões:0 txqueuelen:1000
          RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
          IRQ:45

lo        Link encap:Loopback Local
          endereço inet6: ::1/128 Escopo:Máquina
          colisões:0 txqueuelen:0
          RX bytes:11430869 (11.4 MB) TX bytes:11430869 (11.4 MB)

wlan0     Link encap:Ethernet  Endereço de HW 88:9f:fa:61:9b:10
          endereço inet6: fe80::8a9f:faff:fe61:9b10/64 Escopo:Link
          colisões:0 txqueuelen:1000
          RX bytes:1077769309 (1.0 GB) TX bytes:242283590 (242.2 M
B)
```

Cifrão: o fim $

Tá errado não, veja só!

```
$ ifconfig | grep -v '[0-9]$' | cat -vet
eth0      Link encap:Ethernet  Endereço de HW 1c:75:08:b0:75:76  $
          colisões:0 txqueuelen:1000 $
          RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)$
          IRQ:45 $
$
lo        Link encap:Loopback Local  $
          endereço inet6: ::1/128 Escopo:Máquina$
          colisões:0 txqueuelen:0 $
          RX bytes:11430898 (11.4 MB) TX bytes:11430898 (11.4 MB)$
$
wlan0     Link encap:Ethernet  Endereço de HW 88:9f:fa:61:9b:10  $
          endereço inet6: fe80::8a9f:faff:fe61:9b10/64 Escopo:Link$
          colisões:0 txqueuelen:1000 $
          RX bytes:1077866815 (1.0 GB) TX bytes:242397033 (242.3 M
B)$
```

Cifrão: o fim $

Então vamos excluir as linhas que terminam com algarismos seguidos ou não de espaços em branco

```
$ ifconfig | grep -v '[0-9] *$'
          RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

lo        Link encap:Loopback Local
          endereço inet6: ::1/128 Escopo:Máquina
          RX bytes:11430927 (11.4 MB) TX bytes:11430927 (11.4 MB)

          endereço inet6: fe80::8a9f:faff:fe61:9b10/64 Escopo:Link
          RX bytes:1077954195 (1.0 GB) TX bytes:242488854 (242.4 M
B)
```

Não gostei das linhas vazias, vamos tirá-las também.

Cifrão: o fim $

Podemos dizer que linhas vazias, têm o início junto do fim, ou seja `^$`. Vejamos:

```
$ ifconfig | grep -v '[0-9] *$' | grep -v ^$
          RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
lo        Link encap:Loopback Local
          endereço inet6: ::1/128 Escopo:Máquina
          RX bytes:11432149 (11.4 MB) TX bytes:11432149 (11.4 MB)
          endereço inet6: fe80::8a9f:faff:fe61:9b10/64 Escopo:Link
          RX bytes:1078122010 (1.0 GB) TX bytes:242621934 (242.6 MB)
```

Mas já te digo, porém sem explicar, o processo nipônico seria:

```
$ ifconfig | grep -vE '[0-9] *$|^$'
```

Cifrão: o fim $

Que acontecerá se eu executar a linha a seguir:

```
$ grep -E '^.{20,40}$' /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
irc:x:39:39:ircd:/var/run/ircd:/bin/sh
saned:x:111:118::/home/saned:/bin/false
```

Cifrão: o fim $

Vamos diminuir a faixa de pesquisa para ver se é isso mesmo:

```
$ grep -E '^.{20,30}$' /etc/passwd
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
```

Borda: a limítrofe \b

A borda é muito útil para casar palavras inteiras, porque ela define os seus limites anterior e/ou posterior.

Palavra aqui é uma sequência de caracteres que atenda à seguinte lista: **`[A-Za-z0-9_]`**, isto é, letras, números e o sublinhado.

Veja como se comportam as ERs nas palavras: **`data, concordata, data-center, datado:`**

```
data      data, concordata, data-center, datado
\bdata    data, data-center, datado
data\b    data, concordata, data-center
\bdata\b  data, data-center
```

Borda: a limítrofe \b

O mesmo papel da borda, pode ser obtido com os sinais de menor e maior escapados `(\< e \>)`, porém...

```
$ grep '\<f..e\>' quebralingua.txt
Eu vi um velho com um fole velho nas costas.
Tanto fede o fole do velho,
quanto o velho do fole fede.
$ grep '\bf..e\b' quebralingua.txt
Eu vi um velho com um fole velho nas costas.
Tanto fede o fole do velho,
quanto o velho do fole fede.
$ grep '\< f..e \>' quebralingua.txt
$ grep '\b f..e \b' quebralingua.txt
Eu vi um velho com um fole velho nas costas.
Tanto fede o fole do velho,
quanto o velho do fole fede.
```

Outros Metacaracteres

Os curingas a seguir não se encaixam em nenhuma das pseudo categorias anteriores, por isso vamos criar uma categoria outros.

Mas costumo dizer que a essência das ERs estão nessa categoria, pois em 90% dos casos reais, usa-se pelo menos um desses caracteres:

- escape ****
- ou **|**
- grupo **( )**
- retrovisor **\n**

Escape: a criptonita \

Você tem duas formas de casar um curinga dentro de uma ER:

Para CPF: `[0-9]{3}[.][0-9]{3}[.][0-9]{3}-[0-9]{2}`

Ou: `[0-9]{3}\.[0-9]{3}\.[0-9]{3}-[0-9]{2}`

Isto é, a contrabarra `(\)` "escapa" qualquer metacaractere, tirando todos os seus poderes.

O escape é tão poderoso que pode escapar a si próprio! O `\\` casa uma barra invertida `(\)` literal.

Ou um ou outro → |

Frequentemente você se depara com um problema quando está escrevendo uma ER: um determinado ponto pode ter mais de um valor. Nesses casos se usa a barra vertical `(|)`, para especificar essa alternância

`flamengo|mengo|mengão`

O *ou*, representado pela barra vertical `(|)`. Essa ER se lê: "`ou flamengo, ou mengo, ou mengão`". mas não se esqueça que a lista também é uma espécie de *ou* `(|)`, mas apenas para uma letra, então:

`[MPST]oda` é o mesmo que `Moda|Poda|Soda|Toda`

Dica:
Breve você verá que `(fla)?meng(o|ão)` casa com `mengo`, `mengão`, `flamengo` e, de quebra, `flamengão`

Grupo: a patota → (...)

Como diz o nome, sua função é fazer grupamentos no interior dos parênteses. Dentro de um grupo podemos ter um ou mais caracteres, metacarateres 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, e pode parecer estranho, mas é essa limitação que lhe dá mais poder.

Grupo: a patota → (...)

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:

`(meta)?caractere`	`caractere, metacaractere`
`(me+ngo,? ?)+`	`mengo, meengo, meeengo, ...`
`(www\.)?zz\.com`	`www.zz.com, zz.com`

E em especial nosso amigo *ou* `(|)` ganha limites e seu poder cresce:

`boa-(tarde|noite)`	`boa-tarde, boa-noite`
`(fla)?meng(o|ão)`	`flamengo, mengo, mengão, flamengão`
`(in|con)?certo`	`incerto, concerto, certo`

Grupo: a patota → (...)

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
```

Ei! E se tivesse **`minimercado`** também?

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

Para listar endereços *mac* do `ifconfig` (3ª Versão)

```
$ ifconfig | grep -E '([[:xdigit:]]{2}:){5}[[:xdigit:]]{2}'
eth0  Link encap:Ethernet  Endereço de HW 1c:75:08:b0:75:76
wlan0 Link encap:Ethernet  Endereço de HW 88:9f:fa:61:9b:10

$ ifconfig | grep -Eo '([[:xdigit:]]{2}:){5}[[:xdigit:]]{2}'
1c:75:08:b0:75:76
88:9f:fa:61:9b:10
```

Retrovisor: o saudosista \1 ... \9

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

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 texto, e não a ER.

Como exemplo, em um texto, procuramos `quero-quero`. Podemos procurar literalmente por `quero-quero`, mas assim não tem graça, vamos ver 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.

Retrovisor: o saudosista \1 ... \9

Mas esse `\1` não é o tal do escape?

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 9 retrovisores por ER, então `\10` é o retrovisor número `1` seguido de um zero.

Retrovisor: o saudosista \1 ... \9

O verdadeiro poder do retrovisor é quando não sabemos exatamente qual texto o grupo casará. Vamos estender o quero do exemplo anterior para "qualquer palavra":

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

Viu 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
```

Com uma modificaçãozinha, fazemos um minicorretor ortográfico para procurar por palavras repetidas como estas, em um texto:

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

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

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

Retrovisor: o saudosista \1 ... \9

Para aplicar o que vimos no slide anterior, vamos ver o arquivo `quero`:

```
$ column -c70 quero
quero-quero   batebate      rebate-bate      bole--bole
quero quero   queroquero    rebate-bateria
come-come     comecome      hoje-joia
bate-bate     bate-bateria  come come
$ grep -E '([[:alpha:]]+) \1' quero  | column -c70
quero quero   come come
$ grep -E '([[:alpha:]]+)[ -]\1' quero | column -c70
quero-quero   come-come     bate-bateria      rebate-bateria
quero quero   bate-bate     rebate-bate       come come
$ grep -E '^([[:alpha:]]+)[ -]\1$' quero | column -c70
quero-quero   come-come     come come
quero quero   bate-bate
$ grep -E '^([[:alpha:]]+)[ -]?\1$' quero | column -c70
quero-quero   come-come     batebate          comecome
quero quero   bate-bate     queroquero        come come
```

Retrovisor: o saudosista \1 ... \9

Vamos ver alguns exemplos, usando o `sed`

```
$ sed -r 's/.*(((band)eira)nte).*/\1 \2 \3a/' <<< "O bandeirante empunhava uma bandeira na banda"
bandeirante bandeira banda
$ sed -r 's/.*((lenta)(mente))/\1 é \3 \2/' <<< "O carro ia lentamente"
lentamente é mente lenta
$ sed -r 's/[a-z]+( .* )[a-z/1ª Palavra\1Nª palavra/'
<< < "arremedo de uma frase que termina na pontuação."
1ª Palavra de uma frase que termina na Nª palavra
```

Alguns Exemplos Úteis

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]
```

Alguns Exemplos Úteis

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]|[12][0-9]|3[01])/(0[1-9]|1[012])/[12][0-9]{3}
```

Alguns Exemplos Úteis

Números

```
[0-9]
[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})?
```

Alguns Exemplos Úteis

telefones

```
....-....
[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}
```

Programas Úteis

O programa **txt2regex** (http://txt2regex.sourceforge.net) é um assistente ("wizard") que constrói ERs em que 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.

E para fechar, 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

Listas Úteis

- http://br.groups.yahoo.com/group/shell-script/
- http://groups.yahoo.com/group/sed-br
- http://groups.yahoo.com/group/perl-br
- http://groups.yahoo.com/group/vi-br
- http://groups.yahoo.com/group/emacs-br
- http://groups.yahoo.com/group/dsjava
- http://groups.yahoo.com/group/listadeweb
- http://lists.allfinder.com.br/php

Aprender a programar em Shell?

http://www.julioneves.com

Um livro livre e on-line

Julio Cezar Neves

julioneves@openoffice.org