Página XVI, penúltimo parágrafo, trocar "Processamanto" por "Processamento";

Página 20, 1º parágrafo, trocar "165571" por "165751";

Página 30, 4º parágrafo, trocar "/n" por "\n";

Página 37, antes da seção "sort – classifica dados" incluir:

Na lista *Shell-Script* do Yahoo, um cara perguntou:

– Gente estou fazendo um *script* e me deparei com um problema assim:

```
$ num_terminal=123456789
$ echo $num_terminal | wc -c
10
```

– Mas como assim, se a cadeia só tem 9 caracteres e ele sempre conta um a mais. Porque isso acontece?

E outro colega da lista respondeu:

– Isso aconteceu, porque o `wc` contou o `\n` que o `echo` adiciona ao final da cadeia. Para o cálculo certo use a opção `-n` do comando `echo,` que não insere o `\n.`

Nisso, um outro entrou na discussão e disse:

– Ainda sobre o tema, veja um exemplo onde isso falha:

```
$ TESTE=ç
$ echo $TESTE | wc -c
3
$ echo -n $TESTE | wc -c
2
$ echo -n $TESTE | wc -m
1
```

– Como foi visto, o `-m` conta o número de caracteres, já o `-c` conta o número de *bytes*. Claro que temos que remover o `\n` também, como os outros colegas haviam falado.

  Mais ainda: na verdade caracteres em UTF-8 podem usar de 1 a 4 *bytes*. Um para os primeiros 127 (ASCII), 2 para latim (acentuados), 3 para caracteres em outros idiomas (como japonês, chinês, hebraico, etc) e 4 *bytes* para caracteres que ainda não foram definidos, mas que futuramente serão colocados na tabela Unicode.

Engraçado é que a forma

```
$ wc -c <<< 123456789
10
$ wc -m <<< 123456789
10
```

Retorna com a quebra de linha também (talvez pelo sinal de fim de instrução), mas se você fizer:

```
$ wc -c <<< 123456789; echo
10
```

Ele retorna com a quebra mesmo assim... É no mínimo engraçado, porque usando o *here string* `(<<<),` não vejo como chegar ao resultado correto...

E, para terminar esta discussão, apareceu outro colega que disse:

– Uma curiosidade: o `wc` tem a opção `-L` que imprime o comprimento da linha mais longa de um texto. se este texto tiver somente uma linha, o resultado coincidira com o tamanho desta. Veja

```
$ echo 123456789 | wc -L
9
```

```
$ wc -L <<< 123456789
9
```

Página 40 trocar **-n, -l, --lines =numero**" por "**-n, -l, --lines=numero**", tirando o espaço entre **lines** e **=;**

Pagina 88 no parágrafo relativo ao Bourne Shell trocar "adptado" por "adaptado";

Página 115, antes do título no início da página, inserir:

A opção -r

Se eu tenho uma data no formato dia/mês/ano, ou seja **dd/mm/aaaa** e desejo passá-la para o formato **aaaa/mm/dd,** eu deveria fazer:

```
$ sed 's/^\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3\/\2\/\1/'
<<< 31/12/2009
2009/12/31
```

Funcionou, mas a legibilidade disso está um horror! Aí tem mais contrabarra **(\)** que qualquer outra coisa!

Justamente para facilitar a montagem das *Expressões Regulares* e sua legibilidade é que o **GNU-sed** tem a opção **-r.** Ela avisa ao **sed** que serão usados *metacaracteres* avançados, que desta forma se encarrega das suas interpretações, não deixando-os para a interpretação do *Shell*.

Esse mesmo **sed** poderia (e deveria) ser escrito da seguinte forma:

```
$ sed -r 's/^([0-9]{2})\/([0-9]{2})\/([0-9]{4})$/\3\/\2\/\1/' <<<
31/12/2009
2009/12/31
```

Melhorou mas ainda não está bom, porque ainda existem contrabarras **(\)** "escapando" as barras **(/)** da data para que o **sed** não as confunda com as barras **(/)** separadoras de campo. Para evitar esta confusão, basta usar outro caractere, digamos o hífen **(-),** como separador, Vejamos como ficaria

```
$ sed -r 's-^([0-9]{2})/([0-9]{2})/([0-9]{4})$-\3/\2/\1-' <<< 31/12/2009
2009/12/31
```

Agora ficou mais fácil de ler e podemos decompor a *Expressão Regular* para destrinchá-la. Vamos lá:

1ª Parte - **^([0-9]{2})/([0-9]{2})/([0-9]{4})$.** Vamos dividi-la:

^([0-9]{2}) - A partir do início **(^)** procuramos dois **({2})** algarismos **([0-9]).** Isso casa com o dia;

/([0-9]{2})/ - Idêntica à anterior, porém por estar entre barras **(/),** casará com o mês;

**([0-9]{4})$** - procuramos quatro **({4})** algarismos **([0-9])** no fim ($). Isso casa com o ano;

2ª Parte - **\3/\2/\1**

Repare que as pesquisas de dia, mês e ano foram colocados entre parênteses. Como os **textos** casados pelas *Expressões Regulares* no interior dos parênteses são guardados para uso futuro, os retrovisores **\3,\2** e **\1** foram usados para recuperar ano, o mês e o dia, respectivamente.

Página 119, antes da seção "**Os comandos para corta e colar**", incluir:

A opção -f (*file*)

Esta opção é muito interessante, pois ela pesquisa um arquivo, procurando por *Expressões Regulares* contidas em outro arquivo, que será definido pela opção -f.

O bacana desta opção é que ela evita 2 laços (*loops*): um de leitura do arquivo de dados e outro de leitura do arquivo a ser pesquisado.

Exemplos:

O meu problema é listar os registros de **/etc/passwd** dos usuários que estão no arquivo **usus.** Veja a cara dele:

```
$ cat usus
bin
irc
root
sys
uucp
```

Nas linguagens convencionais, eu teria de fazer um programa que lesse cada registro de **usus** e o pesquisasse linha-a-linha em **/etc/passwd,** já que **usus** está em ordem alfabética e **/etc/passwd** está na sequência do cadastramento.

Aqui eu só tenho um probleminha: no **/etc/passwd** tem **bin** em profusão, devido aos usuários que usam **/bin/bash** como *Shell* padrão (*default*). Temos então de mudar **usus** para que a pesquisa seja feita somente no início do **/etc/passwd.**

Resolvido este problema, surge outro, já que a pesquisa pelo usuário **sys** localizará no início da linha, além do **sys** propriamente dito, também o **syslog.** Então temos mudar **usus** para delimitar também o fim do campo, o que faremos colocando um dois-pontos **(:)** ao final de cada registro. Veja como eu mato 2 coelhos com uma só "**sed**ada". ;):

```
$ sed -i 's/^/\^/;s/$/:/' usus
```

No duro fiz um dois-em-um: no primeiro comando **s** do **sed,** substituí o início de cada registro **(^)** por um circunflexo (que está "escapado" para para ser interpretado como um literal e não como um *metacaractere* como o anterior. No segundo comando **s,** substituí o final de cada registro por um dois-pontos **(:).** A opção **-i** foi usada para que a saída editada fosse no mesmo arquivo. Veja como ele ficou:

```
$ cat usus
^bin:
^irc:
^root:
^sys:
^uucp:
```

Agora é mole, basta executar o **grep,** veja só:

```
$ grep -f usus /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
irc:x:39:39:ircd:/var/run/ircd:/bin/sh
```

A opção -o (*Only matching*)

Com esta opção o **grep** devolve somente a parte que casou com a *Expressão Regular* especificada. Esta opção é ótima para depurar *Expressões Regulares*.

Exemplos:

```
$ grep -o 'a.*a' <<< batatada
atatada
$ grep -o 'a.*a' <<< batatuda
atatuda
```

Não era isso que você esperava? Lembre-se que o asterisco `(*)` é guloso (veja apêndice sobre *Expressões Regulares*).

Antes ou depois

Algumas vezes, temos um arquivo que é formado por diversos tipos de registros. Por exemplo: um cadastro de pessoal tem pelo menos os seguintes tipos de registro:

- Dados pessoais;
- Dados profissionais;
- Histórico de pagamentos

Entre outros. É para casos assim que existem duas opções do grep:

`-A n` (`A` de *After* – depois) - pega além da linha que casou com a *Expressão Regular*, `n` linhas após;

`-B n` (`B` de *Before* – Depois) - pega além da linha que casou com a *Expressão Regular*, `n` linhas antes.

Exemplos:

```
$ ifconfig | grep -A7 '^wlan0'
wlan0  Link encap:Ethernet  Endereço de HW 08:28:58:04:e1:53
       inet end.: 192.168.2.2  Bcast:192.168.2.5  Masc:255.255.255.0
       endereço inet6: fe80::252:58ff:fe04:e153/64 Escopo:Link
       UP BROADCAST RUNNING MULTICAST  MTU:1500  Métrica:1
       pacotes RX:28 erros:0 descartados:0 excesso:0 quadro:0
       Pacotes TX:80 erros:0 descartados:0 excesso:0 portadora:0
       colisões:0 txqueuelen:1000
       RX bytes:4170 (4.1 KB) TX bytes:12324 (12.3 KB)
```

Usando o exemplo anterior, suponha agora que eu queira saber o meu endereço IP. Como este está situado na linha seguinte à começada por `wlan0` (se eu estiver via *wireless*, óbvio), posso fazer:

```
$ ifconfig | grep -A1 '^wlan0'
wlan0   Link encap:Ethernet  Endereço de HW 00:25:56:04:e1:53
        inet end.: 192.168.2.2  Bcast:192.168.2.5  Masc:255.255.255.0
```

Como só interessa o endereço IP, vamos mandar essa saída para uma *Expressão Regular* para filtrar estes campos:

```
$ ifconfig | grep -A1 '^wlan0' | grep -Eo '[0-9]{1,3}\.[0-9]{1,3}\.[0-9]
{1,3}\.[0-9]{1,3}'
192.168.2.2
192.168.2.5
255.255.255.0
```

Como você viu a filtragem foi feita e a opção `-o` fez com que fosse para a saída apenas o que casou com a *Expressão Regular*. Mas eu não quero o meu IP de *broadcast* nem a máscara. Só me interessa o primeiro. Estão vamos pegar somente a primeira linha da saída:

```
$ ifconfig | grep -A1 '^wlan0'| grep -Eo '[0-9]{1,3}\.[0-9]{1,3}\.[0-9]
{1,3}\.[0-9]{1,3}' | head -1
192.168.2.2
```

Um conselho: caso você não tenha entendido esta *Expressão Regular*, neste livro tem um apêndice que fala somente sobre o assunto. Vale a pena dar uma olhada, pois *Expressões Regulares* são usadas por 90% das linguagens de programação e quando você conhecê-las, elas passarão a ter uma importância enorme na sua vida profissional. Vale muito a pena!

Pagina 121 acima da palavra "Saudacoes" o texto do e-mail está, "... que que ...". Retirar um "que ";

Na página 134, antes do parágrafo "Uma forma bacana..." colocar o seguinte título:

bc – A calculadora

Página 135 no parágrafo "Se eu mandasse desse jeito..." na linha de baixo ta escrito daque sinal, trocar por daquele sinal.

Na página 137, após o primeiro parágrafo, colocar o seguinte título:

O interpretador aritmético do Shell

Página 140, no alto antes de "Voltando à vaca fria..." inserir:

Esse tipo de construção **`(${...})`** que acabamos de ver, é genericamente chamada de Expansão de Parâmetro (*Parameter Expansion*).

Encorajo muito seu uso, apesar da perda de legibilidade do código, porque elas são intrínsecas do *Shell* (*builtins*) e por isso são, no mínimo, umas 100 vezes mais velozes que as suas similares.

Veremos a Expansão de Parâmetros de forma muito mais detalhada no capítulo 7.

Página 140, antes do parágrafo: "Para encontrar um caractere em uma cadeia, usamos a sintaxe:", incluir:

Essa Expansão de Parâmetros que acabamos de ver, também pode extrair uma sub cadeia, do fim para o principio, desde que seu segundo argumento seja negativo.

```
$ echo ${TimeBom: -5}
mengo
$ echo ${TimeBom:-5}
Flamengo
$ echo ${TimeBom:(-5)}
mengo
```

Existe outra Expansão de Parâmetros, que veremos mais tarde, que tem a sintaxe **`${parm:-Valor}`** e por isso não podemos colar o hífen **`(-)`** ao dois-pontos **`(:)`**. Como vimos podemos usar um espaço em branco ou até mesmo usar parênteses para separar os dois sinais.

Página 179 aparece o código: **`if [[ $Hora =~ '([01][0-9]|2[0-3]):[0-5][0-9]' ]]`** duas vezes. Em ambas retirar os apóstrofos.

Na página 180, antes da seção O caso que o case casa melhor, inserir:

Agora que vocês entenderam o uso desta variável, vou mostrar uma de suas grandes utilizações, já que aposto como não perceberam que enrolei vocês desde o primeiro exemplo desta seção. Para isso, vamos voltar ao exemplo da crítica de horas, porém mudando o valor da variável **`$Hora`**. Veja só como as *Expressões Regulares* podem nos enganar:

```
$ Hora=54321:012345
```

```
$ if  [[ $Hora =~ ([01][0-9]|2[0-3]):[0-5][0-9] ]]
> then
>     echo Horario OK
> else
>     echo O horario informado esta incorreto
> fi
Horario OK
```

Epa! Isso era para dar errado! Vamos ver o que aconteceu:

```
$ echo ${BASH_REMATCH[0]}
21:01
```

Ihhh, casou somente com os dois caracteres que estão antes e os dois que estão depois dos dois-pontos **(:).** Viu como eu disse que tinha te enrolado?

Para que isso ficasse perfeito faltou colocar as âncoras das *Expressões Regulares*, isto é, um circunflexo **(^)** para marcar o início e um cifrão **($)** para marcar o final. Veja:

```
$ Hora=54321:012345
$ if  [[ $Hora =~ ^([01][0-9]|2[0-3]):[0-5][0-9]$ ]]
> then
>     echo Horario OK
> else
>     echo O horario informado esta incorreto
> fi
O horario informado esta incorreto
```

Esse erro é muito comum e dei destaque nele, porque publiquei errado na minha *home page* **(http://www.julioneves.com)** e assim permaneceu por mais de 6 meses sem que ninguém notasse, e olha que a página tem, em média, 30.000 acessos/mês.

Página 183, antes dos exercícios

O bash 4.0 introduziu duas novas facilidades no comando **case.** A partir desta versão, existem mais dois terminadores de bloco além do **;;**, que são:

;;& - Quando um bloco de comandos for encerrado com este terminador, o programa não sairá do **case,** mas testará os próximos padrões;

;& - Neste caso, o próximo bloco será executado, sem sequer testar o seu padrão.

Exemplos:

Suponha que no seu programa possam ocorrer 4 tipos de erro e você resolva representar cada um deles por uma potência de 2, isto é 2^0=**1**, 2^1=**2**, 2^2=**4**, 2^3=**8**, de forma que a soma dos erros ocorridos gerem um número único para representá-los (é assim que se formam os números binários). Assim, se ocorrem erros dos tipos 1 e 4, será passado 5 (4+1) para o programa, se os erros forem 1, 4 e 8, será passado 13 (8+4+1). Observe a tabela a seguir:

Soma	Erros			
	8	4	2	1
8	x	-	-	-
7	-	x	x	x
6	-	x	x	-
5	-	x	-	x
4	-	x	-	-
3	-	-	x	x
2	-	-	x	-
1	-	-	-	x
0	-	-	-	-

$ cat case.sh

```
#!/bin/bash
#  Recebe um código formado pela soma de 4 tipos
#+ de erro e dá as msgs correspondentes. Assim,
#+ se houveram erros tipo 4 e 2, o script receberá 6
#+ Se os erros foram 1 e 2, será passado 3. Enfim
#+ os códigos de erro seguem uma formação binária.

Bin=$(bc <<< "obase=2; $1")        Passa para binário
Zeros=0000
Len=${#Bin}                        Pega tamanho de $Bin
Bin=${Zeros:$Len}$Bin              Preenche com zeros à esquerda
#  Poderíamos fazer o mesmo que foi feito acima
#+ com um cmd printf, como veremos no capítulo 6
case $Bin in
    1[01][01][01]) echo Erro tipo 8;;&
    [01]1[01][01]) echo Erro tipo 4;;&
    [01][01]1[01]) echo Erro tipo 2;;&
    [01][01][01]1) echo Erro tipo 1;;&
             0000) echo Não há erro;;&
                *) echo Binário final: $Bin
esac
```

Repare que todas as opções serão testadas para saber quais são *bits* ligados (zero=desligado, um=ligado). No final aparece o binário gerado para que você possa comparar com o resultado. Testando:

```
$ case.sh 5
Erro tipo 4
Erro tipo 1
Binário final: 0101
$ case.sh 13
Erro tipo 8
Erro tipo 4
Erro tipo 1
Binário gerado: 1101
```

Veja também este fragmento de código adaptado de http://tldp.org/LDP/abs/html/bashver4.html, que mais parece uma continuação do **testchar** que acabamos de ver.

```
case "$1" in
    [[:print:]] )  echo $1 é um caractere imprimível;;&
    # O terminador ;;& testará o próximo padrão
    [[:alnum:]] )  echo $1 é um carac. alfa/numérico;;&
    [[:alpha:]] )  echo $1 é um carac. alfabético   ;;&
    [[:lower:]] )  echo $1 é uma letra minúscula    ;;&
    [[:digit:]] )  echo $1 é um caractere numérico  ;&
    # O terminador ;& executará o próximo bloco...
    %%%@@@@@    )  echo "************************"  ;;
#   ^^^^^^^^  ... mesmo com um padrão maluco.
esac
```

Sua execução passando 3 resultaria:

```
3 é um caractere imprimível
3 é um carac. alfa/numérico
3 é um caractere numérico
********************************
```

Passando m:

```
m é um caractere imprimível
m é um carac. alfa/numérico
m é um carac. alfabético
m é uma letra minúscula
```

Passando / :

```
/ é um caractere imprimível
```

Pagina 186, antepenúltimo parágrafo, trocar a frase "Nestes comandos usei a opção -s " " para que o separador entre o números ...",por "Nestes comandos usei a opção -s " " para que o separador entre o**s** números ...";

Pagina 191 na observação (que está à direita) colocada no resultado do comando "echo "$IFS" | od -h" trocar "hd" por "od -h";

Pagina 227, último parágrafo, trocar "ksh" por "bash" e "PID=23188" por "PID=4418";

Página 241, quase no final, após o exemplo $ mv $Arq ${Arq}1, incluir:

- ${!parâmetro}

Isto equivale a uma indireção, ou seja devolve o valor apontado por uma variável cujo nome está armazenada em parâmetro.

Exemplo:

```
$ Ponteiro=VariavelApontada
$ VariavelApontada="Valor Indireto"
$ echo "A variável \$Ponteiro aponta para \"$Ponteiro\"
> que indiretamente aponta para \"${!Ponteiro}\""
A variável $Ponteiro aponta para "VariavelApontada"
que indiretamente aponta para "Valor Indireto"
```

- ${!parâmetro@}
- ${!parâmetro*}

Ambas expandem para os nomes das variáveis prefixadas por parâmetro. Não notei nenhuma diferença no uso das duas sintaxes.

Exemplos:

Vamos listar as variáveis do sistema começadas com a cadeia **GNOME:**

```
$ echo ${!GNOME@}
GNOME_DESKTOP_SESSION_ID GNOME_KEYRING_PID GNOME_KEYRING_SOCKET
$ echo ${!GNOME*}
GNOME_DESKTOP_SESSION_ID GNOME_KEYRING_PID GNOME_KEYRING_SOCKET
```

- ${parâmetro:valor}

Na página 242 antes do 2o. Parágrafo "A forma convencional ...", inserir:

Para entender a diferença de uso entre ${parâmetro:valor} e ${parâmetro:-valor}, vamos ver os exemplos a seguir:

Exemplos:

```
$ unset var                              var não existe
$ echo ${var:-Variável não definida}
Variável não definida
$ echo ${var-Variável não definida}
Variável não definida
$ var=                                   var tem valor nulo (como var="")
$ echo ${var:-Variável não definida}
Variável não definida
$ echo ${var-Variável não definida}

$ var=1                                  var é igual a 1
$ echo ${var:-Variável não definida}
1
$ echo ${var-Variável não definida}
1
```

Como você pode ver, a diferença de uso somente ocorre quando o parâmetro que está sendo expandido tem valor nulo.

Note também que a Expansão de Parâmetros por si só não altera o valor do parâmetro, mas somente manda a substituição para a saída.

Vou repetir o exemplo dado, porém agora alterando o valor do parâmetro (variável **$User**), mas para isso entenda a nova Expansão de Parâmetros a seguir:

- ${parâmetro:=valor}
- ${parâmetro=valor}

Idêntico ao anterior, porém quando usado desta forma, a expansão é feita para o valor da variável.

Para entender melhor, vamos fazer o exemplo anterior usando esta sintaxe:

```
read -p"Login Name em $Site ($LOGNAME): " User
echo ${User:=$LOGNAME}
```

Após a execução deste código, se fizermos

```
echo $User
```

Teremos como resposta o campo teclado durante o **read** ou o conteúdo da variável de sistema **$LOGNAME,** caso o valor oferecido (*default*) tenha sido aceito, isto é, não foi necessário fazer a atribuição do valor.

No caso de não desejar que a mensagem vá para a tela, troque o echo por let.

```
read -p"Login Name em $Site ($LOGNAME): " User
let ${User:=$LOGNAME}
```

Assim como o anterior, este também tem duas formas de uso, mas as observações feitas para o outo, também valem para esse.

- `${parâmetro:?valor}`

Parecido com o anterior pois atua somente nos casos em que o parâmetro é nulo ou não está declarado, quando então mandará uma mensagem para a saída de erro padrão.

Veja esse trecho de código:

```
read -p "Deseja continuar? " Responda
echo ${Responda:?(S)im ou (N)ão}
```

Caso a seja teclado **<ENTER>** sem informar nenhuma resposta, isto é, caso a variável **$Responda** permaneça vazia, a mensagem abaixo será exibida:

```
bash: Responda: (S)im ou (N)ão
```

Página 246, antes do quadro de dicas, inserir:

- `${parâmetro^}`
- `${parâmetro,}`

Essas expansões foram introduzidas a partir do Bash 4.0 e modificam a caixa das letras do texto que está sendo expandido. Quando usamos circunflexo **(^),** a expansão é feita para maiúsculas e quando usamos vírgula **(,),** a expansão é feita para minúsculas.

Exemplo:

```
$ Nome="botelho"
$ echo ${Nome^}
Botelho
$ echo ${Nome^^}
BOTELHO

$ Nome="botelho carvalho"
$ echo ${Nome^}
Botelho carvalho                    Que pena...
```

Um fragmento de *script* que pode facilitar a sua vida:

```
read -p "Deseja continuar (s/n)? "
[[ ${REPLY^} == N ]] && exit
```

Esta forma evita testarmos se a resposta dada foi um **N** (maiúsculo) ou um **n** (minúsculo).

No *Windows*, além dos vírus e da instabilidade, também são frequentes nomes de arquivos com espaços em branco e quase todos em maiúsculas. No exemplo anterior, vimos como trocar os espaços em branco por sublinha **(_),** no próximo veremos como passá-los para minúsculas:

```
$ cat trocacase.sh
#!/bin/bash
#  Se o nome do arquivo tiver pelo menos uma
#+ letra maiúscula, troca-a para minúscula

for Arq in *[A-Z]*                  Pelo menos 1 minúscula
do
    if  [ -f "${Arq,,}" ]           Arq em minúsculas já existe?
    then
        echo ${Arq,,} já existe
else
    mv "$Arq" "${Arq,,}"
    fi
```

done

Na página 247, no 2o. parágrafo, trocar "porém os nomes de ... existem." por porém o que for gerado, não tem nada a ver com arquivo. Elas são usadas para gerar cadeias arbitrárias, produzindo todas as combinações possíveis, levando em consideração os prefixos e sufixos.

Existiam 5 sintaxes distintas, porém o Bash 4.0 incorporou uma 6ª. Elas são escritas da seguinte forma:

1. `{lista},` onde `lista` são cadeias separadas por vírgulas;
2. `{inicio..fim};`
3. `prefixo{****},` onde os asteriscos `(****)` podem ser substituídos por `lista` ou por um par `inicio..fim;`
4. `{****}sufixo,` onde os asteriscos `(****)` podem ser substituídos por `lista` ou por um par `inicio..fim;`
5. `prefixo{****}sufixo, onde os asteriscos (****) podem ser substituídos por lista ou por um par inicio..fim;`
6. `{inicio..fim..incr},` onde `incr` é o incremento (ou razão, ou passo). Esta foi introduzida a partir do Bash 4.0.

Na página 247, após a última linha, inserir os seguintes exemplos:

```
$ echo {1..A}                          Letra e número não funfa
{1..A}
$ echo {0..15..3}                      Incremento de 3, só no Bash 4
0 3 6 9 12 15
$ echo {G..A..2}                       Incr de 2 decresc, só no Bash 4
G E C A
$ echo {000..100..10}                  Zeros à esquerda, só no Bash 4
000 010 020 030 040 050 060 070 080 090 100
```

Página 248, após os exemplos e antes da seção, incluir o seguinte:

```
$ eval \>{a..c}.{ok,err}\;             Veja o eval no 8º capítulo
$ ls ?.*
a.err  a.ok  b.err  b.ok  c.err  c.ok
```

A sintaxe deste último exemplo pode parecer rebuscada, mas substitua o `eval` por `echo` e verá que aparece:

```
$ echo \>{a..c}.{ok,err}\;
>a.ok; >a.err; >b.ok; >b.err; >c.ok; >c.err;
```

Ou seja o comando para o Bash criar os 6 arquivos. A função do `eval` é executar este comando que foi montado.

O mesmo pode ser feito da seguinte maneira:

```
$ touch {a..z}.{ok,err}
```

Mas no primeiro caso, usamos Bash puro, o que torna esta forma pelo menos 100 vezes mais rápida que a segunda que usa um comando externo `(touch).`

Na página 249, após Obrigado Tiago Peczenyj, incluir:

O Tiago me deu as dicas acima, mas existem outras... Teste as `opcoes` a seguir dentro do comando `shopt -s <opcoes>.`

cdspell

Com esta opção setada, pequenos erros de digitação no nome do diretório para o qual

você deseja fazer um `cd` serão ignorados, isto é, caracteres extras ou esquecidos serão automaticamente corrigidos sem necessidade de redigitação.

No exemplo a seguir, queria fazer um `cd rede,` que é um diretório abaixo do meu *home*.

```
$ cd red
bash: cd: red: Arquivo ou diretório inexistente
$ shopt -s cdspell
$ cd red
rede
$ cd -
/home/jneves
$ cd ede
rede
$ cd -
/home/jneves
$ cd redes
rede
```

Já viu que essa é uma boa opção para você ter no seu `.bashrc,` não é?

cmdhist

Essa opção é bacana, pois transforma comandos criados em diversas linhas (como um `for` ou um `while,` por exemplo) em uma única linha, com os comandos separados por ponto-e-vírgula `(;).` Isso é particularmente útil para editar comandos e portanto é uma opção setada por *default* pelo *Bash*.

dotglob

Esta opção permite que nomes de arquivos começados por um ponto (.), conhecidos como "arquivos escondidos", sejam expandidos com os metacaracteres coringa.

```
$ ls *bash*
ls: impossível acessar *bash*: Arquivo ou diretório inexistente
$ shopt -s dotglob
$ ls *bash*
bash_history  .bash_logout  .bashrc
```

Ainda na página 249, substituir o primeiro parágrafo após o título da seção "Vetores ou Arrays" por:

Um vetor ou *array* é um método para se tratar diversas informações (normalmente do mesmo tipo) sob um único nome, que é o nome do vetor. Poderíamos ter por exemplo um vetor chamado cervejas, que armazenasse Skol, Antártica, Polar, ... Para acessarmos estes dados, precisamos de um índice.

Assim, `NomeDoVetor[Indice]` contém um `valor,` ou seja `cervejas[0]` contém `Skol` e `cerveja[2]` contém `Polar.`

Existem 4 formas de se declarar um vetor:

`vet=(el0 el1,...eln)`	Cria o vetor `vet,` inicializando-o com os elementos `el1, el2, …, eln).` Se for usado como `vet=(),` o vetor `vet` será inicializado vazio. Se `vet` já existir, perderá os antigos valores e receberá os novos, ou será esvaziado;
`vet[n]=val`	Cria o elemento índice `n` do vetor `vet` com o valor `val.` Se não existir, o vetor `vet` será criado;
`declare -a vet`	Cria o vetor `vet` vazio. Caso `vet` já exista, se manterá inalterado;

`declare -A vet` Cria um vetor associativo (aquele cujos índices não são numéricos). Esta é a única forma de declarar um vetor associativo, que só é suportado a partir da versão 4.0 do Bash.

Página 253, após o primeiro parágrafo: "Voilà ...", inserir:

Existe uma outra forma de fazer o mesmo. Para mostrá-la vamos montar um vetor esparso, formado por animais de nomes compostos:

```
$ Animais=([2]="Mico Leão" [5]="Galinha d'Angola" [8]="Gato Pardo")
```

Agora veja como podemos listar os índices:

```
$ echo ${!Animais[*]}
2 5 8
$ echo ${!Animais[@]}
2 5 8
```

Estas construções construções listam os índices do vetor, sem diferença alguma na resposta. Assim sendo, podemos escolher uma delas para mostrar também todos os elementos de `Animais,` da seguinte forma:

```
$ for Ind in ${!Animais[@]}          Ind recebe cada um dos índices
> do
>     echo ${Animais[Ind]}
> done
Mico Leão
Galinha d'Angola
Gato Pardo
```

No fim da página 254, inserir:

Primeiro vou te dar um exemplo do que acabei de falar, depois você inventa uns *scripts* para testar outras Expansões de Parâmetros como as provocadas por `%, %%, #, ##, ..., OK?`

Então vamos criar o vetor Frase:

```
$ Frase=(Alshançar o shéu é sensashional. Um sushesso\!)
$ echo ${Frase[*]//sh/c}
Alcançar o céu é sensacional. Um sucesso!
```

Para coroar isso tudo, no dia 01/01/2010, após o primeiro *reveillon* da nova década, estava finalizando o texto para a 8ª edição deste livro quando surgiu na lista shell-script do Yahoo (não deixe de se inscrever porque ela é ótima e lá se aprende muuuuito) um colega perguntando como ele poderia contar a quantidade de cada anilha teria de usar para fazer uma cabeação.

Anilha são aqueles pequenos anéis numerados que você vê nos cabos de rede, que servem para identificá-los. Em outras palavras, o problema era dizer quantas vezes ele iria usar cada algarismo em um dado intervalo. Vejamos a proposta de solução:

```
$ cat anilhas.sh
Tudo=$(eval echo {$1..$2})          Recebe os num. entre $1 e $2
for ((i=0; i<${#Tudo}; i++))
{
    [ ${Tudo:i:1} ] || continue     Espaço entre 2 números
    let Algarismo[${Tudo:i:1}]++    Incrementa vetor do algarismo
}
for ((i=0; i<=9; i++))
{
    printf "Algarismo %d = %2d\n" \
        $i ${Algarismo[$i]:-0}      Se o elemento for vazio, lista zero
}
```

```
$ ./anilhas.sh 12 15
Algarismo 0 =  0
Algarismo 1 =  4
Algarismo 2 =  1
Algarismo 3 =  1
Algarismo 4 =  1
Algarismo 5 =  1
Algarismo 6 =  0
Algarismo 7 =  0
Algarismo 8 =  0
Algarismo 9 =  0
```

Me divirto muito escrevendo um *script* como esse, no qual foi empregado somente Bash puro. Além de divertido, é antes de mais nada, rápido.

Vetores associativos

A partir do Bash 4.0, passou a existir o vetor associativo. Chama-se vetor associativo, aqueles cujos índices são alfabéticos. As regras que valem para os vetores inteiros, valem também para os associativos, porém antes de valorar estes últimos, é obrigatório declará-los.

Exemplo:

```
$ declare -A Animais                    Obrigatório para vetor associativo
$ Animais[cavalo]=doméstico
$ Animais[zebra]=selvagem
$ Animais[gato]=doméstico
$ Animais[tigre]=selvagem
```

É impossível gerar todos os elementos de uma só vez, como nos vetores inteiros. Assim sendo, não funciona a sintaxe:

```
Animais =([cavalo]=doméstico [zebra]=selvagem \
          [gato]=doméstico [tigre]=selvagem)
```

```
$ echo ${Animais[@]}
doméstico selvagem doméstico selvagem
$ echo ${!Animais[@]}
gato zebra cavalo tigre
```

Repare que os valores não são ordenados, ficam armazenados na ordem que são criados, diferentemente dos vetores inteiros que ficam em ordem numérica.

Supondo que esse vetor tivesse centenas de elementos, para listar separadamente os domésticos dos selvagens, poderíamos fazer um *script* assim:

```
$ cat animal.sh
#!/bin/bash
# Separa animais selvagens e domésticos
declare -A Animais
Animais[cavalo]=doméstico               Criando vetor para teste
Animais[zebra]=selvagem                 Criando vetor para teste
Animais[gato]=doméstico                 Criando vetor para teste
Animais[tigre]=selvagem                 Criando vetor para teste
Animais[urso pardo]=selvagem            Criando vetor para teste
for Animal in "${!Animais[@]}"          Percorrendo vetor pelo índice
do
```

```
    if  [[ "${Animais[$Animal]}" == selvagem ]]
    then
        Sel=("${Sel[@]}" "$Animal")   Gerando vetor p/ selvagens
    else
        Dom=("${Dom[@]}" "$Animal")   Gerando vetor p/ domésticos
    fi
done
#  Operador condicional, usado para descobrir qual
#+ vetor tem mais elementos. Veja detalhes na seção
#+ O interpretador aritmético do Shell
Maior=$[${#Dom[@]}>${#Sel[@]}?${#Dom[@]}:${#Sel[@]}]
clear
tput bold; printf "%-15s%-15s\n" Domésticos Selvagens; tput sgr0
for ((i=0; i<$Maior; i++))
{
    tput cup $[1+i]  0; echo ${Dom[i]}
    tput cup $[1+i] 14; echo ${Sel[i]}
}
```

Gostaria de chamar a sua atenção para um detalhe: neste *script* me referi a um elemento de vetor associativo empregando **`${Animais[$Animal]}`** ao passo que me referi a um elemento de um vetor inteiro usando **`${Sel[i]}`**. Ou seja, quando usamos uma variável como índice de um vetor inteiro, não precisamos prefixá-la com um cifrão **`($)`**, ao passo que no vetor associativo, o cifrão **`($)`** é obrigatório.

Lendo um arquivo para um vetor

Ainda falando do Bash 4.0, eis que ele surge com uma outra novidade: o comando intrínseco (*builtin*) **`mapfile`**, cuja finalidade é jogar um arquivo de texto inteiro para dentro de um vetor, sem *loop* ou substituição de comando

- EPA! Isso deve ser muito rápido!
- E é. Faça os teste e comprove!

Exemplo:

```
$ cat frutas
abacate
maçã
morango
pera
tangerina
uva
$ mapfile vet < frutas            Mandando frutas para vetor vet
$ echo ${vet[@]}                  Listando todos elementos de vet
abacate maçã morango pera tangerina uva
```

Obteríamos resultado idêntico se fizéssemos:

```
$ vet=($(cat frutas))
```

Porém isso seria mais lento porque a substituição de comando é executada em um *subshell*.

Uma outra forma de fazer isso que logo vem à cabeça é ler o arquivo com a opção **`-a`** do comando **`read`**. Vamos ver como seria o comportamento disso:

```
$ read -a vet < frutas
$ echo ${vet[@]}
abacate
```

Como deu para perceber, foi lido somente o primeiro registro de **frutas.**

Nas páginas 257 e 258, a expressão:

```
i='expr $i + 1'
```

ocorre 4 vezes. Trocar todas por:

```
let i++
```

Página 259, antes da seção wait a minute Mr. Postman, incluir:

Um dos participantes da lista de *Shell-script* do Yahoo, disse que tinha um arquivo chamado **dados.txt,** cuja linha era assim:

```
Model=Samsung 0411N, Serial=00000005464, blablabla=asdadqddq
```

Porém deste, só interessava o nome do modelo **(Samsung 0411N).**

As soluções apresentadas foram as mais variadas possíveis (como todas daquela excelente lista, que aconselho a todos se inscreverem). Veja só algumas, para que sirvam como exemplo:

```
$ cut -f1 -d, dados.txt | cut -f2 -d=
```

Só mais umazinha:

```
$ sed -n 's/Model=\([^,]*\).*$/\1/p' dados.txt
```

O Fabiano Caixeta Duarte, grande conhecedor de Bash, deu a seguinte solução, desculpando-se pela mesma ser deselegante:

```
$ Linha=$(<dados.txt)
$ Linha=${Linha// /_}
$ eval ${Linha%%,*}
$ echo ${Model//_/ }
Samsung 0411N
```

Escrevi imediatamente para a lista, dizendo que a solução não tinha nada de deselegante, muito pelo contrário, pois usava Bash puro e que, por isso, deveria ser muito mais eficiente e veloz que todas as soluções apresentadas. Vou explicá-la linha-a-linha:

linha 1 - Carrega todo o arquivo **dados.txt** para a variável **$Linha;**

linha 2 - Para não ter problemas na linha 3, esta substituição de parâmetros troca os espaços por sublinha **(_);**

linha 3 - Esta substituição de parâmetros despreza tudo à direita da primeira virgula **(,).** Observe o que acontece se trocarmos o **eval** por **echo:**

```
$ echo ${Linha%%,*}
Model=Samsung_0411N
```

Ou seja: uma atribuição, porém com um sublinha **(_)** onde havia um espaço em branco. O **eval,** efoi usado para que esta atribuição fosse feita. Veja este teste após a execução desta linha:

```
$ echo $Model
Samsung_0411N
```

linha 4 - Pronto! Agora só falta trocar o sublinha **(_)** por um espaço em branco, e foi isso que a substituição de parâmetros desta linha fez.

Na página 275 antes do título da seção "Script também é um comando", inserir:

Coprocessos

A partir da versão 4.0, o Bash incorporou o comando **coproc.** Este novo intrínseco (*builtin*) permite dois processos assíncronos se comunicarem e interagirem. Como cita Chet Ramey no Bash FAQ, ver. 4.01:

> "Há uma nova palavra reservada, **coproc,** que especifica um coprocesso: um comando assíncrono que é executado com 2 *pipes* conectados ao *Shell* criador. **coproc** pode receber nome. Os descritores dos arquivos de entrada e saída e o PID do coprocesso estão disponíveis para o *Shell* criador em variáveis com nomes específicos do **coproc.**"

George Dimitriu explica:

> "**coproc** é uma facilidade usada na substituição de processos que agora está publicamente disponível."

A comprovação do que disse Dimitriu não é complicada, quer ver? Veja a substituição de processos a seguir:

```
$ cat <(echo xxx; sleep 3; echo yyy; sleep 3)
```

Viu?! O **cat** não esperou pela conclusão dos comandos entre parênteses, mas foi executado no fim de cada um deles. Isso aconteceu porque estabeleceu-se um *pipe* temporário/dinâmico e os comandos que estavam sendo executados, mandavam para ele as suas saídas, que por sua vez as mandava para a entrada do **cat.**

Isso significa que os comandos desta substituição de processos rodaram paralelos, sincronizando somente nas saídas dos **echo** com a entrada do **cat.**

A sintaxe de um coprocesso é:

```
coproc [nome] cmd redirecionamentos
```

Isso criará um coprocesso chamado **nome.** Se **nome** for informado, **cmd** deverá ser um comando composto. Caso contrário (no caso de **nome** não ser informado), **cmd** poderá ser um comando simples ou composto.

Quando um **coproc** é executado, ele cria um vetor com o mesmo nome **nome** no *Shell* criador. Sua saída padrão é ligada via um *pipe* a um descritor de arquivo associado à variável **${nome[0]}** à entrada padrão do *Shell* pai (lembra que a entrada padrão de um processo é sempre associada por *default* ao descritor zero?). Da mesma forma, a entrada do **coproc** é ligada à saída padrão do *script*, via *pipe,* a um descritor de arquivos chamado **${nome[1]}.**

Assim, simplificando, vemos que o *script* mandará dados para o **coproc** pela variável **${nome[0]},** e receberá sua saída em **${nome[1]}**.

Note que estes *pipes* serão criados antes dos **redirecionamentos** especificados pelo comando, já que eles serão as entradas e saídas do coprocesso.

A partir daqui, vou detalhar rapidamente uns estudos que fiz. Isso contém um pouco de divagações e muita teoria. Se você pular para depois desses **ls's,** não perderá nada, mas se acompanhar, pode ser bom para a compreensão do mecanismo do **coproc.**

Após colocar um **coproc** rodando, se ele está associado a um descritor de arquivo, vamos ver o que tem ativo no diretório correspondente:

```
$ ls -l /dev/fd
lrwxrwxrwx 1 root root 13 2010-01-06 09:31 /dev/fd -> /proc/self/fd
```

Hummm, é um link para o **/proc/self/fd...** O que será que tem lá?

```
$ ls -l /proc/self/fd
total 0
lrwx------ 1 julio julio 64 2010-01-06 16:03 0 -> /dev/pts/0
```

```
lrwx------ 1 julio julio 64 2010-01-06 16:03 1 -> /dev/pts/0
lrwx------ 1 julio julio 64 2010-01-06 16:03 2 -> /dev/pts/0
lr-x------ 1 julio julio 64 2010-01-06 16:03 3 -> /proc/3127/fd
```

Epa, que o **0, 1** e **2** apontavam para **/dev/pts/0** eu já sabia, pois são a entrada padrão, saída padrão e saída de erros padrão apontando para o pseudo terminal corrente, mas quem será esse maldito *device* **3?** Vejamos:

```
$ ls -l /proc/$$/fd                    $$ É o PID do Shell corrente
total 0
lr-x------ 1 julio julio 64 2010-01-06 09:31 0 -> /dev/pts/0
lrwx------ 1 julio julio 64 2010-01-06 09:31 1 -> /dev/pts/0
lrwx------ 1 julio julio 64 2010-01-06 09:31 2 -> /dev/pts/0
lrwx------ 1 julio julio 64 2010-01-06 16:07 255 -> /dev/pts/0
l-wx------ 1 julio julio 64 2010-01-06 16:07 54 -> pipe:[168521]
l-wx------ 1 julio julio 64 2010-01-06 16:07 56 -> pipe:[124461]
l-wx------ 1 julio julio 64 2010-01-06 16:07 58 -> pipe:[122927]
lr-x------ 1 julio julio 64 2010-01-06 16:07 59 -> pipe:[168520]
l-wx------ 1 julio julio 64 2010-01-06 16:07 60 -> pipe:[121302]
lr-x------ 1 julio julio 64 2010-01-06 16:07 61 -> pipe:[124460]
lr-x------ 1 julio julio 64 2010-01-06 16:07 62 -> pipe:[122926]
lr-x------ 1 julio julio 64 2010-01-06 16:07 63 -> pipe:[121301]
```

Epa, aí estão os *links* apontando para os *pipes.* Esse monte de arquivo de *pipe* que foi listado, deve ser porque estava testando exaustivamente essa nova facilidade do Bash.

Para terminar esta teoria chata, falta dizer que o **PID** do *Shell* gerado para interpretar o **coproc** pode ser obtido na variável **$nome_PID** e o comando **wait** pode ser usado para esperar pelo fim do coprocesso. O código de retorno do coprocesso **($?)** é o mesmo de **cmd.**

Exemplos:

Vamos começar com o mais simples: um exemplo sem nome e direto no *prompt*:

```
$ coproc while read Entra              coproc ativo
> do
>     echo -=-=- $Entra -=-=-
> done
[2] 3030
$ echo Olá >&${COPROC[1]}              Manda Olá para a pipe da saída
$ read -u ${COPROC[0]} Sai             Lê do pipe da entrada
$ echo $Sai
-=-=- Olá -=-=-
$ kill $COPROC_PID                     Isso não pode ser esquecido...
```

Como você viu, o vetor **COPROC,** está associado a dois *pipes*; o **${COPROC[1]}** que contém o endereço do pipe de saída, e por isso a saída do **echo** esta redirecionada para ele e o **${COPROC[0]}** que contém o endereço do pipe de entrada, e por isso usamos a opção **-u** do **read** que lê dados a partir de um descritor de arquivo definido, ao invés da entrada padrão.

Como o coprocesso utilizava a sintaxe sem **nome,** o padrão do nome do vetor é **COPROC.**

Só mais uma teoriazinha chata:

```
$ echo ${COPROC[@]}                    Lista todos os elementos do vetor
59 54
```

Como você viu **${COPROC[0]}** estava usando o pipe apontado por **/proc/$$/fd /59** e **${COPROC[1]}** usava **/proc/$$/fd /54.**

Agora chega de teoria mesmo! Vamos agora usar **nome** neste mesmo exemplo, para ver

que pouca coisa muda:

```
$ coproc teste {
> while read Entra
> do
>     echo -=-=- $Entra -=-=-
> done
> }
[6] 3192
$ echo Olá >&${teste[1]}
$ read -u ${teste[0]} Sai
$ echo $Sai
-=-=- Olá -=-=-
$ kill $teste_PID
```

Nesse momento, é bom mostrar uma coisa interessante: Quais são os processos em execução?

```
$ ps                                  Somente um Bash em execução
  PID TTY          TIME CMD
 1900 pts/0    00:00:01 bash
 2882 pts/0    00:00:00 ps
```

Vamos executar 2 coprocessos simultâneos:

```
$ coproc nome1 {                      Coprocesso nome1
> while read x
> do
>    echo $x
> done; }
[1] 2883
$ coproc nome2 {                      Coprocesso nome2
> while read y
> do
>    echo $y
> done; }
bash: aviso: execute_coproc: coproc [2883:nome1] still exists
[2] 2884
```

Xiiii! Acho que deu zebra! Mas será que deu mesmo? Repare que além do **PID 2883** de **nome1,** ele também me devolveu o **PID 2884,** que deve ser de **nome2.** Vamos ver o que está acontecendo:

```
$ ps
  PID TTY          TIME CMD
 1900 pts/0    00:00:01 bash       Esse já existia
 2883 pts/0    00:00:00 bash       Esse está executando nome1
 2884 pts/0    00:00:00 bash       Esse está executando nome2
 2885 pts/0    00:00:00 ps
```

Parece que foi só um aviso, pois os dois **PIDs** informados quando iniciamos os dois coprocessos, estão ativos. Então vamos testar esses 2 caras:

```
$ echo xxxxxxxxx >&${nome1[1]}        Mandando cadeia para nome1
$ echo yyyyyyyyy >&${nome2[1]}        Mandando cadeia para nome2
$ read -u ${nome1[0]} Recebe
$ echo $Recebe
xxxxxxxxx
$ read -u ${nome2[0]} Recebe
$ echo $Recebe
yyyyyyyyy
```

```
$ kill $nome1_PID
$ kill $nome2_PID
```

Mergulhando fundo no nautilus

O nautilus permite que você crie seus próprios *scripts* e os incorpore ao seu ambiente de forma a facilitar a sua vida. *Scripts* são tipicamente mais simples em operação que extensões do Nautilus e podem ser escritos em qualquer linguagem de *script* que possa ser executada no seu computador. Para executar um script, escolha Arquivo ▸ Scripts, e então escolha o *script* que você quer executar a partir do submenu.

Você também pode acessar *scripts* a partir do menu de contexto, isto é, clicando no botão da direita.

Se você não tiver nenhum script instalado, o menu de scripts não aparecerá.

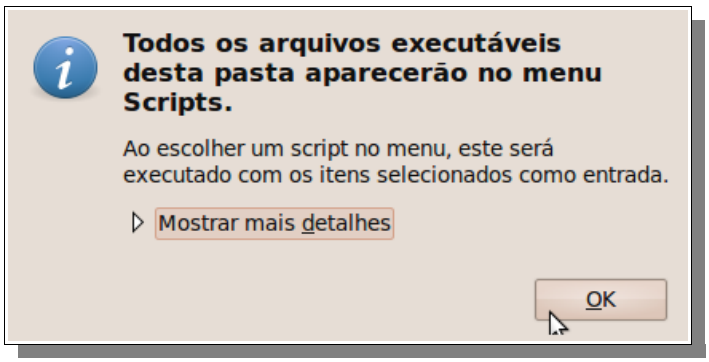
Instalando *scripts* do gerenciador de arquivos

O gerenciador de arquivos inclui uma pasta especial em que você pode armazenar seus *scripts*. Todos os arquivos executáveis nessa pasta aparecerão no menu *scripts* (Arquivos ▸ Scripts). Seu caminho completo é `$HOME/.gnome2/nautilus-scripts.`

Para instalar um *script*, simplesmente copie-o para esta pasta de *scripts* e dê a ele a permissão de execução `(chmod +x <nome do script>).`

Para visualizar o conteúdo da sua pasta de scripts, se você já tiver scripts instalados, escolha Arquivos ▸ Scripts ▸ Abrir Pasta de Scripts.

Aparecerá o seguinte diálogo:



Outra forma de visualizar os *scripts* será navegando até a pasta de scripts com o gerenciador de arquivos, se você ainda não tiver quaisquer scripts. Pode ser necessário visualizar arquivos ocultos, para isto utilize Ver ▸ Exibir Arquivos Ocultos (até porque se você estiver no seu diretório *home* `(~),` sem esta opção ativada, você não verá o diretorio `.gnome2`)

Escrevendo *Scripts* do gerenciador de arquivos

Quando são executados em uma pasta local, os *scripts* receberão como entrada os arquivos selecionados. Quando são executados em uma pasta remota (por exemplo uma pasta mostrando conteúdo web ou de FTP), o *scripts* não receberão parâmetros algum.

A tabela a seguir mostra as variáveis passadas ao *scripts*:

Variável de ambiente	Descrição
NAUTILUS_SCRIPT_SELECTED_FILE_PATHS	Caminhos para os arquivos selecionados, um por linha (apenas para arquivos locais)
NAUTILUS_SCRIPT_SELECTED_URIS	URIs para os arquivos selecionados, um por linha
NAUTILUS_SCRIPT_CURRENT_URI	URI para a localização atual
NAUTILUS_SCRIPT_WINDOW_GEOMETRY	posição e tamanho da janela atual

Quando executados a partir de uma pasta local, podemos passar para um desses *scripts* os conteúdos dessas variáveis. Quando executados a partir dda seleção de arquivo(s) em um computador remoto isso não acontecerá.

Algumas variáveis do nautilus podem e devem ser usadas, elas são:

- NAUTILUS_SCRIPT_SELECTED_FILE_PATHS - são listados os arquivos com caminhos absolutos e com quebra de linha entre eles. Essa é a melhor variável a ser usada, mas ela tem um problema, não funciona em arquivos que estejam na área de trabalho e só funciona em arquivos locais, ou seja, só funciona em rede smb:// se você montar a pasta da rede usando o **`mount`** e o **`smbfs.`**
- NAUTILUS_SCRIPT_SELECTED_URIS - a função dessa variável é idêntica a anterior, com uma diferença, o caminho gerado é sempre no formato file://, smb://, ftp://, http:// etc..., ou seja, ele pode listar qualquer localização no computador, rede ou internet, mas tem um problema crítico, os acentos e espaços são convertidos em códigos, o que impede o seu uso em *scripts*. (nada que não possa ser resolvido por um bom e velho **`sed`** filtrando e convertendo para seu valores originais). Mas porque mencioná-lo? Porque ele é a melhor opção para usar com programas que usem o gnome-vfs, como o gnome-open, Totem, Rhythmbox, etc...
- NAUTILUS_SCRIPT_CURRENT_URI - esta variável contém a pasta atual de execução, equivalente ao comando **`dirname.`** Como a primeira variável, essa aqui não funciona na área de trabalho.
- NAUTILUS_SCRIPT_WINDOW_GEOMETRY - esta variável informa a posição e tamanho da janela do nautilus com da qual foi chamado o *script*. Se você quiser abrir uma janela de diálogo (com o zenity, por exemplo), saberá a posição e o tamanho da janela do nautilus para não superpor. A geometria é informada da seguinte maneira:

  `<largura>x<altura>+<desloc_horizontal>+<desloc_vertical>`

As definições acima foram inspiradas no excelente artigo do Lincoln Lordello em http://www.vivaolinux.com.br/artigo/Nautilus-Scripts/?pagina=1.

Imagine você que ao listar a variável **`$NAUTILUS_SCRIPT_SELECTED_URIS,`** recebi a seguinte cadeia: **`file:///home/julio/%C3%81rea%20de%20Trabalho,`** então tratei de jogá-la em um filtro com **`sed:`**

```
$ sed 's/%C3%81/Á/g;s/%20/ /g' <<< $NAUTILUS_SCRIPT_SELECTED_URIS
file:///home/julio/Área de Trabalho
```

Como um livro não permite que você se afaste muito do assunto, o filtro de **`sed`** que usei como exemplo, só filtrava **`Á`** e espaço em branco, mas se você for usar muito esse tipo de conversão, aconselho a escrever um *script* com todas as conversões possíveis e a cada **`URI`** que você receber, passe a variável como parâmetro para esse *script* (fica melhor ainda se esse *script* for incorporado como uma função externa, sendo executado com o comando **`source`**).

Uma coisa que descobri por acaso, mas não vi nenhuma literatura a respeito, é que se você criar diretórios dentro do **`$HOME/.gnome2/nautilus-scripts`** e dentro desta pasta colocar *scripts* executáveis, esses diretórios também aparecerão no menu (Arquivos ▸ *scripts*), ajudando a organizar e hierarquizar as suas ferramentas.

Exemplos de scripts

Existem 3 tipos de *scripts* que podem ser executados dentro do nautilus:

1. Com interface gráfica – Esta é a melhor forma de interagir com o nautilus, além de ser a mais elegante. A melhor forma de fazê-lo, é usando zenity (você precisa conhecer bem esta interface do *Shell*. É muito fácil, simples, elegante e eficiente);
2. Com interface a caractere sem receber dados externos – Também é simples e o *Shell* sozinho resolve essa parada;
3. Com interface a caractere recebendo dados externos – Essa é de longe a mais chata porque não é óbvia. O problema que surge é em abrir um xterm. Demorei muito e fiz muitas tentativas até conseguir fazê-lo. Mas depois de descoberto o segredo do sucesso, é só seguir a receita do bolo e tudo fica fácil.

Vamos ver alguns exemplos que uso no meu dia-a-dia:

Com interface gráfica:

`$ cat redimen_zen.sh`

```
#!/bin/bash
# Redimensiona fotos direto no Nautilus

IFS="
"           # IFS passa a ser somente o new line
Tipo=$(zenity --list \
    --title "Redimensiona imagens" \
    --text "Informe se redimensionamento\né percentual ou absoluto" \
    --radiolist --column Marque --column "Tipo" \
    true Percentual false Absoluto) || exit 1

if  [ $Tipo = Percentual ]
then
    Val=$(zenity --entry \
        --title "Redimensiona imagens" \
        --text "Informe o percentual de redução" \
        --entry-text 50)% || exit 1 # Concatenando % em $Val
else
    Val=$(zenity --entry \
        --title "Redimensiona imagens" \
        --text "Informe a largura final da imagem" \
        --entry-text 200)x || exit 1
fi

Var=$(zenity --list --title "Redimensiona imagens" \
    --text "Escolha uma das opções abaixo" \
    --radiolist --height 215 --width 390 --hide-column 2 \
    --column Marque --column "" --column Opções \
       false 0 "Saída da imagem em outro diretório" \
       false 1 "Saída da imagem com sufixo" \
       true  2 "Saída da imagem sobregravando a inicial") || exit 1
case $Var in
    0)  Dir=$(zenity --file-selection \
            --title "Escolha diretório" \
            --directory) || exit 1 ;;
```

```
    1)  Suf=$(zenity --entry \
            --title "Redimensiona imagens" \
            --text "Informe o sufixo dos arquivos" \
            --entry-text _redim) || exit 1 ;;
    2) mogrify --resize $Val
            "$NAUTILUS_SCRIPT_SELECTED_FILE_PATHS"
       exit ;;
esac
Arqs=$(echo "$NAUTILUS_SCRIPT_SELECTED_FILE_PATHS" | wc -l)
#  No for a seguir um echo numérico atualiza
#+ a barra de progresso e um echo seguido de um
#+ jogo-da-velha (#) atualiza o texto do cabeçalho
for Arq in $NAUTILUS_SCRIPT_SELECTED_FILE_PATHS
do
    echo $((++i * 100 / $Arqs))
    echo "# Redimensionando $(basename $Arq)"
    sleep 3
    if  [ $Var -eq 0 ]
    then
        convert "$Arq" -resize $Val "$Dir/${Arq##*/}"
    else
        convert "$Arq" -resize $Val "${Arq%%.*}$Suf.${Arq#*.}"
    fi
done | zenity --progress \
        --title "Aguarde. Em redimensionamento" \
        --auto-close --auto-kill
```

Com interface a caractere sem receber dados externos:

$ cat naut_root.sh

```
#!/bin/bash
#  Executa uma sessão de Nautilus como root
#+ O gksudo para pegar rapidamente a senha de root
#+ caso o tempo de sudo tenha expirado.
#+ O comando executado pelo gksudo não produz nada .
#+ isso foi feito para o sudo da lina seguinte ganhar
#+ o privilégio de root, sem pedir senha pela
#+ linha de comando, o que complicaria.

gksudo -u root -k -m "Informe sua senha" true
sudo nautilus --no-desktop $NAUTILUS_SCRIPT_CURRENT_URI
```

Fui informado que existe um *bug* reportado sobre uma mensagem de *warning* que o nautilus, somente sob o Ubuntu, mas que aparentemente não influi na execução da tarefa. Provavelmente quando você ler isso, o *bug* já foi consertado, mas de qualquer forma, vou colocar um outro *script* muito semelhante a este, que é muito útil e está rodando redondinho.

$ cat gedit_root.sh

```
#!/bin/bash
#  Executa uma sessão de gedit como root.
#+ O gksudo para pegar rapidamente a senha de root
#+ caso o tempo de sudo tenha expirado.
#+ O comando executado pelo gksudo não produz nada .
#+ isso foi feito para o sudo da linha seguinte ganhar
#+ o privilégio de root, sem pedir senha pela
#+ linha de comando, o que complicaria.

gksudo -u root -k -m "Informe sua senha" true
sudo gedit $NAUTILUS_SCRIPT_SELECTED_FILE_PATHS
```

Com interface a caractere recebendo dados externos

```
#!/bin/bash
#  Coleta informações para fazer redimensionamento
#+ de imagens diretamente do nautilus
#+ Autor:       Julio Neves
#+ Colaboração: Luiz Carlos Silveira (aka Dom)

#  Abre um xterm para executar o programa
#+ a sintaxe pode parecer estranha, mas
#+ acho que esta é a melhor forma
xterm -T "Redimensiona Imagens" -geometry 500x500 -bg darkred -fg
lightgray -fn '-dejavu-dejavu sans mono-medium-r-*-*-*-*-*-*-*-*-*-*-*' -e
bash -c "source <(tail -n +15 $0)"
exit 0

################Programa propriamente dito.

Verde=$(tput setaf 2; tput bold)  # Valores default em verde
Norm=$(tput sgr0)                 # Restaura cor
clear
# Preparando o basename dos arquivos para listá-los
for Arq in "$NAUTILUS_SCRIPT_SELECTED_FILE_PATHS"
do
    Arqs=$(echo -e "$Arqs${Arq%%*/}\n")
done
echo Os arquivos a redimensionar são:
echo == ======== = ============= ====
column -c$(tput cols) <(echo "$Arqs") # Listando arqs em colunas
read -n1 -p "
Certo? (${Verde}S${Norm}/n): "
[[ $REPLY == [Nn] ]] && exit 1

echo
read -n1 -p "Informe se redimensionamento é:
    ${Verde}P${Norm} - ${Verde}P${Norm}ercentual
    ${Verde}A${Norm} - ${Verde}A${Norm}bsoluto
==> " Tipo

case ${Tipo^} in   #  Conteudo passa para maiuscula (bash 4.0)
    P)  echo ercentual
        read -p "Informe o percentual de redução: " Val
     grep -Eq '^[0-9]+$' <<< $Val || {    # $Val não numérico
         tput flash
         read -n1 -p"Percentual inválido"
         exit 1
         }
     Val=$Val%
     ;;
    A)  echo bsoluto
        read -p "Informe a largura final da imagem: " Val
     grep -Eq '^[0-9]+$' <<< $Val || {
         tput flash
         read -n1 -p"Largura inválida"
         exit 1
         }
     ;;
    *)  tput flash
        read -n1 -p"Informação inválida"
        exit 1
```

```
esac

read -n1 -p "
Informe a saída da imagem que vc deseja:
    ${Verde}D${Norm} - saída da imagem em outro ${Verde}D${Norm}iretório
    ${Verde}S${Norm} - saída da imagem com ${Verde}S${Norm}ufixo
    ${Verde}G${Norm} - saída da imagem sobre${Verde}G${Norm}ravando a
inicial
==> " Saida

case ${Saida^^} in
    D)  echo -e '\010Outro diretório'
        read -p 'Informe o diretório: ' Dir
     [ -d "$Dir" ] || {
         tput flash
         read -n1 -p "Diretório inexistente"
         exit 1
         }
        ;;
    S)  echo -e '\010Sufixo'
            read -p "Informe o sufixo dos arquivos (${Verde}_redim$
{Norm}): " Suf
         Suf=${Suf:-_redim}
        ;;
    G)  echo -e '\010Sobregravando'
            mogrify --resize $Val $NAUTILUS_SCRIPT_SELECTED_FILE_PATHS
            exit
        ;;
    *)  read -n1 -p "Você devia ter escolhido ${Verde}D${Norm}, $
{Verde}S${Norm} ou ${Verde}G${Norm}"
        exit 1
esac
    IFS='
'   # A variável $IFS ficou só com um \n (<ENTER>)
# Agora vamos redimensionar
for Arq in $NAUTILUS_SCRIPT_SELECTED_FILE_PATHS
do
    if  [ ${Saida^^} = D ]
    then
        convert "$Arq" -resize $Val "$Dir/${Arq##*/}"
        echo "$Dir/${Arq##*/}" redimensionado
    else
        convert "$Arq" -resize $Val "${Arq%%.*}$Suf.${Arq#*.}"
        echo "${Arq%%.*}$Suf.${Arq#*.}" redimensionado
    fi
done
```

A primeira linha deste código, a que inicializa um xterm, merece uma atenção maior porque foi nela que perdi muito tempo para associar o programa ao terminal.

Primeiramente vamos entender as opões do xterm usadas:

-T Especifica o título da janela do terminal;

-geometry Especifica <largura>x<altura> do terminal;

-bg Especifica a cor de fundo;

-fg Especifica a cor dos caracteres;

-fn Especifica a fonte adotada (use o programa xfontsel para escolhê-la);

-e Especifica um programa para ser executado na janela.

A partir desse ponto é que demorei a descobrir a melhor forma de associar um programa ao terminal sem ter de criar um segundo script. Achei muito pobre a solução de ter um programa que abria o xterm e chamava um outro para executar a atividade fim, isto é, redimensionar as imagens. Então saiu o finalzinho desta linha, ou seja:

```
-e bash -c "source <(tail -n +15 $0)"
```

Nesse trecho, como já vimos a opção **`-e`** comanda a execução de um **`bash -c`** que, por sua vez, manda executar o comando **`source`** (ou ponto **`(.)`**). Como já vimos no capítulo 7, este último comando chama um *script* externo para ser executado no mesmo ambiente do *script* chamador, isto é, não cria um sub-shell. Como este comando precisa carregar o código de um arquivo, optamos por usar o mesmo Shell chamador, a partir da 15ª linha **`(tail +15)`** do programa corrente **`($0)`**. A construção **`<(...)`** foi usada, porque sabemos que isso faz uma substituição de processos (veja mais no capítulo 8), isto é, passa para a execução do **`source`** a saída do **`tail`** vinda de um arquivo temporário **`(/dev/fd/63)`**.

Inserir na página 320, após o 2o. parágrafo (Uma *Expressão Regular* (ER) ...):

Este texto foi extraído de http://pt.wikipedia.org/wiki/Expressão_regular e não é para assustar, é somente para os que gostam de conhecer a teoria fundamental e o histórico das coisas. Se esse não é o seu caso, isto é, se seu negócio é ir logo metendo a mão na massa, salte direto para a seção “Então vamos meter as mãos na massa” que fica um pouco à frente.

Um pouco de teoria

Em ciência da computação, uma *Expressão Regular* (ou o estrangeirismo *regex*, abreviação do inglês *regular expression*) provê uma forma concisa e flexível de identificar cadeias de caracteres de interesse, como caracteres particulares, palavras ou padrões de caracteres. *Expressões Regulares* são escritas numa linguagem formal que pode ser interpretada por um processador de *Expressão Regular*, um programa que ou serve um gerador de analisador sintático, ou examina o texto e identifica partes que casam com a especificação dada.

O termo deriva do trabalho do matemático norte-americano Stephen Cole Kleene, que desenvolveu as *Expressões Regulares* como uma notação ao que ele chamava de álgebra de conjuntos regulares. Seu trabalho serviu de base para os primeiros algoritmos computacionais de busca, e depois para algumas das mais antigas ferramentas de tratamento de texto da plataforma Unix.

O uso atual de *Expressões Regulares* inclui procura e substituição de texto em editores de texto e linguagens de programação, validação de formatos de texto (validação de protocolos ou formatos digitais), realce de sintaxe e filtragem de informação.

Conceitos básicos

Uma *Expressão Regular* (ou, um padrão) descreve um conjunto de cadeias de caracteres, de forma concisa, sem precisar listar todos os elementos do conjunto. Por exemplo, um conjunto contento as cadeias "Handel", "Händel" e "Haendel" pode ser descrito pelo padrão **`H(ä|ae?)ndel`**. A maioria dos formalismos provêm pelo menos três operações para construir *Expressões Regulares*.

A primeira delas é a alternância, em que uma barra vertical **`(|)`** separa alternativas. Por exemplo, **`psicadélico|psicodélico`** pode casar "*psicadélico*" ou "*psicodélico*". A segunda operação é o agrupamento, em que parênteses (**`(, )`**) são usados para definir o escopo e a precedência de operadores, entre outros usos. Por exemplo, **`psicadélico|psicodélico`** e **`psic(a|o)délico`** são equivalentes e ambas descrevem "*psicadélico*" e "*psicodélico*". Por fim, a terceira operação é a quantificação (ou repetição). Um quantificador após um *token* (como um caractere) ou um agrupamento especifica a

quantidade de vezes que o elemento precedente pode ocorrer. Os quantificadores mais comuns são `?`, `*` e `+`. O ponto de interrogação `(?)` indica que há zero ou uma ocorrência do elemento precedente. Por exemplo, `ac?ção` casa tanto "*acção*" quanto "*ação*". Já o asterisco `(*)` indica que há zero ou mais ocorrências do elemento precedente. Por exemplo, `ab*c` casa "*ac*", "*abc*", "*abbc*", "*abbbc*", e assim por diante. Por fim, o sinal de adição `(+)` indica que há uma ou mais ocorrências do elemento precedente. Por exemplo, `ab+c` casa "*abc*", "*abbc*", "*abbbc*", e assim por diante, mas não "*ac*".

Essas construções podem ser combinadas arbitrariamente para formar expressões complexas, assim como expressões aritméticas com números e operações de adição, subtração, multiplicação e divisão. De forma geral, há diversas *Expressões Regulares* para descrever um mesmo conjunto de cadeias de caracteres. A sintaxe exata da *Expressão Regular* e os operadores disponíveis variam entre as implementações.

História

A origem das *Expressões Regulares* estão na teoria dos autômatos e na teoria das linguagens formais, e ambas fazem parte da teoria da computação. Esses campos estudam modelos de computação (autômatas) e formas de descrição e classificação de linguagens formais. Na década de 1950, o matemático Stephen Cole Kleene descreveu tais modelos usando sua notação matemática chamada de "conjuntos regulares", formando a álgebra de Kleene. A linguagem SNOBOL foi uma implementação pioneira de casamento de padrões, mas não era idêntica às *Expressões Regulares*. Ken Thompson construiu a notação de Kleene no editor de texto QED como uma forma de casamento de padrões em arquivos de texto. Posteriormente, ele adicionou essa funcionalidade no editor de texto Unix ed, que resultou no uso de *Expressões Regulares* na popular ferramenta de busca grep. Desde então, diversas variações da adaptação original de Thompson foram usadas em Unix e derivados, incluindo expr, AWK, Emacs, vi e lex.

O uso de *Expressões Regulares* em normas de informação estruturada para a modelagem de documentos e bancos de dados começou na década de 1960, e expandiu na década de 1980 quando normas como a ISO SGML foram consolidadas.

Então vamos meter as mãos na massa

Página 324 antes do 3o. Parágrafo, Suponhamos que você esteja editando um texto..., incluir:

Para facilitar o aprendizado, dividimos estes metacaracteres em 4 grupos:

- Âncoras;
- Representantes;
- Quantificadores;
- Outros.

Vejamos então cada um deles, procurando dar exemplos bem significativos.

Âncoras

O primeiro grupo de metacaracteres que veremos é formado pelas âncoras. Elas têm esse nome, porque sua finalidade não é combinar (casar) com um texto, mas sim indicar a posição na qual o texto será pesquisado. Veja o quadro resumo a seguir:

ER	Função
`^`	Pesquisar texto no início das linhas
`$`	Pesquisar texto no fim das linhas
`\b`	Pesquisar texto no início e/ou no fim das palavras
`\B`	Negação de `\b`

Exemplos:

No */etc/passwd* do Fedora se você fizer

```
$ grep root /etc/passwd
```

Achará 2 linhas: a primeira é referente ao `root` propriamente dito e a segunda é referente ao usuário `operator` que, por ter seu diretório *home* em `/root`, também é localizado pelo comando `grep`. Para evitar este tipo de coisas, procuramos o `root` somente no início das linhas de `/etc/passwd`. E para isso fazemos:

```
$ grep '^root' /etc/passwd
root:x:0:0:root:/root:/bin/bash
```

Agora veja só esse arquivo:

```
$ cat -vet nums
1$
$
2$
$
$
3$
```

Como sabemos que o `cat` com as opções `-vet` marca o fim de linha com um cifrão `($)`, deduz-se que algumas linhas de `nums` estão vazias. Vamos então brincar com ele um pouquinho:

```
$ sed 's/^/:/' nums
:1
:
:2
:
:
:3
$ sed 's/$/:/' nums
1:
:
2:
:
:
3:
$ sed '/^$/d' nums
1
2
3
```

Primeiramente trocamos o início `(^)` de cada linha por um dois-pontos `(:)`, no segundo

fizemos o mesmo ao final de cada linha `($)` e finalmente deletamos as linhas que tinham o início `(^)` colado no final `($).`

Veja só esse arquivo:

```
$ cat ave
avestruz
ave-do-paraíso
trave
cavei
traveco
```

Como você pode notar, todos os seus registros contém a cadeia `ave.` Primeiramente vejamos os exemplos mais óbvios do uso de bordas:

```
$ grep -E '\bave' ave
avestruz
ave-do-paraíso
$ grep -E 'ave\b' ave
ave-do-paraíso
trave
$ grep -E '\bave\b' ave
ave-do-paraíso
```

Como você pôde ver, o hífen também é considerado borda. Para efeito do `\b` sob *Shell*, só não são bordas as letras, os números e o sublinhado, isso é: `[A-Za-z0-9_]`

Vamos ver outros exemplos, um pouco mais complexos:

```
$ grep -E '\Bave\b' ave
trave
$ grep -E '\bave\B' ave
avestruz
$ grep -E '\Bave\B' ave
cavei
traveco
```

Como vimos, o `\B` casa com tudo que não for borda.

Representantes

Veremos agora os metacaracteres representantes, que são assim chamados porque representam determinados caracteres em um texto. São eles:

ER	Nome	Significado
`.`	Ponto	Qualquer caractere uma vez
`[ ]`	Lista	Qualquer dos caracteres dentro dos colchetes uma vez
`[^ ]`	Lista negada	Nenhum dos caracteres da lista

Página 322 antes do último parágrafo Se você quisesse procurar..., incluir:

Uma dica importante: dentro de uma lista, não existem *metacaracteres*, quero dizer: dentro da lista os *metacaracteres* perdem seus superpoderes e são tratados como simples e reles caracteres mortais.

Exemplos:

`12[:. ]34h`, casará com: `12:34h`, `12.34h` e `12 34h`. Desta forma o ponto `(.)` dentro da lista não representa qualquer caractere, mas somente o literal ponto `(.)`.

Quanto à lista negada, veremos logo à frente, para não perder a didática desta sequência de exemplos,

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.

E tenha sempre na sua memória: todos os quantificadores são gulosos.

Eles são os seguintes:

ER	Nome	Significado
?	Opcional	Torna a entidade anterior opcional
*	Asterisco	Zero ou mais ocorrências da entidade anterior
+	Mais	Uma ou mais ocorrências da entidade anterior
{ }	Chaves	Especifica exatamente a quantidade de repetições da entidade anterior

Vou repetir: todos os Quantificadores são gulosos e, por isso, casarão com o máximo que conseguirem. Mais tarde voltaremos a abordar isso, mas já fique atento e de orelha em pé.

Página 324, quase no final do último parágrafo, trocar: curinga anterior, "qualquer caractere" por: curinga que vimos há pouco, que substitui "qualquer caractere". Aquele é um caractere usado para a expansão de nomes de arquivos, providenciada pelo *Shell*. Esse aqui é um *metacaractere* de *Expressão Regular*. Por favor, não confundam expansão de nomes de arquivos com *Expressões Regulares*. São duas coisas totalmente distintas, apesar de seus *metacaracteres* por vezes serem semelhantes.

Na página 325 excluir o parágrafo: Então até aqui vimos 4 ferramentas: e a tabela imediatamente após, porém em seu lugar, inserir:

Como acabamos de ver, o ponto `(.)` representa qualquer caractere e o asterisco `(*)` representa zero ou mais caracteres da entidade anterior. Desta forma o `.*` é o tudo e o nada, isto é, casa com qualquer coisa: o tudo. Quando o asterisco `(*)` representar zero caracteres da entidade anterior, será o nada.

Este tipo de construção deve, quando possível, ser evitado por ser demasiadamente gulosa. No final deste apêndice, haverão algumas dicas para evitar isso, procurando ser o mais específico possível.

No início da página 326 após o parágrafo: Vamos às respostas:, incluir:

Para exemplificar, vamos usar nos exemplos o arquivo `besteira.txt`, que tem o seguinte conteúdo:

```
$ cat besteira.txt
Eu vi um velho com um fole velho nas costas . tanto fede o fole do
```

```
velho,quanto o velho do fole fede.

Um desafio :diga isso bem rápido !
```

Logo a seguir, na mesma página, antes de 2. Logo após o sinal de pontuação, incluir:

Exemplos:

```
$ sed 's/ [?!.:;,]/X/g' besteira.txt
Eu vi um velho com um fole velho nas costasX tanto fede o fole do
velho,quanto o velho do fole fede.

Um desafioXdiga isso bem rápidoX
```

Como ainda não sabemos guardar o texto casado por *Expressões Regulares*, substituí o que estava incorreto (um sinal de pontuação precedido por um espaço em branco) por um X somente para mostrar que os erros foram localizados corretamente. Por enquanto continuaremos usando este artifício, porém, mais tarde, quando estudarmos **grupos**, voltaremos a estes exemplos, fazendo os acertos definitivos.

Logo a seguir, na mesma página, após: ou sinais repetidos como "??", incluir:

Exemplos:

```
$ sed 's/[?!.:;,][A-Za-z]/X/g' besteira.txt
Eu vi um velho com um fole velho nas costas . tanto fede o fole do
velhoXuanto o velho do fole fede.

Um desafio Xiga isso bem rápido !
```

No fim da mesma página, antes de: 3. Logo após um sinal de pontuação, incluir:

Exemplos:

```
$ sed 's/[?!.:;,][^ ]/X/g' besteira.txt
Eu vi um velho com um fole velho nas costas . tanto fede o fole do
velhoXuanto o velho do fole fede.

Um desafio Xiga isso bem rápido !
```

Como prometi, esse exemplo serviu para explicar o conceito de lista negada.

Na página 327, antes de TÁTICA 2: inserir:

Exemplos:

```
$ sed 's/[?!.:;,][^ ]/X/g' besteira.txt
Eu vi um velho com um fole velho nas costas . tanto fede o fole do
velhoXuanto o velho do fole fede.

Um desafio Xiga isso bem rápido !
```

No 1o. parágrafo da página 328 trocar de Como já foi visto, ... em diante, por:

Como já foi visto, o asterisco `(*)` e o sinal de adição `(+)`, são quantificadores, pois indicam repetição da entidade anterior. Os colchetes `([])` são chamados de classe de caracteres, e o ponto `(.)` é ponto mesmo.

Na página 328, após o texto recuado, inserir:

Fingindo ser lista

Existem 2 classes de caracteres que parecem listas, exercem papel semelhante a elas, mas que não podem ser classificadas como tal. São elas:

- Classes POSIX;

- Sequências de escape.

Classes POSIX

As Casses POSIX parecem listas, representam diversos caracteres, têm sintaxe semelhante à das listas, mas não são listas. Elas foram desenvolvidas para compensar o locale. Mais especificamente isto significa que elas representam todos os caracteres de cada idioma. Ou seja, quando falamos de pt_BR, essas classes envolvem todas as letras acentuadas, além do cedilha.

Classe	Significado	Conteúdo
`[:alnum:]`	Caracteres alfanuméricos	`[A-Za-z0-9]`
`[:alpha:]`	Caracteres alfabéticos	`[A-Za-z]`
`[:blank:]`	Espaço e tabulação	`[ \t]`
`[:cntrl:]`	Caracteres de controle	`[\x00-\x1F\x7F]`(em hexadecimal)
`[:digit:]`	Dígitos	`[0-9]`
`[:graph:]`	Caracteres visíveis	`[\x21-\x7E]` (em hexadecimal)
`[:lower:]`	Caracteres em caixa baixa	`[a-z]`
`[:print:]`	Caracteres visíveis e espaços	`[\x20-\x7E]`(em hexadecimal)
`[:punct:]`	Caracteres de pontuação	`[-!"#$%&'()*+./:;<=>?@[\\\]_`{|}~]`
`[:space:]`	Caracteres de espaços em branco	`[ \t\r\n\v\f]`
`[:upper:]`	Caracteres em caixa alta	`[A-Z]`
`[:xdigit:]`	Dígitos hexadecimais	`[A-Fa-f0-9]`

Exemplos:

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

É uma lista (e os colchetes mais externos é que definem isso), formada pelas classes (que não são listas) de letras maiúsculas, letras minúsculas e números. Mas repare que **`[:upper:] + [:lower:] = [:alpha:].`** Então poderíamos reescrever esta lista da seguinte forma:

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

Agora temos uma lista formada por duas classes: a de letras e a de números. Mas noete ainda que **`[:alpha:] + [:digit:] = [:alnum:].`** Reescrevendo novamente vem:

`[[:alnum:]]`

Repare agora que temos uma lista formada por somente uma classe, e para ser lista, a classe precisa estar envolvida pelos colchetes que definem a sintaxe destes Representantes.

Vamos supor que a variável **`$EndHw`** tenha o endereço de *hardware* (*mac addres*) de um computador. Usando as classes POSIX, fica muito fácil montar uma *Expressão Regular* para verificar se o valor da variável é válido ou não. Veja este fragmento de código:

```
if [[ $EndHw =~ [[:xdigit:]][[:xdigit:]]:[[:xdigit:]][[:xdigit:]]:
[[:xdigit:]][[:xdigit:]]:[[:xdigit:]][[:xdigit:]]:[[:xdigit:]]
[[:xdigit:]]:[[:xdigit:]][[:xdigit:]] ]]
then
    echo Endereço de hardware aprovado
else
    echo Endereço de hardware com formato irregular
fi
```

Pode parecer complicado, mas não é mesmo! Vamos analisá-lo:

Como nós vimos na seção "E tome de test" o comando **`[[...]]`** é um intrínseco (*builtin*) do *Shell* que equivale ao comando **`test`**, e que com o operando **`=~`**, compara *Expressões Regulares*.

Dentro do comando **`test`**, notamos que a dupla **`[[:xdigit:]][[:xdigit:]]:`** se repete 6 vezes. Isto significa dois hexadecimais (algarismos que variam de **`0`** a **`f`**) seguido de dois-pontos **`(:)`**, exceto na última dupla, que não tem os dois-pontos **`(:)`** ao fim**`.`** Ou seja, este é o *mac* propriamente dito.

Este último exemplo foi feito de forma muito rudimentar devido a falta de bagagem de *Expressões Regulares*. Quando estudarmos Grupos, veremos este código reduzir-se a menos de um terço deste.

Sequências de escape

Estas nos já vimos ao longo do livro e estão descritas na seção relativa ao comando **`tr`**, porém estas já vistas, cada uma delas representa um único caractere e, por isso, não se enquadram aqui, porém elas têm extensões que cada uma representa uma gama de caracteres e são essas que nos importam no escopo das *Expressões Regulares*. Vejamos:

Escape	Significado
`\s`	Casa espaços em branco, `\r` ou `\t`
`\S`	Negação de `\s:` casa o que não for espaço em branco, `\r` ou `\t`
`\w`	Casa letras, dígitos, ou `'_'`
`\W`	Negação de `\w`

Exemplos:

`$ cat -vet DOS.txt`

```
Este arquivo foi^M$
gerado por um ftp^M$
mal feito do DOS^M$
ou rWin para o Linux.^M$
```

Repare que ao final de cada linha existe um **^M** antes do cifrão **($).** Isso ocorre porque no DOS/rWin, o fim de linha precisa de um CR (*carriage return*, cujo ascii é **13,** octal **\015** e é representado pela sequência de escape **\r**) e de um LF (*line feed*, cujo ascii é **10,** octal **\012** e é representado pela sequência de escape **\n**), ao passo que os UNIX/LINUX usam somente um CR. Esses dois caracteres de controle são mostrados pelas opções **-vet** do comando **cat** como **^M** e cifrão **($),** respectivamente (veja o comando **cat** no capítulo 3 da primeira parte deste livro.

Veja o que acontece agora:

```
$ sed 's/\s/x/g' DOS.txt
Estexarquivoxfoix
geradoxporxumxftpx
malxfeitoxdoxDOSx
ouxrWinxparaxoxLinux.x
```

Esta linha de comandos trocou os finais de linha do arquivo, além dos caracteres em branco por um **x.** Como o arquivo é do estilo DOS, termina com um CR e um LF. Vamos usar o comando **tr** para remover os CR **(\r).**

```
$ tr -d '\r' < DOS.txt | tee arq.DOS | cat -vet
Este arquivo foi$
gerado por um ftp$
mal feito do DOS$
ou rWin para o Linux.$
```

O **tr -d** removeu os CR e o comando **tee** jogou a saída do **tr** para o arquivo **arq.DOS** e para o comando **cat** com a opção **-vet,** para que você veja que o CR já era. Vamos executar o mesmo **sed** novamente:

```
$ sed 's/\s/x/g' arq.DOS
Estexarquivoxfoi
geradoxporxumxftp
malxfeitoxdoxDOS
ouxrWinxparaxoxLinux.
```

Como você viu, o **x** no fim da linha foi causado pela sequência de escape **\s,** do **sed,** atuando sobre o CR e não sobre o LF.

Vamos agora ver o uso do \w (de word) usando o mesmo sed:

```
$ sed 's/\w/x/g' <<< 'Batatinha frita 1, 2, 3!'
xxxxxxxxx xxxxx x, x, x!
$ sed 's/\W/x/g' <<< 'Batatinha frita 1, 2, 3!'
Batatinhaxfritax1xx2xx3x
```

Ou seja, a sequência de escape **\w** casou com todas as letras e todos os algarismos. Se houvesse sublinha **(_),** ele também casaria. No exemplo seguinte, o **\W** casou exatamente o oposto.

Na página 328, no parágrafo após o texto recuado, apagar: que relembramos as ferramentas já vistas, ficando o texto: Agora vamos aumentar nosso arsenal...

No fim da página 328, após revistas?, inserir:

Exemplos:

```
$ Manchete="A obra foi revista e publicada nas revistas"
$ grep -Eo 'revistas?' <<< "$Manchete"
```

```
revista
revistas
```

Só para relembrar, a opção `-E` do `grep` serve para usarmos *Expressões Regulares* estendidas (como é o caso do opcional `(?)`) e a opção `-o` é usada, para que o `grep` devolva somente o trecho casado. Desta forma vimos que ele casou tanto `revista,` quanto `revistas`.

Na página 329 trocar:

`z{,5}`

ou

`z{3,}`

que seriam "até 5" e "no mínimo 3" letras `z,` respectivamente.

Por:

`z{5}`

ou:

`z{3,}`

que equivaleria a "exatamente 5" ou "no mínimo 3" letras `z,` respectivamente.

Exemplos:

```
$ sed -r 's/e{3}/www./g' <<< eeejulioneves.com
www.julioneves.com
$ grep -Eo 'a{2,}' <<< "Um espirro faz assim: aaatchim"
aaa
```

Neste último, usamos as *Expressões Regulares* expandidas do `grep` (opção `-E`) e a opção `-o` que mostra somente o que casou. Daí vimos que apesar de haverem diversas letras `a` no texto, a *Expressão Regular* só casou com `aaatchim,` onde, como solicitado, haviam pelo menos duas letras `a` consecutivas.

Podemos agora dar uma melhoradinha na nossa rotina de crítica do *mac address,* que vimos quando falamos das classes POSIX. Veja:

```
if [[ $EndHw =~ [[:xdigit:]]{2}:[[:xdigit:]]{2}:[[:xdigit:]]{2}:
[[:xdigit:]]{2}:[[:xdigit:]]{2}:[[:xdigit:]]{2} ]]
then
    echo Endereço de hardware aprovado
else
    echo Endereço de hardware com formato irregular
fi
```

Melhorou, mas quando aprendermos as ferramentas que virão logo a seguir, ficará muito melhor ainda.

Na página 329, penúltimo parágrafo, apagar a frase: Vamos ver mais uma ferramenta, o agrupamento com parênteses ().e inserir o texto a seguir:

Outros

Lembra que eu disse que os *metacaracteres,* para efeito didático, se dividiam em 4 grupos: Âncoras; Representantes; Quantificadores e Outros?

- Mas porque Outros? Você vai me perguntar.
- Ora, porque não se encaixam em nenhum dos grupos anteriores.

Veja quais são os componentes do grupo Outros:

ER	Nome	Significado
`\`	escape	Tira os poderes do caractere seguinte
`|`	ou	Escolhe entre opções
`( )`	grupo	Reúne caracteres
`\1...\9`	retrovisor	Retorna texto casado pelo grupo

Algumas vezes você não deseja que um *metacaractere* **`. ^ $ * + ? \ [ ( { | )`** seja interpretado como tal, então precisamos tirar seus superpoderes. Existem duas formas de fazer isso:

- Colocando o *metacaractere* dentro de uma lista (como já havíamos visto);
- Precedendo o *metacaractere* com uma contrabarra.

Exemplos:

Vamos montar uma linha de comandos para criticar um número de CEP, que como você sabe, tem o formato NN.NNN-NNN, onde cada N equivale a um algarismo de zero a nove. Como primeira tentativa vamos fazer:

```
$ [[ $cep =~ [0-9]{2}.[0-9]{3}-[0-9]{3} ]] && echo CEP $cep OK
```

Agora vamos executá-la duas vezes. A primeira com a variável **`cep=12.345-678`** e a segunda com **`cep=123456-789:`**

```
$ cep=12.345-678
$ [[ $cep =~ [0-9]{2}.[0-9]{3}-[0-9]{3} ]] && echo CEP $cep OK
CEP 12.345-678 OK
$ cep=123456-789
$ [[ $cep =~ [0-9]{2}.[0-9]{3}-[0-9]{3} ]] && echo CEP $cep OK
CEP 123456-789 OK
```

Hiii, a segunda execução deu um resultado imprevisto pois o número do CEP não era válido. Vamos botar uma contrabarra antes do ponto **`(.)`** e vamos testar novamente:

```
$ [[ $cep =~ [0-9]{2}\.[0-9]{3}-[0-9]{3} ]] && echo CEP $cep OK
$
```

Haaa, agora sim! Agora não deu a mensagem dizendo que o CEP estava certo. Isso ocorreu porque o ponto **`(.)`** é um *metacaractere* que casa com qualquer caractere. Quando o precedemos com uma contrabarra, ele perde seus superpoderes e passa a ser simplesmente o literal ponto **`(.)`**.

Suponha que de um rol de frutas, você queira aceitar somente **`pera`** ou **`uva`** ou **`maçã.`** Para isso constrói-se uma *Expressão Regular* usando o alternador, que é representado por uma barra vertical **`(|).`** Vejamos:

Exemplos:

```
$ cat frutas
abacate
maçã
morango
pera
```

```
tangerina
uva
$ grep -E 'pera|uva|maçã' frutas
maçã
pera
uva
```

Não se esqueça que uma lista é uma espécie de ou para somente uma letra. Assim `[gpr]ato`, é o mesmo que `gato|pato|rato`. Mas quando isso for possível, lembre-se sempre de usar a lista.

Você verá que o uso dos parênteses formando um grupo, restringe a abrangência do ou `(|)`, e pode parecer estranho, mas é essa limitação que lhe dá mais poder.

No início da página 330, abaixo da palavra Exemplo:, inserir:

Com o uso de grupos podemos melhorar aquele fragmento de código que usamos para criticar o endereço de *hardware* (*mac address*). Veja:

```
if [[ $EndHw =~ ([[:xdigit:]]{2}:){5}[[:xdigit:]]{2} ]]
then
    echo Endereço de hardware aprovado
else
    echo Endereço de hardware com formato irregular
fi
```

Note que com o uso do grupo (parênteses) foi possível montar um conjunto formado por 2 dígitos hexadecimais e um dois-pontos `(:)`, o que possibilita fazer este grupamento como um todo ocorrer 5 vezes, ficando de fora somente os 2 últimos dígitos hexadecimais, porque estes não são sucedidos por dois-pontos `(:)`.

No alto da página 332 abaixo de Exemplo: inserir:

Vamos ver um exemplo prático do uso de grupos com retrovisores. Suponha que temos um cadastro de aniversariantes com o seguinte formato:

```
cat aniv
1919-11-08 Hedy Coutinho
1947-07-05 Silvina Duarte
1980-01-17 Juliana Duarte
1984-11-08 Paula Duarte
```

Para procurar os aniversariantes de um determinado mês seria muito trabalhoso, pois ele está ordenado por ano de nascimento. Para listar os aniversariantes de Novembro (11) montamos o seguinte `sed`.

```
$ sed -rn 's/[0-9]{4}-11-[0-9]{2} (.*)/\1/p' aniv
Hedy Coutinho
Paula Duarte
```

Neste `sed`, definimos uma *Expressão Regular* para casar o ano `([0-9]{4})`, `-11-` (que é o mês `11` entre seus dois separadores), o dia de nascimento `([0-9]{2})`, um espaço em branco (que é o separador entre a data e o nome) e o nome `((.*))`.

Como a *Expressão Regular* do nome está entre parênteses, o texto casado foi armazenado para uso futuro. Como o `sed` usava o comando de substituição `(s)`, todo o texto casado foi substituído pelo valor que casou com o nome `(\1)`, pois este é o retrovisor que recuperou o que havia sido previamente armazenado.

A opção `-n` do `sed` diz pra ele só jogar para a saída, o que for ordenado e o comando *print* `(p)`, que está no fim da linha de comando, diz ao `sed` para imprimir as linhas casadas. Veja o que aconteceria se não usássemos a opção `-n` juntamente com o comando *print* `(p)`:

```
$ sed -r 's/[0-9]{4}-11-[0-9]{2} (.*)/\1/' aniv
Hedy Coutinho
1947-07-05 Silvina Duarte
1980-01-17 Juliana Duarte
Paula Duarte
```

Ou seja, jogaria para a saída o nome dos aniversariantes de Novembro, mas mandaria também os registros inteiros das outras pessoas.

Este é um exemplo complexo, porém completo.

Conforme prometido, vamos agora corrigir o bom e velho **besteira.txt.** Ele está assim:

```
$ cat -vet besteira.txt
Eu vi um velho com um fole velho nas costas . Tanto fede o fole do
velho,quanto o velho do fole fede.$
$
Um desafio^I:diga isso bem rM-CM-!pido !$
```

Repare na última linha que temos um **^I** entre desafio e os dois-pontos **(:)** e temos ainda **rM-CM-!pido.** O **^I** é uma **<TAB>** que está no texto e **M-CM-!** É a representação da letra **'a'** acentuada **(á).**

Primeiramente vamos eliminar os espaços em branco e as **<TAB>** entre as palavras e a pontuação:

```
$ sed -r 's/([[:alnum:]])[      ]([[:punct:]])/\1\2/g' besteira.txt | tee
besteira.txt
Eu vi um velho com um fole velho nas costas. Tanto fede o fole do
velho,quanto o velho do fole fede.

Um desafio:diga isso bem rápido!
```

Como não sabia os textos que iriam casar, usei os parênteses para montar 2 grupos:

- O primeiro casando letras e números (poderia ter usado **([[:alnum:]]+)** para casar palavras);
- O segundo com os caracteres de pontuação.

Entre os dois, há uma lista que em seu interior tem um espaço em branco e uma **<TAB>** (poderia ter usado **\s** ou **[:space:]**).

Dei a saída para um **tee,** porque este a divide para o arquivo indicado (no caso **besteira.txt**) e a saída padrão. Se não fosse para mandar para a saída padrão, poderia ter usado somente a opção **-i** do **sed.**

Na saída do **sed,** juntei o texto casado pelo primeiro grupo **(\1)** com o texto casado pelo segundo **(\2)** desta forma jogando fora os espaços em branco e os **<TAB>.**

Vamos agora pegar este arquivo já modificado e procurar pelos caracteres de pontuação que não estão separados por um espaço em branco da palavra seguinte.

```
$ sed -r 's/([[:punct:]])([[:alnum:]]+)/\1 \2/g' besteira.txt | tee
besteira.txt
Eu vi um velho com um fole velho nas costas. Tanto fede o fole do velho,
quanto o velho do fole fede.

Um desafio: diga isso bem rápido!
```

Os grupos formados agora foram para os caracteres de pontuação e para palavras, mandando para a saída o texto casado pelo primeiro grupo (um caractere de pontuação), um espaço em branco e o texto casado pelo segundo grupo (uma palavra). Para o **tee,** vale a mesma observação do exemplo anterior.

Mas poderíamos fazer isso tudo em um único **sed.** Veja

```
$ sed -ri 's/([[:alnum:]])[     ]([[:punct:]])/\1\2/g;s/([[:punct:]])([[:alnum:]]+)/\1 \2/g' besteira.txt
$ cat besteira.txt
Eu vi um velho com um fole velho nas costas. Tanto fede o fole do velho, quanto o velho do fole fede.

Um desafio: diga isso bem rápido!
```

Pronto! Agora só falta tirar a penúltima linha, que está vazia:

```
$ sed -i '/^$/d' besteira.txt
$ cat besteira.txt
Eu vi um velho com um fole velho nas costas. Tanto fede o fole do velho, quanto o velho do fole fede.
Um desafio: diga isso bem rápido!
```

Para remover a linha vazia, pesquisei por **^$,** isto é, o início está junto do fim e usei o comando **d** (*delete*) do **sed.**

Se quisesse remover as linhas vazias ou as que continham somente espaços em branco e/ou **<TAB>,** deveríamos fazer:

```
$ sed -i '/^[   ]*$/d' besteira.txt
```

Isto é, entre o início **(^)** e o fim **($)** da linha montei uma lista formada por espaço em branco ou **<TAB>** e seguida por um asterisco **(*).** Quando o asterisco **(*)** representa zero ocorrências, a linha será vazia, caso contrário, entre o início e o fim da linha poderão haver diversas ocorrências, mas somente destes caracteres.

Mais uma dica sobre correção de texto: é comum repetirmos uma palavra quando estamos escrevendo um texto (eu mesmo, me considero um escritor gago...:). Podemos corrigir isso procedendo assim:

Consideramos que palavras são formadas por letras, números e sublinhados **(_).** Então vamos montar um *Expressão Regular* para isso e repeti-la com o uso dos retrovisores.

1ª Tentativa

```
([A-Za-z0-9_]+) \1
```

Fiz uma lista com maiúsculas, minúsculas, algarismos e sublinhado, ou seja, tudo que pode formar uma palavra. O sinal de adição **(+)** após a lista, diz que ao menos um de seus componentes tem de ocorrer pelo menos uma vez. Isso tudo está entre parênteses de modo a formar um grupo, que salvará o texto casado com esta *Expressão Regular*. Seguindo este grupo vem um espaço em branco que seria o fim da 1ª palavra e o retrovisor **(\1)** que traz o texto salvo pelo grupo que descrevemos. Vamos ver se isso funciona. Parece que sim... Para testar vamos montar um **sed** que elimine uma das repetições.

```
$ sed -r 's/([A-Za-z0-9_]+) \1/\1/' <<< 'Gosto de de delícias'
Gosto de delícias
```

Parece que funcionou, vamos repetir o exemplo, agora com o texto já corrigido:

```
$ sed -r 's/([A-Za-z0-9_]+) \1/\1/' <<< 'Gosto de delícias'
Gosto delícias
```

Iiiihh, a preposição **de** foi pro brejo! Vamos usar o **grep** com a opção **-o** para entender o que aconteceu:

```
$ grep -Eo '([A-Za-z0-9_]+) \1' <<< 'Gosto de delícias'
de de
```

Ahh, então foi isso! Realmente a preposição `'de'` se repete na primeira sílaba de `dedilhar.` Podemos acabar com esta confusão usando as bordas `(\b).` Então vejamos:

```
$ sed -r 's/\b([A-Za-z0-9_]+) \1\b/\1/' <<< 'Gosto de de delícias'
Gosto de delícias
$ sed -r 's/\b([A-Za-z0-9_]+) \1\b/\1/' <<< 'Gosto de delícias'
Gosto de delícias
```

Com as 3 bordas (dois `\b:` um no início e outro no final e o espaço em branco entre as palavras) a expressão funcionou como esperado. Lembre-se sempre de usar Âncoras quando possível. A falta delas provoca erros dificílimos de serem detectados.

Na página 334 tirar o quadro de dicas e inserir todo o texto a seguir:

Expressões Regulares (no BrOffice.org)

Apesar de não ter nada a ver com *Shell*, todos os "shelleiros" usam o BrOffice.org. Como vocês já sabem o básico de *Expressões Regulares*, vou inserir esta seção como uma colher de chá (ou será colher de *Shell*?:).

Obviamente, não mostrarei tudo novamente, mas somente as diferenças que já testei entre as sintaxes das *Expressões Regulares* do *Shell* e do BrOffice.org, dando ênfase para as surras que tomei para descobrir essas diferenças.

As surras foram maiores, porque à época não se achava documentação sobre o uso de *Expressões Regulares* no BrOffice.org. Atualmente, existe um bom manual em http://wiki.services.openoffice.org/wiki/Documentation/How_Tos/Regular_Expressions_in_Writer, o qual algumas vezes consultei para escrever este texto.

Obs: quero destacar que a última manutenção que dei neste texto foi no verão de 2010 e atualmente existe um movimento no BrOffice.org para mudar a máquina (engine) de *Expressões Regulares* para um novo modelo, descrito em http://userguide.icu-project.org/strings/regexp. Veja bem: isso não é certo que vá acontecer, e caso ocorra, as mudanças de sintaxe não incompatibilizarão com a atual.

Dito isso, vamos ao que interessa.

Onde usar *Expressões Regulares* no BrOffice.org

Você pode usar *Expressões Regulares* nas seguintes situações:

No *Writer*:

- Editar → Localizar e Substituir;
- Editar → Alterações → Aceitar ou Rejeitar → Tab Filtro

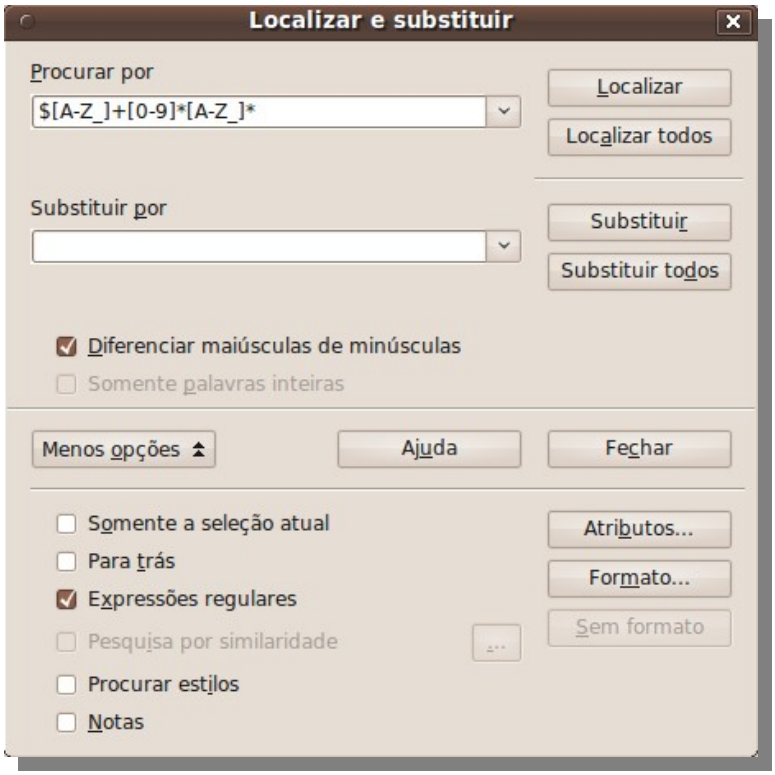
No *Calc*:

- Editar → Localizar e Substituir;
- Dados → Filtro → Filtro Padrão & Filtro Avançado;
- Algumas funções como SOMASE, PROCURAR e outras.

No *Base*:

- No comando Find Record (isso foi um chute que dei, porque não conheço o Base)

As caixas de diálogos que aparecem quando você usa os comandos acima, geralmente têm uma opção para usar *Expressões Regulares* (que normalmente está desligada). Por exemplo:



Diferenças na lógica de uso

Existem diferenças de sintaxe com as *Expressões Regulares* que aprendemos, e existem também diferenças na lógica de aplicação. Essas são mais chatas para os veteranos de *Expressões Regulares* sob o *Shell*, pois é necessário uma mudança na forma de raciocinar. Veja estes casos:

Havia notado um erro no meu livro, mas esqueci de anotar o número da página. Quando fui fazer a alteração, a única coisa que me lembrava era que o erro estava em uma variável do *Shell*. O que fiz? Montei uma *Expressão Regular* começando por cifrão **($)** e casando letras maiúsculas, algarismos e sublinhados **(_),** e estes algarismos não podiam suceder o cifrão **($).** Era assim:

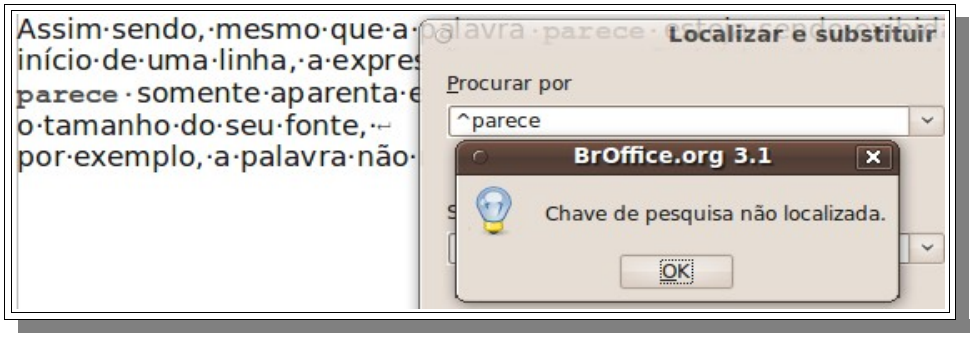
```
$[A-Z_]+[0-9A-Z_]*
```

Mas, apesar de certa, a *Expressão Regular* não funcionava pois casava com as variáveis em minusculas também. Demorei muito para descobrir que o *check button* "Diferenciar maiúsculas de minúsculas" é que decide quanto à caixa das letras (veja a figura anterior).

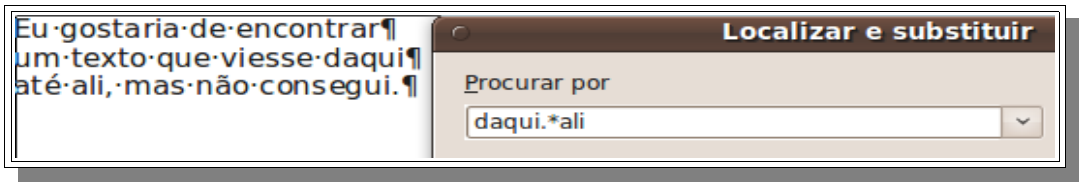
Um outro problema de diferença de lógica, é que no BrOffice.org as *Expressões Regulares* dividem o texto a ser pesquisado em porções e examinam cada porção separadamente

No *Writer,* o texto é dividido em parágrafos e o que define um parágrafo é a porção de texto entre um **enter** ou um *hard enter* **(↵)** (caractere não imprimível obtido com **<SHIFT>+<ENTER>** também chamado de *new line*) e outro desses.

Assim sendo, mesmo que a palavra **parece** esteja sendo exibida no início de uma linha, a expressão **^parece** não irá localizá-la, pois **parece** somente aparenta estar no início da linha, mas se alterarmos o tamanho do seu fonte, por exemplo, a palavra não necessariamente estará mais lá situada.



Por outro lado, a *Expressão Regular* **`daqui.*ali,`** não irá casar o texto formado pelo **`daqui`** em um parágrafo até o **`ali`** em outro. Normalmente os parágrafos são tratados individualmente.



Além disso, o *writer* considera cada célula de tabela e quadro separadamente. Os quadros são examinados após todos os textos e celulas de tabela terem sido examinados

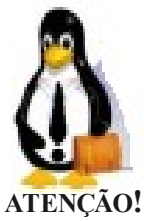
Na caixa de diálogo Localizar e Substituir, as *Expressões Regulares* devem ser usadas somente no *box* "Procurar por". Elas não podem ser usadas no "Substituir por:" porque *Expressões Regulares* casam com textos e portanto o que deve ser substituído é o texto e não a *Expressão Regular*.

Diferenças de sintaxe

A partir de agora, veremos as diferenças de sintaxe entre o que aprendemos nas *Expressões Regulares* do Bash e as que veremos do BrOffice.org.

Âncoras

Como já vimos existe uma diferença na lógica de uso das Âncoras pois no Bash os *metacaracteres* de início **`(^)`** e de fim **`($),`** procuram respectivamente os textos que os sucedem ou precedem no início ou no fim de uma linha. No BrOffice.org, esta procura é feita no início ou no fim de um parágrafo.


ATENÇÃO!

Podemos também citar um caso que já foi reportado ao BrOffice.org para correção. O **`alternativo`** ou **`ou (|)`** pode atrapalhar o início **`(^).`** Assim, pensando no mengão, se procurarmos **`^vermelho|preto,`** o BrOffice.org devolverá as ocorrências de **`vermelho`** iniciando parágrafos ou **`preto`** em qualquer lugar. Contudo se fizermos: **`vermelho|^preto`** serão devolvidos o **`vermelho`** e o **`^preto,`** ambos em qualquer lugar, desta forma não reconhecendo o *metacaractere* **`^`** como a Âncora que marca o início de um parágrafo.

Ué, mas não íamos falar sobre as diferenças de sintaxe? Sim, mas é bom frisar bastante este conceito de parágrafo porque isso algumas vezes me dá uma derrubada.

Dentre as Âncoras, as únicas diferenças de sintaxe que existem no uso de *Expressões Regulares* no Bash e no BrOffice.org estão nos *metacaracteres* que definem as bordas. As

contrabarras `(\)` que são *metacaracteres* (*escape*), assumem, na pesquisa uma função diferente quando seguidas por um menor `(\<)` ou um maior `(\>).` Vejamos:

Para procurar palavras que começam por texto:

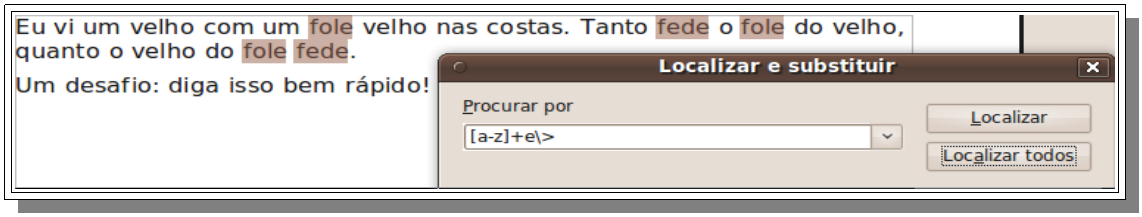
- No Bash fazemos: `\btexto;`
- No BrOffice.org fazemos: `\<texto.`

Para procurar palavras que terminam por texto:

- No Bash fazemos: `texto\b;`
- No BrOffice.org fazemos: `texto\>.`

Exemplos:

A *Expressão Regular* `[a-z]+e\>` neste exemplo, foi usada para procuramos por uma ou mais letras `([a-z]+),` terminando com a letra `(e\>).`



Representantes

Quase todos os *metacaracteres* Representante teem uso idêntico ao seus congêneres sob o Bash, porém nem tudo neste mundo é perfeito, porque mesmo que poucas, ainda existem algumas diferença de uso.

Sob o Bash, todos os caracteres dentro de uma lista `([])` são tratados como literais, exceto o circunflexo que é tratado como um negador da lista (e assim mesmo quando ele é o seu primeiro elemento). Sob o BrOffice.org usamos a contrabarra `(\)` em uma lista para "escapar" alguns alguns *metacaracteres e* também para formar códigos hexadecimais.

Somente os caracteres abre colchetes `([),` hífen `(-)` e contrabarra `(\)` devem ser "escapados" no interior de uma lista, já que nesse ambiente, seus significados são especiais.

Por exemplo a *Expressão Regular* `[[\]a-z]` casa um abre colchetes `([),` ou um fecha colchetas `(]),` ou uma letra minúscula. `[\\]` casa com uma contrabarra literal, `[\t]` casa com a letra `'t'.` Para casarmos um `<TAB>,` devemos fazer: `[\x0009]` que é a representação ascii em hexadecimal do `<TAB>.`

Todos os outros caracteres são tratados pelos seus valores normais, não precisando mais nenhum artifício.

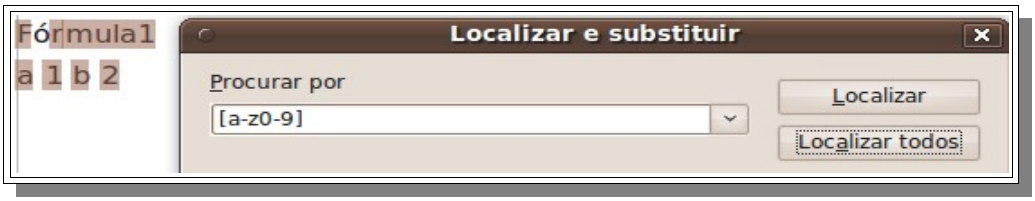
ATENÇÃO!

Usando Classes de Caracteres POSIX, surge um caso que também já foi reportado, isso porque uma classe como `[[:alpha:]],` deveria casar com qualquer algarismo decimal, mas isso não se verifica.

Pelo site do BrOffice.org, este fato reportado funcionaria se fizéssemos `[:alpha:],` o que de cara já te tráz algumas restrições, como exemplo suponha que eu queira procurar em um arquivo todas as letras minúsculas e todos os algarismos decimais. Nesse caso, sob o BrOffice.org, não poderia usar Classes de Caracteres POSIX. No Bash faríamos `[[:lower:][:digit:]]` porém isso no BrOffice.org é impossível, pois em virtude dos

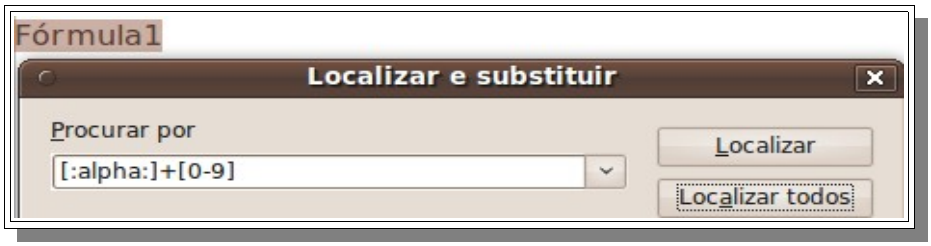
colchetes `([])` mais externos, ele encararia isso como uma lista formada todos os caracteres internos a esses colhetes `([])` e não como duas classes.

Da mesma forma, também não funcionaria `[a-z[:digit:]],` nem `[[:lower:]0-9].` A saída seria usarmos uma lista de caracteres, mas assim mesmo, veja só isso:



Viu! Neste caso as letras acentuadas não casaram com o nosso padrão.

Por outro lado, pela facilidade que tenho no uso de *Expressões Regulares*, venho há muito tempo testando-as no BrOffice.org, e não sei se por erro meu, mas até à versão que estou escrevendo este documento (3.1.1), não consegui casar `[:digit:]` nenhuma vez. Veja:



Duas observações:

1. Usando a classe [:alpha:], conseguimos casar as letras acentuadas;
2. A classe [:digit:] não funfou!

Resumindo: essa facilidade ainda está em fase de consolidação e por isso sempre que posso usar as listas convencionais (como no caso de algarismo), não penso duas vezes, uso-as.

Quantificadores

Aqui não há o que temer nem acrescentar. Os Quantificadores do BrOffice.org se comportam da mesma forma que os do Bash. Poupe seu fôlego para a seção seguinte.

Outros

Aqui que a porca torce o rabo! Nesta classe de *metacaracteres* existem diferenças substancias no uso de *Expressões Regulares*, a começar pelo que já vimos, de uma Âncora sucedendo um `ou (|).` Recapitulando:

Se procurarmos `^vermelho|preto,` o BrOffice.org devolverá as ocorrências de `vermelho` iniciando parágrafos ou `preto` em qualquer lugar. Contudo se fizermos: `vermelho|^preto` serão devolvidos o `vermelho` e o `^preto,` ambos em qualquer lugar, desta forma não reconhecendo o *metacaractere* `^` como a Âncora que marca o início de um parágrafo.

Vamos ver como funcionam os grupos e os retrovisores: Como já sabemos os grupos são criados com o uso de parênteses. Os **textos** casados (veja bem, são os textos e não as *Expressões Regulares*) ficam guardados e podem ser recuperados dentro da mesma *Expressão Regular*. No exemplo a seguir, procuramos por duas palavras iguais, separadas por um hífen (-). Veja:

A *Expressão Regular* era a seguinte:

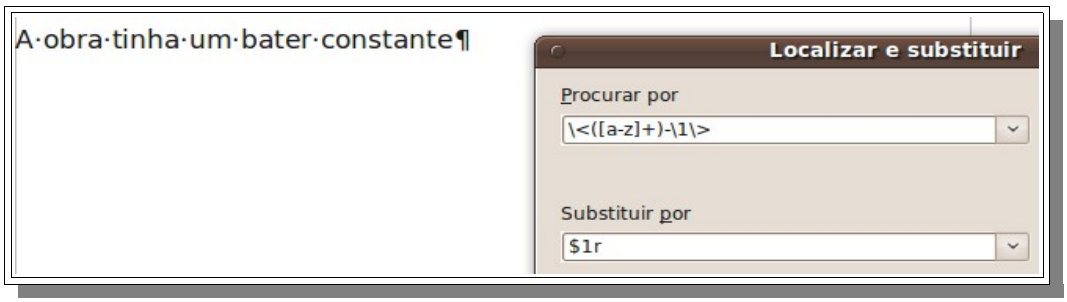
\<([a-z]+)-\1\>

Onde:

- `\<` e `\>` Formam a borda esquerda e direita do texto, respectivamente;
- `([a-z]+)` Define uma ou mais letras seguidas (palavra) dentro de um grupo;
- -\1 O hífen seguido do texto casado pelo grupo.

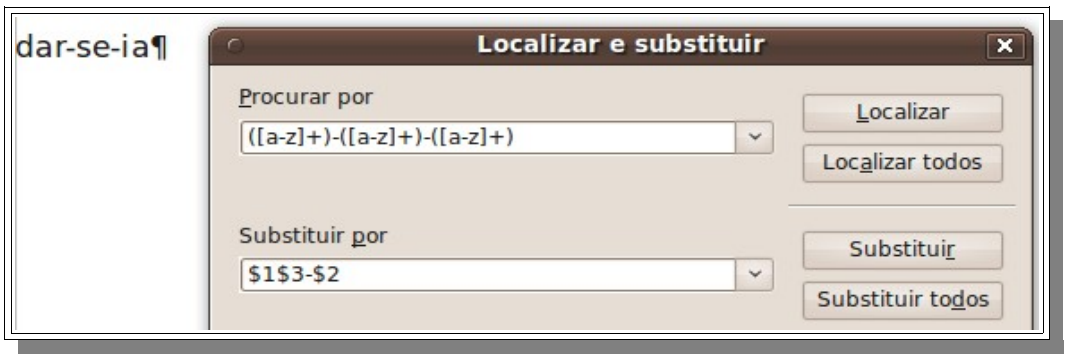
Até agora não mudou nada do que foi dito sobre *Expressões Regulares* no Bash e tudo funcionou às mil maravilhas. Mas se você prestar a atenção, o que fizemos foi clicar em um dos botões de **localizar**. A diferença começa quando desejamos usar o texto casado para **substituir** alguma coisa.

Vamos aproveitar este exemplo para trocar `bate-bate` por `bater:`

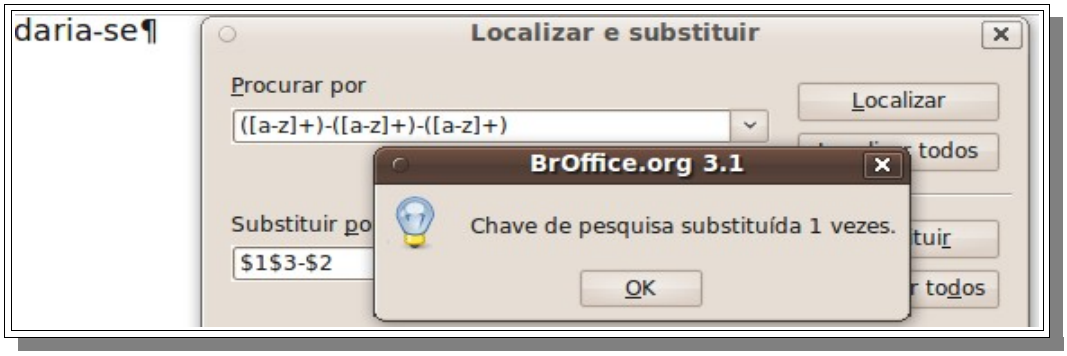


Repare que na caixa "Substituir por" eu usei `$1r`, ou seja, o texto casado não é mais o `\1`, agora é o `$1.`

Eu odeio mesóclises e tinha o seguinte texto:



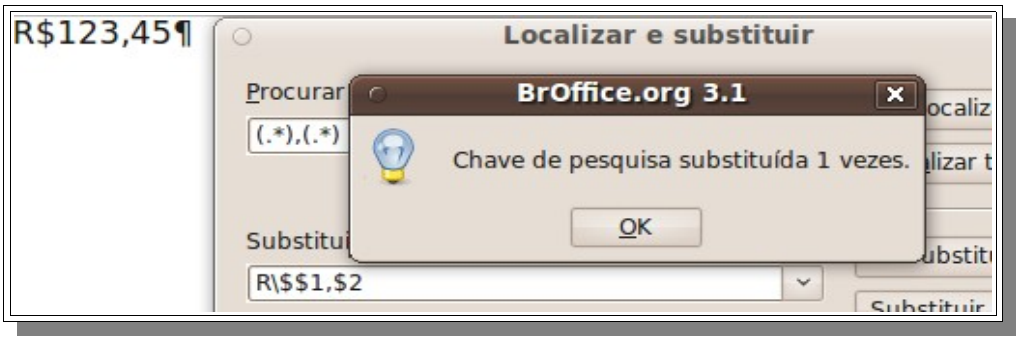
Este `dar-se-ia` estava me torturando, então troquei-o usando uma *Expressão Regular* em que eu montava 3 grupos separados por hífen `(-).` Então, o primeiro `($1)` recebeu `dar,` o segundo `($2)` recebeu `se` e o terceiro `($3)` recebeu `ia.` Colocando-os na ordem que eu queria `($1$3-$2),` veio:



Esta sintaxe pode parecer estranha, mas não é, ela é semelhante à usada na linguagem Perl.

Duas observações sobre o uso de cifrão **($)** para substituir texto:

1. Se no seu texto tem um valor **123,45** e você desejar substitui-lo por **R$123,45** você deverá colocar na caixa "Substituir por" **R\$123,45,** veja:



   O primeiro grupo casou tudo até a vírgula **(,)** e o segundo ficou com o resto do número. Na substituição, o primeiro cifrão foi precedido por uma contrabarra **(\$)** para especificar que era um literal

2. O $0 na caixa "Substituir por" substitui por todo o texto casado.

A contrabarra **(\)** nas *Expressões Regulares* do BrOffice.org também tem suas peculiaridades:

\t Quando fora de uma lista equivale a um **<TAB>.** Como já foi dito, dentro de uma lista o **<TAB>** deve ser usado com seu código hexadecimal **[\x0009].**

Então para trocar todos os <TAB> por espaços em branco, ponha na caixa "Procurar por" um \t e na caixa "Substituir por" um espaço em branco;

\n Casa com um *hard enter* ou *new line* (↵) (formado por um **<SHIFT>+<ENTER>).**

Suponhamos que você tenha **vermelho** seguido de um *hard enter* e na linha seguinte **preto,** se na caixa "Procurar por" colocar **vermelho\npreto** e na caixa "Substituir por" colocar **vermelho-e-preto,** após a substituição, o *hard enter* terá morrido, sobrando **vermelho-e-preto.** O mesmo poderia ter sido feito, colocando nas caixas **\n** e **-e-,** respectivamente;

$ Casa com uma marca de parágrafo **(¶).**

Diferentemente do *hard enter*, não podemos definir na caixa "Procurar por" a cadeia no início da linha seguinte um uma *Expressão Regular* que case com ela. Assim sendo, não poderíamos agir como no exemplo anterior, preenchendo as caixas com **vermelho\npreto** e com **vermelho-e-preto** respectivamente, porém funcionaria se as caixas fossem preenchidas com **$** e com **-e-.**

\x Quando temos um **\xNNNN,** onde **NNNN** são algarismos hexadecimais **[0-9A-Fa-f]** na caixa "Procurar por", o BrOffice.org localizará (e eventualmente

trocará) o caractere definido pelo código hexadecimal formado por **NNNN.** Se este código estiver na caixa "Substituir por", será tratado como um literal.

Isso me lembra um macete de edição que uso. Ao longo de um texto grande, escrevemos diversos ordinais como **2ª.** Isso pode ser feito em **Inserir → Caractere especial,** mas fazer isso um monte de vezes num texto, enche o saco. Como faço? Para fazer **2ª,** como disse, escrevo **2.a.** e assim vou fazendo com todos os ordinais até terminar de escrever o documento.

Ao final, coloco na caixa "Procurar por" a *Expressão Regular* **([0-9])\.a\.,** e na caixa "Substituir por" coloco um **$1ª,** ou seja o texto que casou com o grupo (um algarismo) seguido do ordinal. Em seguida clico em "Substituir todos". Repito mais uma vez esta operação para substituir os ordinais masculinos como em **12º**

Vou terminar este texto com uma dica que demorei muito para descobrir. Ela é muito útil quando se copia um texto de um navegador e se cola no BrOffice.org. Normalmente o nove texto fica com um monte de *hard enter*, não é?

Se você quiser trocar todos *hard enter* **(↵)** por um parágrafo **(¶),** primeiramente localize todos os **\n,** clicando em "Substituir todos". Agora coloque na caixa "Substituir por" o mesmo **\n,** já que nesta caixa ele equivale ao parágrafo. Agora basta clicar em "Substituir todos". Bizarro...

Como vocês viram ainda existem algumas coisas a serem acertadas e outras a serem feitas, mas o concorrente direto do BrOffice.org, o MS Office, nem sonha ter *Expressões Regulares*. Creio que essas pequenas alterações e incrementos sejam sanados com a nova versão (bastante modificada), descrita em http://userguide.icu-project.org/strings/regexp.