Julio Neves

Dá pra fazer em SHELL?

dicas-l
www.dicas-l.com.br

Sumário

Prefácio

Muito me alegra o fato de você gostar de *Shell* e ter acompanhado (ou, pelo menos, tentado acompanhar) os vídeos que desenvolvi com algumas pequenas dicas desta linguagem.

Confesso que quando fiz os vídeos minha preocupação não era ensinar *Shell*, já que o assunto é muito vasto e não seria meia dúzia de vídeos que conseguiria passar um material tão vasto. Tinha três preocupações:

1. Mostrar comandos ou técnicas que em 90% dos casos que via, estavam usados de maneira indevida. Acho que o workshop que melhor representou esse meu desejo foi a trilogia **if + test + conectores;**
2. Mostrar soluções de problemas usando poucas linhas de código, como nos *workshops* sobre o comando paste e o de *one-liners;*
3. Mostrar comandos ou técnicas daqueles que as pessoas olham e torcem o nariz, porque não fazem a menor ideia do que aquele "troço", como foi as dicas sobre yad e xargs.

Essa nova coleção de dicas continua sem a pretensão de ensinar *Shell*, tendo como diretriz ir mostrando comandos, técnicas e macetes pouco conhecidos como:

- A suíte ImageMagick (formada por 11 utilitários, mas que os poucos que já ouviram falar dela, sabem usar um pouquinho só do utilitário convert);
- O yad, que é conhecido como o zenity com esteróides;
- O notify-send;
- O getopts que separa as opções passadas para o programa; coprocessos e
- Outros

Tem também algumas coisinhas que quebram o maior galho, tais como:

- Named Pipes;
- Substituição de processos;
- A variável $IFS;

- Como receber dados via *pipe* e como mandar dados para a saída de erros (no programa colunador.sh).

E ostras (já que o assunto é *Shell*) coisas mais...

Pessoal, espero que vocês gostem e aproveitem. Foi feito de muito boa vontade com o objetivo de divulgar essa excelente e pouquíssima conhecida linguagem *Shell*.

1. IFS: Inter Field Separator

O *Shell* tem uma variável interna chamada IFS - *Inter Field Separator* (será Tabajara? :) - cujo valor *default* podemos obter da seguinte forma:

```
$ echo "$IFS" | od -h
0000000 0920 0a0a
0000004
```

O programa od com a opção -h foi usado para gerar um *dump* hexadecimal da variável. E lá podemos ver:

Valor Hexadecimal	Significado
09	<TAB>
20	Espaço
0a	<ENTER>

Ou seja os separadores entre campos (tradução livre de IFS) *default* são o <TAB>, o espaço em branco e o <ENTER>. O IFS é usado em diversas instruções, mas seu uso é muito comum em par com o for e/ou com o read. Vejamos mais um exemplo:

```
$ cat script1.sh
#!/bin/bash
while read linha
do
    awk -F: '{print $1}' <<< $linha > /dev/null
done < /etc/passwd
```

Como podemos ver este *script* não faz nada (sua única saída foi enviada para /dev/null para não deturpar os tempos de execução), mas vamos usá-lo para avaliação dos tempos de execução.

Este *script* foi alterado, trocando o awk pelo cut, e ficando com a seguinte cara:

```
$ cat script2.sh
#!/bin/bash
while read linha
do
     echo $linha | cut -f1 -d: > /dev/null
done < /etc/passwd
```

Mais uma outra alteração, usando somente intrínsecos do *Shell* desta vez tomando partido do IFS:

```
$ cat script3.sh
#!/bin/bash
while IFS=: read user lixo
do
     echo $user> /dev/null
done < /etc/passwd
```

Neste último exemplo, transformamos o separador padrão em dois-pontos (:) e usamos sua propriedade em conjunto com o read, isto é, o primeiro campo veio para a variável user e o resto para a variável lixo.

É interessante notar que alguns comandos do *Shell* permitem que se troque o valor de algumas variáveis do sistema somente durante o tempo de sua execução e foi esse macete que apliquei no IFS que acabamos de ver, que ficou com o valor de dois pontos (:) somente durante a execução do read.

Quer ver outro caso? Então olha só:

```
$ echo $LANG       # Contém a linguagem em uso
pt_BR.UTF-8
$ date             # Em pt_BR.UTF-8
Sáb Mar 17 18:36:56 -03 2018
$ LANG=en_EN date  # Em inglês
Sat Mar 17 18:37:56 -03 2018
$ echo $LANG       # Voltou ao pt_BR.UTF-8?
pt_BR.UTF-8
```

Em seguida, fiz um *script* usando *Shell* puro. Repare a construção ${linha%%:*}, que é um intrínseco (*built-in*) do *Shell* que serve para excluir da variável linha o maior casamento com o padrão especificado (:* - que significa "de dois-pontos em diante"), ou seja, excluiu de linha tudo a partir do último dois pontos, contando da direita para a esquerda.

```
$ cat script4.sh
#!/bin/bash
while read linha
do
     echo ${linha%%:*} > /dev/null
done < /etc/passwd
```

Para finalizar, adaptei o *script* escrito pelo incansável Rubens Queiroz que, exceto pelo awk, é *Shell* puro.

```
$ cat script5.sh
#!/bin/bash
for user in `awk -F: '{print $1}' /etc/passwd`
```

```
do
    echo $user > /dev/null
done
```

Vou criar mais dois *script*s, que, no duro, são simples comandos porque o *loop* é desnecessário, mas para esse estudo são importantes:

```
$ cat script6.sh
#!/bin/bash
cut -f1 -d: /etc/passwd > /dev/null
$ cat script7.sh
#!/bin/bash
awk -F: '{ print $1; }' /etc/passwd > /dev/null
```

Agora, o mais importante: reparem os tempos da execução de cada um deles:

```
$ Vermelho=$(tput setaf 1)
$ Branco=$(tput setaf 7)
$    # Saída vermelha em fundo preto para realçar e homenagear meu mengão
;)
$ for i in script[1-7].sh
> do
>     echo $Vermelho===Tempos do $i===$Branco
>     time $i
> done
===Tempos do script1.sh===

real 0m0.065s
user 0m0.004s
sys  0m0.000s
===Tempos do script2.sh===

real 0m0.032s
user 0m0.008s
sys  0m0.004s
===Tempos do script3.sh===

real 0m0.002s
user 0m0.000s
sys  0m0.000s
===Tempos do script4.sh===

real 0m0.002s
user 0m0.000s
sys  0m0.000s
===Tempos do script5.sh===

real 0m0.003s
user 0m0.000s
sys  0m0.000s
```

```
===Tempos do script6.sh===

real 0m0.002s
user 0m0.000s
sys  0m0.000s
===Tempos do script7.sh===

real 0m0.003s
user 0m0.000s
sys  0m0.000s
```

Reparem que estas diferenças de tempo foram obtidas para um arquivo com somente 41 linhas. Veja:

```
$ wc -l /etc/passwd
41 /etc/passwd
```

Um outro uso interessante do IFS é o que vemos a seguir, primeiramente usando o IFS *default* que como vimos é <TAB>, Espaço e <ENTER>:

```
$ Frutas="Pera Uva Maçã"
$ set $Frutas
$ echo $1
Pera
$ echo $3
Maçã
```

Agora, vamos alterar o IFS para fazer o mesmo com uma variável qualquer, e para tal vamos continuar usando o famigerado /etc/passwd:

```
$ Root=$(head -1 /etc/passwd)
$ echo $Root
root:x:0:0:root:/root:/bin/bash
$ oIFS="$IFS"
$ IFS=:
$ set - $Root
$ echo $1
root
$ echo $7
/bin/bash
$ IFS="$oIFS"
```

Neste artigo pretendi mostrar duas coisas:

1. O uso do IFS que, infelizmente para nós, é uma variável pouco conhecida do *Shell* e
2. Mostrar que o *Shell* não é lento, normalmente é mal usado/programado. Quanto mais intrínsecos do *Shell* usamos, mais veloz e performático fica o *script*,

igualando em tempo até um *loop* de instruções com simples e únicos comandos.

2. Sincronismo de processos assíncronos com named pipes

Vamos falar hoje em *named pipes*. Você sabia que pode sincronizar 2 ou mais processos assíncronos, trocando informações entre eles usando esta técnica? Deixa eu te mostrar: abra 2 terminais no mesmo diretório e em um deles faça:

```
$ mkfifo paipi
$ ls -l paipi
prw-r--r-- 1 julio julio 0 Nov  4 18:08 paipi
```

Viu!? É um arquivo do tipo p e se o seu ls for colorido, verá que seu nome tem uma cor de burro quando foge. Agora em um terminal escreva:

```
$ cat paipi
```

Calma, não se desespere! Ele não congelou (pinguim não congela, janelas congelam ;), ele está ouvindo uma ponta do *named pipe*, esperando que se fale algo na outra ponta. Então vamos para o outro terminal para falar. Redirecione qualquer saída para o *named pipe* que ela "miraculosamente" aparecerá no primeiro terminal, que a esta altura já não terá aparência de "congelado". Por exemplo, faça:

```
$ ls -l > paipi
```

E dessa forma podemos trocar dados entre 2 processos. Genial, não é?

3. Substituição de Processos

O *Shell* também usa os *named pipes* de uma maneira bastante singular, que é a substituição de processos (*process substitution*). Uma substituição de processos ocorre quando você põe um < ou um > grudado na frente do parêntese da esquerda. Teclando-se o comando:

```
$ cat <(ls -l)
```

Resultará no comando ls -l executado em um sub*Shell* como é normal, porém redirecionará a saída para um *named pipe* temporário, que o *Shell* cria, nomeia e depois remove. Então o cat terá um nome de arquivo válido para ler (que será este *named pipe* e cujo dispositivo lógico associado é /dev/fd/63), e teremos a mesma saída que a gerada pela listagem do ls -l, porém dando um ou mais passos que o usual.

Como poderemos constatar isso? Fácil ... Veja o comando a seguir:

```
$ ls -l >(cat)
l-wx------ 1 jneves jneves 64 Aug 27 12:26 /dev/fd/63 -> pipe:[7050]
```

É... Realmente é um *named pipe.*

Você deve estar pensando que isto é uma maluquice de nerd, né? Então suponha que você tenha 2 diretórios: dir e dir.bkp e deseja saber se os dois estão iguais (aquela velha dúvida: será que meu backup está atualizado?). Basta comparar os dados dos arquivos dos diretórios com o comando cmp, fazendo:

```
$ cmp <(cat dir/*) <(cat dir.bkp/*) || echo backup furado
```

ou, melhor ainda:

```
$ cmp <(cat dir/*) <(cat dir.bkp/*) >/dev/null || echo backup furado
```

Este é um exemplo meramente didático, mas são tantos os comandos que produzem mais de uma linha de saída, que serve como guia para outros. Eu quero gerar uma listagem dos meus arquivos, numerando-os e ao final dar o total de arquivos do diretório corrente:

```
while read arq
  do
      ((i++)) # assim nao eh necessario inicializar i
```

```
    echo "$i: $arq"
done < <(ls)
echo "No diretorio corrente (``pwd``) existem $i arquivos"
```

Tá legal, eu sei que existem outras formas de executar a mesma tarefa. Mas tente fazer usando while, sem usar substituição de processos que você verá que este método é muito melhor.

4. Algumas implementações do Bash 4.0

4.1. Novas substituições de parâmetros

```
${parâmetro:n}
```

Equivale a um cut com a opção -c, porém muito mais rápido. Preste atenção, pois neste caso, a origem da contagem é zero.

```
$ TimeBom=Flamengo
$ echo ${TimeBom:3}
 mengo
```

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

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

4.2. Substituição de chaves

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.

```
$ 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} # Incremento 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
$ eval \>{a..c}.{ok,err}\;
$ 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).

5. Colunador

É muito raro vermos um *script Shell* que trabalhe devidamente com as opções de ***stdin***, ***stdout*** e ***stderr***, ou seja, entrada primária, saída primária e saída de erros primária. O *script* a seguir, habilita a captura dos dados de entrada pelo *pipe* (|), por um redirecionamento da entrada (<) ou por passagem de parâmetro e manda os erros para a saída de erros padrão, o que permite redirecionar os erros para onde você desejar, ou até trancar a saída de erro (2>&-) para evitar essas mensagens.

```
$ cat colunador.sh
#!/bin/bash
#  Recebe dados via pipe, arquivo ou passagem de parâmetros e
#+ os coloca em coluna numerando-os

if  [[ -t 0 ]]        # Testa se tem dado em stdin
then
     (($# == 0)) || {  # Testa se tem parâmetro
           echo Passe os dados via pipe, arquivo ou passagem de
parâmetros >&2  # Redirecionando para stderr
           exit 1
     }
     Parms="$@"
else
    Parms=$(cat -)    # Seta parâmetros com conteúdo de stdin
fi
set $Parms
for ((i=1; i<="$#"; i++))
{
    Lista=$(for ((i=1; i<="$#"; i++)); { printf "%0${##}i %s\n" $i "${!
i}"; })
}
echo "$Lista" | column -c $(tput cols)
$ echo {A..Z} | colunador.sh
01 A   05 E   09 I   12 L   15 O   18 R   21 U   24 X
02 B   06 F   10 J   13 M   16 P   19 S   22 V   25 Y
03 C   07 G   11 K   14 N   17 Q   20 T   23 W   26 Z
04 D   08 H
```

Esse *script* começa testando se o FD 0 (zero), que é o descritor da entrada primária, está aberto em um terminal e isso ocorrerá se o programa não estiver recebendo dados por um *pipe* (|), ou por um redirecionamento de entrada (<), ou ainda por um *here strings*.Assim sendo vamos testar se a quantidade de parâmetros passados ($#) é zero, ou seja: os dados não vieram por ***stdin*** nem por passagem de parâmetros quando então é indicado um erro, que é desviado para ***stderr*** (>&2).

Bem, agora vamos tratar os dados que recebemos, passando-os para a variável $Parms. Caso eles tenham vindo por passagem de parâmetros, aí é simples, é só atribuir a $Parms o valor de todos os parâmetros passados ($@).

Caso os dados tenham vindo de ***stdin***, temos mais 2 macetes:

1. A quase totalidade dos utilitários do *Shell*, aceitam o hífen como representando os dados recebido de ***stdin***. Veja isso:

```
$ cat arq
Isto é um teste
Outra linha de teste
$ echo Inserindo uma linha antes de arq | cat - arq
Inserindo uma linha antes de arq
Isto é um teste
Outra linha de teste
```

Ou seja, o comando primeiramente listou o que recebeu da entrada primária e em seguida o arquivo arq.

2. O outro macete é o comando set. Para entendê-lo vá para o *prompt* e faça:

```
$ set a b c
$ echo $1:$2:$3
a:b:c
```

Conforme você viu, este comando atribuiu os valores passados como os parâmetros posicionais.

Uma vez entendido isso, vimos que os valores recebidos de ***stdin*** foram colocados em $Parms que por sua vez teve seu conteúdo passado para os parâmetros posicionais, unificando dessa forma, os dados de qualquer tipo de entrada recebida.

O for terminará quando alcançar a quantidade de parâmetros passados ($#) e o comando printf, em seu primeiro parâmetro, (%0${##}d), diz que a linha será formatada (%) com um decimal (d) com ${##} algarismos, preenchido com zeros à esquerda (0).

Veja isso para entender melhor: ${#var} devolve o tamanho da variável var e também sabemos que $# retorna a quantidade de parâmetros passados. Juntando os dois, vemos que ${##} devolve quantos algarismos têm a quantidade de parâmetros. Faça este teste para você ver:

```
$ set {a..k}            # Passando 11 parâmetros
$ echo ${##} algarismos
2 algarismos
```

```
$ set a b c             # Passando 3 parâmetros
$ echo ${##} algarismos
1 algarismos
```

Uma vez montada a lista, é só passá-la para o comando tr, cuja opção -T diz que a formatação não é para impressora e o -8 é para dividir a saída em 8 colunas (trabalhar para que? O *Shell* já tem comando pronto para tudo!)

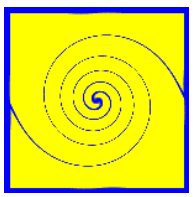
6. Criando Animações com ImageMagick

O pacote ImageMagick é extremamente poderoso e possui funcionalidades que muitos desconhecem. O *script* entorta.sh, cria uma animação simples, que pode ser visualizada em qualquer browser web.

script entorta.sh

```
# cat entorta.sh
#!/bin/bash
#  Montando uma animação no ImageMagick

#  Vou fazer uma figura que servirá como base da animação.
#+ Ela será composta por 1 quadrado azul com 2 retângulos
#+ inscritos, formando a figura base.png
convert -size 150x150 xc:blue                   \
    -fill yellow -draw 'rectangle 5,5 145,72.5' \
    -fill yellow -draw 'rectangle 5,77.5 145,145' base.png

for ((i=1; i<=40; i++))
{
    #  Gero 40 imagens de trabalho, torcendo (swirl)
    #+ a imagem base.png com incrementos de 35 graus
    convert -swirl $((35*$i)) base.png Trab_$i.png

    #  Concateno o nome de todas as imagens em Arqs

    Arqs="$Arqs Trab_$i.png"                    }

#  A animação a seguir é garantida pela opção -coalesce.
#+ A opção -dither é usada para diminuir a perda de
#+ qualidade com a redução da qtd de cores.
#+ A opção -colors 32 reduz a qtd de cores.
#+ A opção -layers optimize, usada com a anterior
#+ visam acelerar o processo.

convert -coalesce -dither -colors 32 -layers optimize $Arqs Anim.gif
#  Agora, se vc abrir Anim.gif no browser, verá a animação.
```

A figura abaixo é o resultado final do processo:



7. notify-send

Essa dica é para aqueles que, como eu, conhecem *Zenity* e notaram que o diálogo **--notification** do YAD não possui a ferramenta **message**. Como acho este acessório do diálogo **--notification** muito útil, mandei um e-mail para o autor do YAD, que me disse que não implementaria esta facilidade pois ela já é contemplada pelo comando **notify-send** e que necessita da **libnotify** e ele queria que a única dependência do YAD fosse o GTK2.

Em para suprir esta falta, vou me afastar um pouco do objetivo e dar uma explicação rapidinha sobre a sintaxe do **notify-send**, que caso não esteja instalado no seu computador, você pode fazê-lo com a seguinte linha (para Debian e seus correlatos):

```
$ sudo apt-get install libnotify-bin
```

A sintaxe é a seguinte:

```
notify-send [OPCOES...] <TITULO> [TEXTO]
```

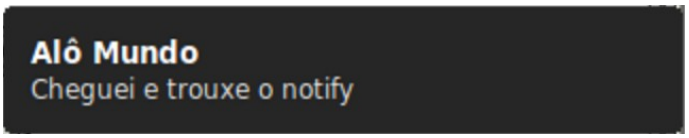
Um exemplo bem bobo passando somente o título **TITULO**:

```
$ notify-send "Alô Mundo"
```

Fazendo isso aparecerá no canto superior esquerdo da tela a seguinte figura:

Mas poderíamos também preencher o TEXTO:

```
$ notify-send "Alô Mundo" "Cheguei e trouxe o notify"
```



Podemos também colocar um ícone:

```
$ notify-send "Alô Mundo" "Cheguei e trouxe o notify" \
      -i gtk-dialog-info
```

O `notify-send` também pode ser usado para te avisar o fim de um *job* muito demorado. Para isso basta fazer um *script* que seja mais ou menos assim:

```
$ cat dispara_job.sh
#!/bin/bash
#  Avisa o término de um job, usando notification-send .
#  Esse script deverá ser executado em background
eval "$1"
notify-send -i terminal "Acabou" "\nFim do job $1"
```

Para testá-lo use algo como:

```
$ dispara_job.sh 'time find / > /dev/null 2>&1' &
```

Assim você lerá em *background* (`&`) o nome de todos os arquivos de seu computador mandando a saída primária (***stdout***) para o buraco negro (`> /dev/null`) e a saída de erros (***stderr***) também (`2>&1`). Ele também funciona em *foreground*, mas aí não te traria vantagem alguma, pois seu terminal ficaria preso até que ele terminasse.

Além dos ícones citados na definição do uso de `–button`, o `notify` (e o yad também) aceita(m) os ícones na tabela a seguir, mas note que esses nomes devem ser precedidos pela cadeia `notification-`. Por exemplo: não use `-i audio-next`, isso não funciona. Use `-i notification-audio-next`, da mesma forma `-i audio-play` deve ser substituído por `-i notification-audio-play`. Essa simplificação foi feita somente para tornar os nomes menores, permitindo que sejam tabulados em dois por linha, de forma a tomar menos espaço no artigo.

Inserir notification-	Inserir notification-
audio-next	audio-play
audio-previous	audio-volume-high
audio-volume-low	audio-volume-medium
audio-volume-muted	audio-volume-off
battery-low	device-eject
device-firewire	display-brightness-full
display-brightness-high	display-brightness-low
display-brightness-medium	display-brightness-off
GSM-3G-full	GSM-3G-high
GSM-3G-low	GSM-3G-medium
GSM-3G-none	GSM-disconnected
GSM-EDGE-full	GSM-EDGE-high
GSM-EDGE-low	GSM-EDGE-medium
GSM-EDGE-none	GSM-full
GSM-H-full	GSM-H-high
GSM-high	GSM-H-low
GSM-H-medium	GSM-H-none
GSM-low	GSM-medium
GSM-none	keyboard-brightness-full
keyboard-brightness-high	keyboard-brightness-low
keyboard-brightness-medium	keyboard-brightness-off
message-email	message-IM
network-ethernet-connected	network-ethernet-disconnected

Inserir notification-	Inserir notification-
network-wireless-disconnected	network-wireless-full
network-wireless-high	network-wireless-low
network-wireless-medium	network-wireless-none
power-disconnected	

Se quiser, ainda é possível temporizar, mas nesse caso, tenho uma observação a fazer:

> Usando o parâmetro de urgência (`-u`) como `critical`, o diálogo só sai da tela quando clicado;

Experimente fazer:

```
$ notify-send "Alô Mundo" "\nCheguei e trouxe o notify" \
      -i gtk-dialog-info -u critical
```

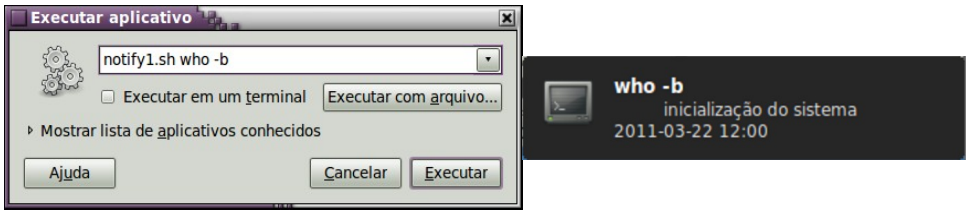
Ainda existem outras opções que não exploramos, por estarem saindo do propósito desta publicação. Abaixo uma tabela com essas opções, que espero que você explore-as, são que este utilitário é muito útil, podendo inclusive passar avisos sobre eventos tais como alterações no volume, chegada de e-mail, *instant messages*, eventos de rede,... Você encontrará todos os detalhes em http://www.galago-project.org/specs/notification/.

Opções	Ações
`-u NIVEL`	Especifica o nível de urgência (`low`, `normal`, `critical`)
`-c TIPO[,TIPO...]`	Especifica a categoria da notificação
`-h TIPO:NOME:VALOR`	Especifica base de dados extra para passar. Os tipos válidos são `int`, `double`, `string` e `bytes`

```
$ cat notify1.sh
#!/bin/bash
#  Executa um comando dando a resposta num
#+ balão do tipo notify-send. Ideal para
#+ usar com <ALT>+F2, sem abrir um terminal
```

```
#+ Thanks Karthik

Saida=$(eval "$*" 2>/dev/null) || {
    yad --text \"Comando \"$*\" errado\" --button gtk-ok
    exit 1
    }
notify-send -t $((1000+300*$(echo -n $Saida | wc -w))) -u low -i gtk-
dialog-info "$*" "$Saida"
```

Se você não quiser abrir um terminal para executar um comando, basta você usar este programa. Como fazê-lo? É simples: após colocar este *script* no seu diretório *home*, pressione **<ALT>+F2** que aparecerá a caixa "**Executar aplicativo**". Nesta caixa digite o nome do *script* (no caso **notify1.sh**) seguido do comando que você deseja executar e que terá sua saída apresentada em um menu *pop-up* gerado pelo **notify-send**. Vejamos sua utilização para o comando **who -b** que informa a data e hora do último *boot.*



Um programa para você verificar a carga da bateria de seu *notebook*.

```
$ cat notify2.sh
#!/bin/bash
# Para pegar a carga da bateria
Carga=$(acpi -b | sed 's/.* //;s/%//')
#  Criando o índice do ícone já que o nome do
#+ ícone é no formato notification-battery-xxx
#+ e xxx deve ser 000, 020, 040, 060, 080,
#+ ou 100, dependendo da carga da bateria
Nivel=$(case $Carga in
    ([01][0-9]) echo 000;;
    ([23][0-9]) echo 020;;
    ([45][0-9]) echo 040;;
    ([67][0-9]) echo 060;;
    ([89][0-9]) echo 080;;
            (*) echo 100;;
esac)
notify-send -i notification-battery-$Nivel "Bateria" \
    "\nO nível de carga da bateria está em $Carga%"
```

Veja como é a saída do **acpi -b**, que é quem me informa a carga reminiscente na bateria:

```
$ acpi -b
Battery 0: Unknown, 98%
```

O `sed` foi usado para limpar essa saída, deixando somente o `98`.

Experimente fazer um *script* para pegar a temperatura de seu *notebook*. Para fazer isso, use o seguinte comando:

```
$ acpi -t
Thermal 0: ok, 29.8 degrees C
Thermal 1: ok, 46.0 degrees C
```

Veja o *script* que fiz para mudar o estado do *touchpad* do meu *notebook*:

```
$ cat tp-on-off
#!/bin/bash
#  O meu note não tem hot key para Habilitar/Desabilitar
#+ o touchpad, então eu fiz esse bacalho

if [ $(synclient | sed '/touchpad/I !d; s/^.*= //') -eq 0 ]
then
    notify-send -i touchpad-disabled.svg "TouchPad OFF" \
        '<b><big>Desliguei</big></b>\to touchpad'
    synclient TouchpadOff=1
else
    notify-send -i input-touchpad "TouchPad ON" \
        '<b><big>Liguei</big></b>\to touchpad'
    synclient TouchpadOff=0
fi
```

O comando `sed` só não deleta `(!d)` a linha que possui a cadeia *touchpad*, ignorando a caixa das letras `(I)`. Veja:

```
$ synclient | sed '/touchpad/I!d'
    TouchpadOff             = 0
```

e ainda no `sed`, foi substituído `(s)` tudo que existia antes do estado do *touchpad* `(s/^.*= /)` por vazio `(//)`, restando desta forma somente o seu estado atual (que no exemplo dado era zero `(0)`

8. YAD - Yet Another Dialog

Sou fã de carteirinha do *Shell* e frequentemente preciso melhorar a interface de um *script* com o usuário. Para isso, invariavelmente usava o *Zenity* que é extremamente fácil e melhora muito a apresentação.

Minha satisfação como o *Zenity* só não era total porque todos que conhecem um pouquinho de GTK 2, sabem que se pode tirar muito mais dessa linguagem do que o *Zenity* tira. Sempre esperei que a qualquer momento o *gnome* lançaria uma nova versão deste software que seria arrasadora, viria com todos os incrementos que eu (e acho que todos) aguardavam.

Estava enganado, acompanhei o lançamento de diversas revisões do software mas elas simplesmente tratavam *bugs* e apresentação. Infelizmente nada de inovação. Porém um dia descobri o *YAD*.

YAD significa *Yet Another Dialog*, mas sua tradução do russo é "veneno" (e essa é a inspiração de seu ícone) e se define como um *fork* do *Zenity*. Porém o *YAD*, distancia-se um pouco dele por não utilizar bibliotecas descontinuadas como *libglade* ou *gnome-canvas*. Utiliza somente o GTK 2, o que lhe dá uma maior portabilidade dentro do ambiente *n?x.

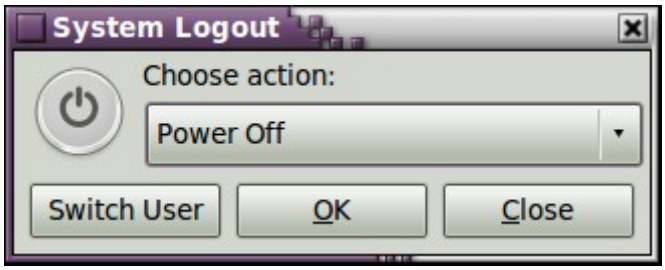
Poderia defini-lo como um *Zenity* com muitos aperfeiçoamentos, como botões customizados, diálogos adicionais, menu *pop-up* no ícone do `notification` (sim é o mesmo `notification` do *Zenity*, porém mais incrementado), diálogo para montar formulários complexos e muito mais.

Este artigo não é para mostrar como se instala, isso eu sei que vocês sabem se virar. O que quero é mostrar algumas (infelizmente muito poucas) novidades que o *YAD* traz para trabalharmos com interface GUI em *Shell*. Vamos ao que interessa:

Este é um dos poucos exemplos que achei pela internet e está no wiki do *YAD*. Lá ele coloca o código completo para dar *shutdown*, *reboot*, ... Mas aqui executaremos somente a parte do *YAD* direto na linha de comandos.

```
$ yad --width 300 --entry --title "System Logout" \
    --image=gnome-shutdown                         \
    --button="_Switch User:2"                      \
    --button="gtk-ok:0" --button="gtk-close:1"     \
```

```
    --text "Choose action:"                          \
    --entry-text                                     \
    "Power Off" "Reboot" "Suspend" "Logout"
```



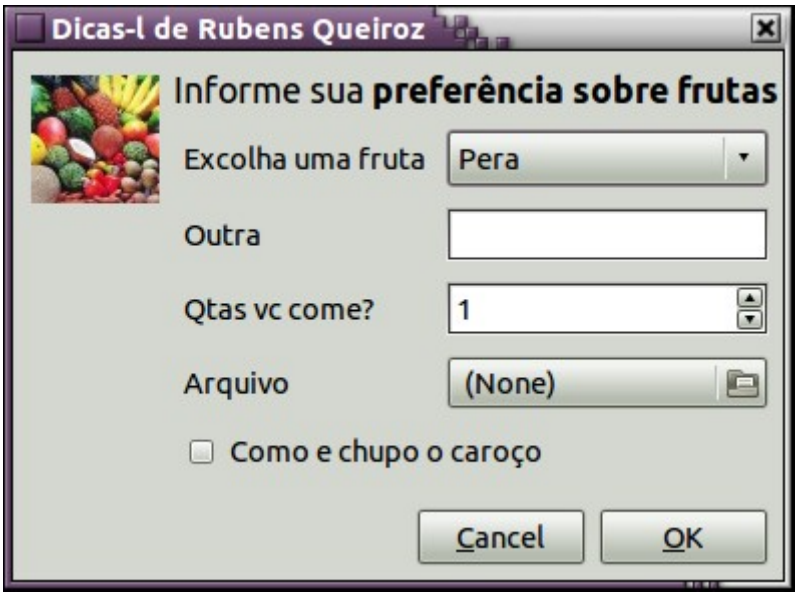
Aqui você pode ver uma ComboBox, diversos botões (um deles *customizado*) e um código extremamente simples. Repare que após a declaração do nome de cada botão, tem um `:N`, onde `N` é o valor que retorna em `$?` quando este botão for clicado. Simples, não?

Um outro exemplo usando o diálogo `--form`:

```
$ yad --form --title "Dicas-l de Rubens Queiroz" \
    --text "Informe sua preferência sobre frutas" \
    --image Frutas.jpg \
    --field "Excolha uma fruta:CB" \
    --field Outra
    --field "Qtas vc come?:NUM" \
    --field "Arquivo:FL" \
    --field "Como e chupo o caroço:CHK" \
    'Pera!Uva!Maçã!Manga' \
    "" \
    '1!10!1'
```

Com certeza você sabe que as contrabarras foram colocadas somente para tornar o texto mais visível e fácil de comentar, mas poderia ter sido escrito em somente uma linha.

O diálogo `--form` permite que eu use uma opção `--field` que aceita um monte de modificadores. Aqui usei, na ordem, `:CB`, nenhum, `:NUM`, `:FL` e `:CHK` que são, respectivamente, *ComboBox*, Entrada de texto, *SpinBox* numérico, Seletor de arquivos e uma *CheckBox* e para finalizar passei os valores das 3 primeiras. Veja o que gerou:



Ainda quero fazer uns comentários:

- Repare no título (`--text`) que usei *tags* de formatação;
- Veja a facilidade de colocar qualquer imagem no diálogo usando a opção `--image`;
- Especifiquei os textos do *ComboBox* separando-os com um ponto de exclamação (`!`). Este é o *default*, mas pode ser alterado;
- O campo *Outra*, por não ter tido nenhum valor especificado assume que é uma simples entrada de dados.
- O campo *Qtas vc come?* É um *SpinBox* e seu valor foi especificado com *Mínimo!Máximo!Incremento.*
- Quando o campo Arquivo for clicado, abrirá um super diálogo para seleção de arquivo;
- O campo *Arquivo* e o *CheckBox* não foram inicializados por serem os últimos e por isso poderem assumir seus valores *default* (*Nome* e *FALSE*, respectivamente).

Fora esses pequenos exemplos ainda existem muitos outros diálogos que não foram implementados no *Zenity* como diálogos *drag'ndrop*, diálogos com ícones para escolha de aplicativos, dialogo para escolha de fonte, diálogo para escolha de cores. E mesmo muitos dos diálogos existentes no *Zenity*, no *YAD* passam a ter mais opções.

Para finalizar, experimente fazer:

```
$ ls -l | yad --text-info --fontname "mono 10" --width 600 --height 300
$ ls -l | zenity --text-info --width 600 --height 300
```

e veja a falta que a opção `--fontname` faz no *Zenity*.

Só para clarear o que falei, existe um diálogo `--fonte` para escolha de fontes e uma opção `--fontname` para especificar qual fonte um determinado diálogo usará. Agora, para finalizar mesmo vou mostrar o exemplo anterior, porém com escolha da fonte:

```
#! /bin/bash
#  Exemplo de uso dos diálogos --font
#+ e --text-info

Fonte=$(yad --font --title "Escolha de fontes" \
    --text "Escolha uma fonte
Dê preferência a fontes monoespacejadas"      \
    --height=300 --width 600) || exit 1

ls -l | eval yad --text-info               \
    --width=600 --height 300               \
    --title \"Listagem do diretório $PWD\" \
    --fontname \'$Fonte\'
```

Espero que você instale o *YAD* e teste esses exemplos que dei. Tenho certeza que você se amarrará no software.

9. Fatiando opções com o getopts

Analisar uma linha de comando e quebrá-la em cada uma de suas opções é uma tarefa complicada, pois, só falando em duas opções (`-A` e `-B`) sendo que `-B` pode ou não ter argumentos (ARG), teríamos de pesquisar por:

-AB	-ABARG	-AB ARG
-A -B	-A -BARG	-a -B ARG
-BA	--	--
-B -A	-BARG -A	-B ARG -A

Isso sem falar que, se uma das opções for facultativa, a gama da análise (provavelmente em um comando `case`) seria o dobro dessa. Em virtude dessa complexidade e para aliviar a vida do programador, foi desenvolvido o nosso amigo `getopts`, que, por ser um intrínseco (*builtin*), é bastante veloz. Ele foi feito para suceder a antiga implementação que chamava-se `getopt` (sem o `s`), que não era intrínseco e tinha alguns bugs, mas seu único pecado é não aceitar opções longas dos comandos, previstas pelo GNU (como em `CMD --help`, p.ex.).

Só para melhorar o entendimento unificando a terminologia que será usada, pelos parágrafos anteriores você já deve ter reparado que chamamos de opção uma letra precedida de um sinal de menos ou traço (`-`) e argumentos são os textos requeridos por algumas opções ou simplesmente passados para a linha de comando.

Exemplo

```
cut -f 2 -s -d : /etc/passwd
```

Trecho	Terminologia
cut	Programa
-f -s -d	Opções
2	Argumento da opção -f
:	Argumento da opção -d
/etc/passwd	Argumento do programa

Sintaxe

```
getopts CADOPTS VAR [ARGS]
```

Onde:

CADOPTS	contém a cadeia de opções e suas eventuais necessidades de argumento. Quando uma opção aceita um argumento, ela deverá vir seguida de um dois pontos (:). Como veremos adiante esta variável receberá um ponto de interrogação (?) em caso de erro. Assim sendo, o ponto de interrogação (?) e os dois pontos (:) são especiais para o getopts e em virtude disso esses dois caracteres não podem ser usados como opções.
VAR	O comando terá de ser executado (normalmente dentro de um *loop* de while) tantas vezes quantas foram as opções válidas. A variável VAR, receberá em cada passada a opção (tirada de CADOPTS) que será analisada, ou terá uma sinalização (flag) de erro.
ARGS	Normalmente são pesquisadas as opções passadas em VAR, mas se os argumentos forem passados em ARGS, eles é que serão analisados, ou seja, a existência desse parâmetro diz ao getopts para analisar o conteúdo de ARGS em vez dos parâmetros posicionais.

9.1. Variáveis usadas pelo getopts:

OPTARG	Contém cada argumento das opções descobertas em cada volta do loop do getopts ou uma marca de erro que analisaremos à frente;
OPTIND	Contém o índice do parâmetro posicional em análise que você gerenciará para saber, pelo valor de OPTIND a opção que está sendo trabalhada;
OPTERR	É uma variável que quando tem valor 1 indica para Shell que os erros irão para a tela e este é o padrão pois é sempre inicializada com valor 1. Com valor zero, os erros são assinalados, mas não são exibidos.

9.2. Manipulação de erros

Como já vimos o getopts pode ser executado de dois modos:

Modo silencioso	Quando a variável do sistema $OPTERR for zero ou quando a cadeia que você passou em OPTARG começa por dois pontos (:);
Modo falador	Este é o normal. OPTERR é igual a um e a cadeia passada em OPTARG não começa por dois pontos (:).

Se for achada uma opção inválida, getopts põe um ponto de interrogação (?) em VAR. Se não estiver em modo silencioso dá uma mensagem de erro e esvazia OPTARG. Se estiver silencioso, o caractere que gerou a opção inválida é colocado em OPTARG.

Se um argumento requerido não for informado e getopts não estiver em modo silencioso, um ponto de interrogação (?) é colocado em VAR, OPTARG é esvaziada e é dada uma mensagem de erro. Caso esteja em modo silencioso, um dois pontos (:) é

colocado em `VAR` e `OPTARG` recebe o caractere da opção da qual não foi informado o argumento requerido

Quando o assunto é programação em bash, aconselho que, em seus *scripts*, sempre use o modo silencioso e monitore eventuais erros.

Como esse cara funciona?

Vamos fazer essa análise por exemplos, que é muito mais simples de aprender:

Exemplo

Suponha que um programa possa receber a opção `-C`. O *script*, para análise das opções deveria ser do tipo:

```
$ cat cut1.sh
#!/bin/bash

while getopts c Opc
do
    case $Opc in
        c)  echo Recebi a opção -c
            ;;
    esac
done
```

Como vamos trabalhar diversos exemplos em cima deste mesmo script, preferi já começá-lo com um case, quando nesse caso bastaria um `if`.

Vamos vê-lo funcionando:

1. Executando-o sem nenhuma opção, ele não produz nenhuma saída;
2. Usando a opção `-c`, que é o esperado:

```
$ cut1.sh -c
```

Recebi a opção `-c`

3. Usando a opção `-z`, que é inválida:

```
$ cut1.sh -z
./cut1.sh: opção ilegal -- z
```

Xiii, quem deu a mensagem foi o *Shell* e eu não soube de nada. Mesmo com erro, o programa continuaria em execução, porque o erro não é tratado pelo programa.

Então vamos alterá-lo colocando-o em modo silencioso, o que, como já vimos, pode ser feito iniciando a cadeia de opções (`CADOPTS`) com um dois pontos ou atribuindo zero à variável `$OPTERR`.

```
$ cat cut2.sh
#!/bin/bash

while getopts :c Opc
do
    case $Opc in
        c)  echo Recebi a opção -c
            ;;
       \?)  echo Caractere $OPTARG inválido
            exit 1
done
```

Agora coloquei um dois pontos à frente de `CADOPTS` e passei a monitorar um ponto de interrogação (`?`) no `case`, porque quando é achado um caractere inválido, o `getopts`, como já vimos, coloca um ponto de interrogação (`?`) na variável `$Opc`. Desta forma, listei `$OPTPARG` e em seguida dei `exit`.

Note que antes da interrogação coloquei uma contrabarra (`\`), para que o Shell não o expandisse para todos os arquivos que contém somente um caractere no nome.

4. Agora que já tenho o ambiente sob meu controle, vamos executá-lo novamente com uma opção não prevista:

```
$ cut2.sh -z
Caractere z inválido
$ echo $?
1
```

Agora aconteceu o que queríamos: deu a nossa mensagem e o programa abortou, passando 1 como código de retorno (`$?`).

Bem, já examinamos todas as possibilidades que a passagem de opções pode ter. Vamos agora esmiuçar o caso que uma opção que tenha parâmetro associado. Para dizer que uma opção pode ter um argumento, basta colocar um dois pontos (`:`) após a letra da opção.

Então, ainda evoluindo o programa de teste que estamos fazendo vamos supor que a opção `-c` requeresse um argumento. Deveríamos então fazer algo como:

```
$ cat cut3.sh
#!/bin/bash
```

```
while getopts :c: Opc
do
    case $Opc in
        c)  echo Recebi a opção -c
            echo Parâmetro passado para a opção -c foi $OPTARG
            ;;
       \?)  echo Caractere $OPTARG inválido
            exit 1
            ;;
        :)  echo -c precisa de um argumento
            exit 1
    esac
done
```

Agora introduzimos o caractere dois pontos (`:`) no case porque, como já vimos, quando um parâmetro não é localizado, o `getopts` em modo silencioso coloca um dois pontos (`:`) em `$Opc`, caso contrário, a falta de argumento é sinalizada, com um ponto de interrogação (`?`) nesta mesma variável.

Vamos então analisar todas as possibilidades:

1. Executando-o sem nenhuma opção, ele não produz nenhuma saída;

2. Passando a opção `-c` acompanhada de seu parâmetro, que é o esperado:

```
$ cut3.sh -c 2-5
Recebi a opção -c
Parâmetro passado para a opção -c foi 2-5
```

3. Passando a opção correta, porém omitindo o parâmetro requerido:

```
$ cut3.sh -c
-c precisa de um argumento
$ echo $?
1
```

Para finalizar esta série, vejamos um caso interessante. Desde o início, venho simulando nesse exemplo a sintaxe do comando `cut` com a opção `-c`. Para ficar igualzinho à sintaxe do `cut`, só falta receber o nome do arquivo. Vejamos como fazê-lo:

```
$ cat cut4.sh
#!/bin/bash

#  Inicializar OPTIND é necessário caso o script tenha
#+ usado getopts antes. OPTIND mantém seu valor
OPTIND=1
while getopts :c: Opc
do
```

```
    case $Opc in
        c)  echo Recebi a opção -c
            echo Parâmetro passado para a opção -c foi $OPTARG
            ;;
       \?)  echo Caractere $OPTARG inválido
            exit 1
            ;;
        :)  echo -c precisa de um argumento
            exit 1
    esac
    shift $((--OPTIND))
    Args="$@"
    echo "Recebi o(s) seguinte(s) argumento(s) extra(s): $Args"
done
```

E executando-o vem:

```
 $ cut4.sh -c 2-5 /caminho/do/arquivo
```

Recebi a opção -c

Parâmetro passado para a opção -c foi 2-5

Recebi o(s) seguinte(s) argumento(s) extra(s): /caminho/do/arquivo

Agora vamos dar um mergulho num exemplo um bem completo e analisá-lo. Para esse script interessam somente as opções -f, -u argumento e -C. Veja o código (mas este ainda não está 100%).

```
#!/bin/bash
printf "%29s%10s%10s%10s%10s\n" Comentário Passada Char OPTARG OPTIND
while getopts ":fu:C" VAR
do
    case $VAR in
        f) Coment="Achei a opção -f"
           ;;
        u) Coment="Achei a opção -u $OPTARG"
           ;;
        C) Coment="Achei a opção -C"
           ;;
       \?) Coment="Achei uma opção invalida -$OPTARG"
           ;;
        :) Coment="Faltou argumento da opção -u"
    esac
    printf "%30s%10s%10s%10s%10s\n" "$Coment" $((++i)) "$VAR" "$OPTARG"
"$OPTIND"
done
```

Agora vejamos a sua execução passando todos as opções juntas e sem passar o parâmetro requerido pela opção `-u` (repare os dois pontos (`:`) que seguem o u na chamada do `getopts` neste exemplo).

```
$ getop.sh -fCxu

                  Comentário   Passada    Char    OPTARG    OPTIND
           Achei a opção -f          1       f                    1
           Achei a opção -C          2       C                    1
 Achei uma opção invalida -x         3       ?         x          1
Faltou argumento da opção -u         4       :         u          2
```

Repare que a variável `$OPTIND` não foi incrementada. Vejamos então o mesmo exemplo, porém passando as opções separadas:

```
$ getop.sh -f -C -x -u

                  Comentário   Passada    Char    OPTARG    OPTIND
           Achei a opção -f          1       f                    2
           Achei a opção -C          2       C                    3
 Achei uma opção invalida -x         3       ?         x          4
Faltou argumento da opção -u         4       :         u          5
```

Ah, agora sim! `$OPTIND` passou a ser incrementado. Para fechar, vejamos um exemplo sem opção inválida e no qual passamos o parâmetro requerido pela opção `-u`:

```
$ getop.sh -f -C -u Util

                  Comentário   Passada    Char    OPTARG    OPTIND
           Achei a opção -f          1       f                    2
           Achei a opção -C          2       C                    3
      Achei a opção -u Util          3       u      Util          5
```

Algumas observações

1. Vimos que mesmo após encontrar erro, o `getopts` continuou analisando as opções, isso se dá porque o que foi encontrado pode ser um parâmetro de uma opção e não um erro;
2. Então como distinguir um erro de um parâmetro? Fácil: se a opção em análise requer argumento, o conteúdo de `$OPTARG` é ele, caso contrário é um erro;
3. Quando encontramos um erro devemos encerrar o programa pois o `getopts` só aborta sua execução quando:
    1. Encontra um parâmetro que não começa por menos (`-`);

2. Quando encontra um erro (como uma opção não reconhecida).

O parâmetro especial `--` marca o fim das opções, mas isso é uma convenção para todos os comandos que interagem com o Shell.

Então a nossa versão final do programa seria:

```
$ cat getop.sh
#!/bin/bash
function Uso
{
    echo "    $Coment
    Uso: $0 -f -C -u parâmetro" >&2
    exit 1
}
(($#==0)) && { Coment="Faltou parâmetro"; Uso; }

printf "%29s%10s%10s%10s%10s\n" Comentário Passada Char OPTARG OPTIND
while getopts ":fu:C" VAR
do
    case $VAR in
        f) Coment="Achei a opção -f"
           ;;
        u) Coment="Achei a opção -u $OPTARG"
           ;;
        C) Coment="Achei a opção -C"
           ;;
       \?) Coment="Achei uma opção invalida -$OPTARG"
           Uso
           ;;
        :) Coment="Faltou argumento da opção -u"
    esac
    printf "%30s%10s%10s%10s%10s\n" "$Coment" $((++i)) "$VAR" \
"$OPTARG" "$OPTIND"
done
```

10. bash - Expansão de Parâmetros

Com o comando `set` podemos passar parâmetros, então vamos fazer, direto no *prompt*, o seguinte:

```
$ set {a..k}
$ echo $@
a b c d e f g h i j k
$ echo $0
bash
```

Isto é, passamos para o Bash parâmetros da letra **a** até a letra **k**. Agora veja:

```
$ echo $1 - $6 - $11
a - f - a1
```

E foi aí que a vaca foi pro brejo? O `$11` é visto como o `$1` concatenado com `1`.

A partir do bash 2.0 começou-se a usar a Expansão de Parâmetros para resolver esse problema.

Uma Expansão de Parâmetros tem a seguinte forma: `${EXP_PARM}`. Então a primeira a ser criada foi para resolver esse problema. E a forma certa para listar essa variável ficou sendo:

```
$ echo ${11}
k
```

A partir daí então começaram a desenvolver uma série de Expansões de Parâmetros que atuam somente sobre variáveis - e por isso, no interior de uma Expansão de Parâmetros, os nomes das variáveis sobre as quais se está atuando não precisam ser precedidos por um cifrão (`$`) - e têm por princípio serem extremamente rápidas e aceitarem metacaracteres de expansão de arquivos (`*`, `?` e `[ ... ]`) na composição do seu escopo. Nesse texto vou citar as principais, dando exemplos.

Vamos fazer:

$ Var1=cadeia

Passando para caixa alta

```
$ Var1=${Var1^}; echo $Var1    # Só a primeira letra
Cadeia
```

```
$ Var1=${Var1^^}; echo $Var1   # Todas as letras
CADEIA
```

Passando para caixa baixa

```
$ Var1=${Var1,}; echo $Var1    # Só a primeira letra
cADEIA
$ Var1=${Var1,,}; echo $Var1   # Todas as letras
cadeia
```

Para complementar texto

```
$ unset Var2                   # Matei Var2
$ echo Var1 tem $Var1 ${Var2:+ e Var2 tem $Var2}
Var1 tem cadeia
$ Var2=STRING                  # Atribui valor a Var2
$ echo Var1 tem $Var1 ${Var2:+ e Var2 tem $Var2}
Var1 tem cadeia e Var2 tem STRING
```

Para criar valor padrão

```
$ read -p "Login ($USER): " LN; LN=${LN:-$USER}; echo $LN
Login (julio):
julio

$ read -p "Login ($USER): " LN; LN=${LN:-$USER}; echo $LN
Login (julio): Botelho
Botelho
```

No exemplo acima ofereci a variável de sistema `$USER` como valor *default* (entre parênteses). Na primeira execução dei `<ENTER>` para aceitar este valor e como você pode ver, a Expansão de Parâmetros botou na variável `$LN`, que estava vazia o valor de `$USER`.

Então essa Expansão de Parâmetros faz o seguinte: se a primeira variável (no caso `$LN`) estiver vazia, ela recebe o valor da segunda (`$USER`). Caso contrário permanece com seu valor original.

Para pegar o tamanho de uma variável

```
$ echo ${#Var1}
6
```

Para cortar cadeias

Para cortar `$Var1` da posição 2 (origem zero) com 3 caracteres, faça:

```
$ echo ${Var1:2:3}
dei
$ echo ${Var1:2}       # Da posição 2 até o fim
deia
$ echo ${Var1:2: -1}   # Da posição 2 até uma posição antes do fim
dei
$ echo ${Var1: -4}     # Da 4a posição antes do fim até o fim
deia
$ echo ${Var1: -4: -1} # Da 4a posição antes do fim até uma antes
dei
```

Para substituir cadeias:

```
$ echo ${Var1/d/n}       # Troca 1o. d por n
caneia
$ echo ${Var1//a/u}      # Troca todos a por u
cudeiu
$ echo ${Var1/d*/veira}  # Metacaractere. A partir do d
caveira
```

Para renomear todos arquivos com espaços no nome

```
for Arq in *' '*
do
    mv "$Arq" ${Arq// /_}
done
```

Apagando à esquerda

```
$ Data=15/08/2019
$ echo ${Data#*/}        # Apaga à esquerda tudo até a 1a /
08/2019
$ echo ${Data##*/}       # Apaga à esquerda tudo até a última /
2019
```

Apagando à direita

```
$ echo ${Data%/*}        # Apaga à direita tudo após a última barra
15/08
$ echo ${Data%%/*}       # Apaga à direita tudo após a 1a. barra
15
```

Mostrei para você somente algumas Expansões de Parâmetros, mas existem inúmeras outras que são menos usadas como é o caso da indireção, veja só. Lá no início eu passei 11 parâmetros para o Bash (de a até k). Como faço para listar o último deles sem saber quantos foram passados?

Solução sem Expansão de Parâmetros

```
shift $(($# - 1))
echo $1
```

Como `$#` tem a quantidade de parâmetros passados eu subtraí 1 fazendo um `shift 10` que joga fora os 10 primeiros. Depois foi só listar o único que sobrou.

Solução com Expansão de Parâmetros

```
echo ${!#}
```

que lista o valor da variável apontada por `$#`, isto é `$11`.

Copie essas linhas para um *script* e experimente passar uma quantidade variada de parâmetros para ele, inclusive colocando espaços dentro de um ou mais parâmetros.

Como você pode ver, as Expansões de Parâmetros podem substituir o `cut`, o `sed`, o `expr`, o `tr`, o `test` e outros comandos externos, que são pesados, lentos e complexos. Espero que este artigo lhe seja útil e que você passe a usar estes *builtins*, para acabarmos com esta lenda que o Shell é lento.

11. Diferenças entre o novo e o velho "test"

`[` (O comando `test`) e `[[` (normalmente chamado de comando novo `test`) são usados para avaliar condições. `[[` funciona somente no Bash, Korn Shell e zsh; `[` e `test` estão implementados em qualquer Shell compatível com o padrão POSIX.

Apesar de todos os Shells modernos terem implementações internas (*builtin*) de `[`, geralmente este ainda é um executável externo, sendo normalmente `/bin/[` ou `/usr/bin/[`. Veja isso:

```
$ which [
/usr/bin/[
$ which [[
$ whereis -b [
[: /usr/bin/[
$ whereis -b [[
[[:
```

A opção `-b` do comando `whereis`, serve para que ele mostre somente onde está o binário (código propriamente dito) do arquivo e, pelo resultado das linhas acima, podemos ver que não existe código externo do `[[`, isto é, ele é um intrínseco (*builtin*) e por isso podemos concluir que é mais veloz.

Embora `[` e `[[` tenham muito em comum, e compartilhem muitos operadores como `-f`, `-s`, `-n`, `-z`, há algumas diferenças notáveis, sendo as duas mais importantes é que o novo `test` (`[[`) permite:

1. Comparações usando os metacaracteres de expansão de arquivos

2. Comparações usando Expressões Regulares.

11.1. Testando com coringas

Esse novo tipo de construção é legal porque permite usar metacaracteres de expansão de arquivos para comparação. Esses coringas atendem às normas de Geração de Nome de Arquivos (*File Name Generation*).

Exemplo

```
$ echo $H
13
```

```
$ [[ $H == [0-9] || $H == 1[0-2] ]] || echo Hora inválida
Hora inválida
```

Nesse exemplo, testamos se o conteúdo da variável `$H` estava compreendido entre zero e nove (`[0-9]`) ou (`||`) se estava entre dez e doze (`1[0-2]`), dando uma mensagem de erro caso não estivesse.

Como você pode imaginar, esse uso de padrões para comparação aumenta muito o poderio do comando `test`.

11.2. Testando com Expressões Regulares

Outro grande trunfo deste comando é que ele suporta expressões regulares para especificar condições usando a seguinte sintaxe:

```
[[ CAD =~ REGEX ]]
```

Onde REGEX é uma expressão regular (que pode inclusive usar metacaracteres das Expressões Regulares avançadas). Assim sendo, poderíamos montar uma rotina para crítica de horários com a seguinte construção:

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

DICA: As subcadeias que casam com expressões entre parênteses são salvas no vetor `BASH_REMATCH`. O elemento de `BASH_REMATCH` com índice 0 é a porção da cadeia que casou com a expressão regular inteira. O elemento de `BASH_REMATCH` com índice `n` é a porção da cadeia que casou com a enésima expressão entre parênteses.

Vamos executar direto no *prompt* o comando anterior para entender melhor:

```
$ Hora=12:34
$ 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
$ echo ${BASH_REMATCH[@]}       # Lista todos elementos do vetor
12:34 12
```

```
$ echo ${BASH_REMATCH[0]}        # Índice zero tem o casamento total
12:34
$ echo ${BASH_REMATCH[1]}        # Índice 1 contem o retrovisor 1. Repare
o grupo
12
```

No primeiro echo que fizemos, o caractere arroba (`@`) representa todos os elementos do vetor, em seguida, vimos que o elemento índice zero [0] está com a hora inteira e o elemento [`1`] está com a subcadeia que casou com `[01][0-9]|2[0-3]`, isto é, a expressão que estava entre parênteses. Em outras palavras o índice `N` (com `N` maior que zero) atua como se fosse o retrovisor `\N`, com uma ressalva importante: só podemos criar até 9 retrovisores, mas nesse vetor, tal limite não existe.

Vamos ver se com outro exemplo pode ficar mais claro.

```
$ if [[ supermercado =~ (mini|(su|hi)per)?mercado ]]
> then
>   echo Todos os elementos - ${BASH_REMATCH[@]}        # O  mesmo  que
echo ${BASH_REMATCH[*]}
>   echo Vetor completo     - ${BASH_REMATCH[0]}        # O mesmo que
echo $BASH_REMATCH
>   echo Elemento indice 1 - ${BASH_REMATCH[1]}
>   echo Elemento indice 2 - ${BASH_REMATCH[2]}
> fi
Todos os elementos  - supermercado super su
Vetor completo      - supermercado
Elemento indice 1   - super
Elemento indice 2   - su
```

Agora que vocês entenderam o uso dessa 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 enrolado você?

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
```

11.3. Uma tabela para referência

Característica	Novo test [[	test antigo [
Comparação de cadeias	>	\>[1]
	<	\<[1]
	= (ou ==)	=
	!=	!=
Comparação de inteiros	-gt	-gt
	-lt	-lt
	-ge	-ge
	-le	-le
	-eq	-eq
	-ne	-ne
Avaliação condicional	&&	-a[2]
	||	-o[2]
Grupamento de expressões	(...)	\(... \)[2]
Padrões de expansão de nomes de arquivos	= (ou ==)	Não disponível
Expressões Regulares	=~	Não disponível

1. Esta é uma extensão do padrão POSIX; alguns Shells podem tê-la e outros não.
2. Os operadores `-a` e `-o`, e o agrupador `( . . . )`, são definidos pelo padrão POSIX, mas apenas para casos estritamente limitados, pois são marcados como obsoletos. O uso desses operadores é desencorajado e é melhor você usar vários comandos [.

Prefira fazer:

```
if [ "$a" = a ] && [ "$b" = b ]; ...
if [ "$a" = a ] || { [ "$b" = b ] && [ "$c" = c ];}; ...
```

no lugar de:

```
if [ "$a" = a -a "$b" = b ]; then ...
if [ "$a" = a ] -o \( [ "$b" = b ] -a [ "$c" = c ] \); ...
```

11.4. Algumas belas dicas!

1. No novo test não será feita expansão de metacaracteres curingas (*globbing* - metacaracteres de expansão de arquivos) nem divisão de palavras (*word spliting*), e, portanto, muitos argumentos não precisam ser colocados entre aspas.

```
$ ls arq*

arq  arq1  arq2  arqr



$ [ -f arq* ] || echo Não existe       # Quando expandir dá erro

bash: [: número excessivo de argumentos

Não existe



$ [[ -f arq* ]] || echo Não existe

Não existe                     # Não expandiu e não achou arq*



$ var="a b"

$ [ -z $var ] || echo Tem dado        # Após a expansão virará 2

bash: [: a: esperado operador binário

Tem dado
```

$ [[-z $var]] || echo Tem dado   # Perfeito!

Tem dado

$ > "Nome Ruim"   # Criei arquivo Nome Ruim

$ Arq="Nome Ruim"

$ [-f $Arq] && echo "$Arq é um arquivo"

bash: [: Arq: esperado operador binário

$ [[-f $Arq]] && echo "$Arq é um arquivo"

Nome Ruim é um arquivo

Como você pode ver, o fato de você não precisar usar aspas nem apóstrofos torna o uso de `[[` mais fácil e menos propenso a erros do que o `[`.

2. Usando `[[`, os parênteses não precisam ser "escapados":

```
$ [[ -f $Arq1 && ( -d $dir1 || -d $dir2) ]]
$ [ -f "$Arq1" -a \( -d "$dir1" -o -d "$dir2" \) ]
```

3. A partir do Bash 4.1, a comparação de cadeias usando maior que (`>`) e menor que (`<`) respeita as definições correntes do comando `locale` quando feita com `[[`. As versões anteriores do Bash não respeitavam o `locale`. O `[` e o `test` também não o respeitam em nenhuma versão (embora as *man pages* digam que sim).

Como regra, aconselho sempre usar `[[` para testar condições que envolvam cadeias e arquivos. Se você quiser comparar números, use uma expressão aritmética `(( ... ))`.

Quando deve ser usado o novo comando test (`[[`) e quando se deve usar o antigo (`[`)? Se a portabilidade POSIX ou o Bourne Shell for uma preocupação, a sintaxe antiga

deve ser usada. Se, por outro lado, o script requer bash, zsh ou korn shell, a nova sintaxe é muito mais flexível.

12. Tipagem no Shell

O Shell está pouco se lixando para tipagem de variáveis. Com isso quero dizer que se você cria uma variável o Shell não está nem aí para se ela é do tipo inteiro, ou uma cadeia ou do tipo ponto flutuante, ou ... Nada disso importa, porém existe um comando intrínseco (*builtin*) que por vezes facilita bastante a vida de um programador, atribuindo não um tipo, mas uma forma de usar a variável, a instrução `declare`.

Sua sintaxe é a seguinte:

```
declare [OPT] [NOME[=VALOR] ...]
```

As opções (`OPT`) podem ser acionadas com `-OPT` e desassociadas com `+OPT`. As principais são as seguintes:

Opção	Descrição
`-a`	`NOME` é um vetor indexado
`-A`	`NOME` é um vetor associativo
`-c`	`NOME` é capitalizado (só a primeira letra em maiúscula)
`-i`	`NOME` é um inteiro
`-l`	Converte conteúdo de `NOME` para minúscula
`-u`	Converte conteúdo de `NOME` para maiúscula
`-r`	`NOME` será uma constante (seu valor não pode ser alterado)
`-x`	`NOME` será exportada para todos os subshells

Vejamos seus principais usos:

12.1. Declarando inteiros

O uso a seguir foi rapidamente abordado na seção referente ao comando `expr`.

Veja só que interessante esse modo de usarmos a aritmética:

```
$ declare -i Num        Num foi declarado como um inteiro (-i)
$ Num=2+3*4
$ echo $Num
14
$ Num=(2+3)*4
$ echo $Num
```

```
20
```

Podemos inclusive já declarar e atribuir simultaneamente:

```
$ declare -i Num=2*(3+4)
$ echo $Num
14
```

12.2. Declarando maiúsculas

Para termos somente maiúsculas em uma variável também podemos declará-la usando a opção `-u` (de *uppercase* = maiúscula). Veja:

```
$ Maiusc="converte para maiúsculas"        # Antes de declarar
$ declare -u Maiusc                        # Só agora declarou
$ echo $Maiusc                             # Continuou minúsculo
converte para maiúsculas
$ Maiusc="converte para maiúsculas"        # Atribuiu após o declare
$ echo $Maiusc                             # Agora funcionou
CONVERTE PARA MAIÚSCULAS
```

12.3. Declarando minúsculas

O inverso disso seria usarmos a opção `-l` (de *lowercase* = minúscula). Veja:

```
$ declare -l Minusc
$ Minusc="CONVERTE PARA MINÚSCULAS"
$ echo $Minusc
converte para minúsculas
```

Mas note, tudo que foi dito sobre essas conversões só é válido após o `declare` ter sido feito.

Repare:

```
$ var="xxxx"                  # Vou atribuir antes de declarar
$ declare -u var
$ echo $var
xxxx                          # Nada mudou, continua minúscula
$ var="xxxx"                  # Atribuindo após declaração
$ echo $var
XXXX                          # Agora funcionou...
```

12.4. Declarando valores constantes

Outro galhão que esse intrínseco (*builtin*) quebra é para criar uma constante, isto é, uma variável que seja somente de leitura.

```
$ declare -r Const=Constante
$ echo $Const
Constante
$ Const=Variável
bash: Const: a variável permite somente leitura
$ unset Const
bash: unset: Const: impossível desconfigurar: variável somente leitura
```

12.5. Capitalizando uma palavra

Usando a opção `-c` do `declare`, podemos capitalizar uma palavra.

```
$ declare -c Nom
$ read -p "Nome: " Nom
Nome: frei
$ echo $Nom
Frei
$ read -p "Nome: " Nom
Nome: FREI
$ echo $Nom
Frei
```

Mas não todas as palavras de uma frase:

```
$ read -p "Nome: " Nom
Nome: frei adão
$ echo $Nom
Frei adão
$ read -p "Nome: " Nom
Nome: FREI ADÂO
$ echo $Nom
Frei adâo
```

13. Tudo o que você sempre quis saber sobre o comando paste

O `paste` é um comando pouco usado por sua sintaxe ser pouco conhecida. Vamos brincar com 2 arquivos criados da seguinte forma:

```
$ seq 10 > inteiros
$ seq 2 2 10 > pares
```

Para ver o conteúdo dos arquivos criados, vamos usar o `paste` na sua forma careta:

```
$ paste inteiros pares
1       2
2       4
3       6
4       8
5       10
6
7
8
9
10
```

Agora vamos transformar a coluna do pares em linha:

```
$ paste -s pares
2       4       6       8       10
```

O separador *default* é `<TAB>`, mas isso pode ser alterado com a opção `-d`. Então para calcular a soma do conteúdo de pares primeiramente faríamos:

```
$ paste -s -d'+' pares # também poderia ser -sd'+'
2+4+6+8+10
```

e depois passaríamos esta linha para a calculadora (`bc`) e então ficaria:

```
$ paste -s -d'+' pares | bc
30
```

Assim sendo, para calcular o fatorial do número contido em `$Num`, basta:

```
$ seq $Num | paste -sd'*' | bc
```

Com o comando `paste` você também pode montar formatações exóticas como esta a seguir:

```
$ ls | paste -s -d'\t\t\n'
arq1 arq2 arq3
arq4 arq5 arq6
```

O que aconteceu foi o seguinte: foi especificado para o comando `paste` que ele transformaria linhas em colunas (pela opção `-s`) e que os seus separadores (é...! Ele aceita mais de um, mas somente um após cada coluna criada pelo comando) seriam uma `<TAB>`, outra `<TAB>` e um `<ENTER>`, gerando desta forma a saída tabulada em 3 colunas. Agora que você já entendeu isto, veja como fazer a mesma coisa, porém de forma mais fácil e menos bizarra e tosca, usando o mesmo comando mas com a seguinte sintaxe:

```
$ ls | paste - - -
arq1 arq2 arq3
arq4 arq5 arq6
```

E isto acontece porque se ao invés de especificarmos os arquivos colocarmos o sinal de menos (`-`), o comando `paste` os substitui pela saída ou entrada padrão conforme o caso. No exemplo anterior os dados foram mandados para a saída padrão (*stdout*), porque o pipe (`|`) estava desviando a saída do `ls` para a entrada padrão (*stdin*) do `paste`, mas veja o exemplo a seguir:

```
$ cat arq1
predisposição
privilegiado
profissional

$ cat arq2
encher
mário
motor

$ cut -c-3 arq1 | paste -d "" - arq2
preencher
primário
promotor
```

Neste caso, o `cut` devolveu as três primeiras letras de cada registro de `arq1`, o `paste` foi montado para não ter separador (`-d""`) e receber a entrada padrão (desviada pelo *pipe*) no traço (`-`) gerando a saída juntamente com `arq2`.

14. xargs - construção e execução de comandos

Existe um comando, cuja função primordial é construir listas de parâmetros e passá-la para a execução de outros programas ou instruções. Este comando é o `xargs` e deve ser usado da seguinte maneira:

```
xargs [comando [argumento inicial]]
```

Caso o comando, que pode ser inclusive um *script* Shell, seja omitido, será usado por *default* o `echo`.

O `xargs` combina o argumento inicial com os argumentos recebidos da entrada padrão, de forma a executar o comando especificado uma ou mais vezes.

Exemplo

Vamos produrar em todos os arquivos abaixo de um determinado diretório uma cadeia de caracteres usando o comando `find` com a opção `-type f` para pesquisar somente os arquivos normais, desprezando diretórios, arquivos especiais, arquivos de ligações, etc, e vamos torná-la mais genérica recebendo o nome do diretório inicial e a cadeia a ser pesquisada como parâmetros. Para isso fazemos:

```
$ cat grepr
#
# Grep recursivo
# Pesquisa a cadeia de caracteres definida em $2 a partir do diretorio $1
#
find $1 -type f -print|xargs grep -l "$2"
```

Na execução deste *script* procuramos, a partir do diretório definido na variável `$1`, todos os arquivos que continham a cadeia definida na variável `$2`.

Exatamente a mesma coisa poderia ser feita se a linha do programa fosse a seguinte:

```
 find $1 -type f -exec grep -l "$2" {} \;
```

Este processo tem duas grandes desvantagens sobre o anterior:

1. A primeira é bastante visível: o tempo de execução deste método é muito superior ao daquele, isso porque o `grep` será feito em cada arquivo que lhe for passado pelo `find`, um-a-um, ao passo que com o `xargs`, será passada toda, ou na pior das hipóteses, a maior parte possível, da lista de arquivos gerada pelo `find`;

2. Dependendo da quantidade de arquivos encontrados que atendem ao `find`, poderemos ganhar aquela famosa e fatídica mensagem de erro "*Too many arguments*" indicando um estouro da pilha de execução do `grep`. Como foi dito no item anterior, se usarmos o `xargs` ele passará para o `grep` a maior quantidade de parâmetros possível, suficiente para não causar este erro, e caso necessário executará o grep o `grep` mais de uma vez.

ATENÇÃO! Aê pessoal do linux que usa o `ls` colorido que nem porta de tinturaria: nos exemplos a seguir que envolvem esta instrução, vocês devem usar a opção `--color=none,` se não existem grandes chances dos resultados não ocorrerem como o esperado.

Vamos agora analisar um exemplo que é mais ou menos o inverso deste que acabamos de ver. Desta vez, vamos fazer um *script* para remover todos os arquivos do diretório corrente, pertencentes a um determinado usuário.

A primeira ideia que surge é, como no caso anterior, usar um comando `find`, da seguinte maneira:

```
$ find . -user cara -exec rm -f {} \;
```

Quase estaria certo, o problema é que desta forma você removeria não só os arquivos do cara no diretório corrente, mas também de todos os outros subdiretórios "pendurados" neste. Vejamos então como fazer:

```
$ ls -l | grep " cara " | cut -c55- | xargs rm
```

Desta forma, o `grep` selecionou os arquivos que continham a cadeia **`cara`** no diretório corrente listado pelo `ls -l`. O comando `cut` pegou somente o nome dos arquivos, passando-os para a remoção pelo `rm` usando o comando `xargs` como ponte.

O `xargs` é também uma excelente ferramenta de criação de *one-liners* (scripts de somente uma linha). Veja este para listar todos os donos de arquivos (inclusive seus links) "pendurados" no diretório `/bin` e seus subdiretórios.

```
$ find /bin -type f -follow | \
xargs ls -al | tr -s ' ' | cut -f3 -d' ' | sort -u
```

Muitas vezes o `/bin` é um link (se não me engano, no Solaris o é) e a opção `-follows` obriga o `find` a seguir o link. O comando `xargs` alimenta o `ls -al` e a seqüência de

comandos seguinte é para pegar somente o 3º campo (dono) e classificá-lo devolvendo somente uma vez cada dono (opção `-u` do comando `sort`).

Você pode usar as opções do `xargs` para construir comandos extremamente poderosos. Para exemplificar isso e começar a entender as principais opções desta instrução, vamos supor que temos que remover todos as arquivos com extensão `.txt` sob o diretório corrente e apresentar os seus nomes na tela. Veja o que podemos fazer:

```
$ find . -type f -name "*.txt" | \
xargs -i bash -c "echo removendo {}; rm {}"
```

A opção `-i` do `xargs` troca pares de chaves (`{}`) pela cadeia que está recebendo pelo pipe (`|`). Então neste caso as chaves (`{}`) serão trocadas pelos nomes dos arquivos que satisfaçam ao comando `find`.

Olha só a brincadeira que vamos fazer com o `xargs`:

```
$ ls | xargs echo > arq.ls
$ cat arq.ls
arq.ls arq1 arq2 arq3
$ cat arq.ls | xargs -n1
arq.ls
arq1
arq2
arq3
```

Quando mandamos a saída do `ls` para o arquivo usando o `xargs`, comprovamos o que foi dito anteriormente, isto é, o `xargs` manda tudo que é possível (o suficiente para não gerar um estouro de pilha) de uma só vez. Em seguida, usamos a opção `-n 1` para listar um por vez. Só para dar certeza veja o exemplo a seguir, quando listaremos dois em cada linha:

```
$ cat arq.ls | xargs -n 2
arq.ls arq1
arq2 arq3
```

Mas a linha acima poderia (e deveria) ser escrita sem o uso de *pipe* (`|`), da seguinte forma:

```
$ xargs -n 2 < arq.ls
```

Outra opção legal do `xargs` é a `-p`, na qual o `xargs` pergunta se você realmente deseja executar o comando. Digamos que em um diretório você tenha arquivo com a extensão `.bug` e `.ok`, os `.bug` estão com problemas que após corrigidos são salvos como `.ok`. Dá uma olhadinha na listagem deste diretório:

```
$ ls dir
arq1.bug
arq1.ok
arq2.bug
arq2.ok
...
arq9.bug
arq9.ok
```

Para comparar os arquivos bons com os defeituosos, fazemos:

```
$ ls | xargs -p -n2 diff -c

diff -c arq1.bug arq1.ok ?...y
....
diff -c arq9.bug arq9.ok ?...y
```

Para finalizar, o `xargs` também tem a opção `-t`, onde vai mostrando as instruções que montou antes de executá-las. Gosto muito desta opção para ajudar a depurar o comando que foi montado.

Então podemos resumir o comando de acordo com a tabela a seguir:

Opção	Ação
`-i`	Substitui o par de chaves (`{}`) pelas cadeias recebidas
`-nNum`	Manda o máximo de parâmetros recebidos, até o máximo de `Num` para o comando a ser executado
`-lNum`	Manda o máximo de linhas recebidas, até o máximo de `Num` para o comando a ser executado
`-p`	Mostra a linha de comando montada e pergunta se deseja executá-la
`-t`	Mostra a linha de comando montada antes de executá-la

Acabou! Foi bom para mim, foi bom para você também? :)

15. Expressões Regulares - Retrovisores

Em uma apresentação, sempre que falo em **Expressões Regulares**, fico atônito com a indiferença da plateia ao assunto. Só posso atribuir essa passividade ao desconhecimento, pois elas são extremamente úteis e as maiores aliadas de um programador, reduzindo drasticamente o tempo de desenvolvimento do código e de sua execução.

Mas não para por aí, elas são sensacionais para definir regras de *proxy* e de *firewall* tornando as consultas bem mais rápidas., além de serem usadas por todas as linguagens de programação, também são usadas em todos os editores de texto.

Esse artigo é para incentivar o mergulho mais profundo daqueles que estão ali na beirinha d'água molhando o pé, isto é, daqueles que já deram uma pesquisada sobre o tema, mas que ainda não o estudaram a fundo para conhecerem a totalidade do seu potencial.

15.1. O básico do básico

As **Expressões Regulares** sempre podem ser usadas quando não conhecemos um valor mas conhecemos as suas possíveis formações.

Exemplo

Descobrir todos os CEPs de um arquivo, usando ou não o hífen (`-`) e o ponto separador de milhares. Pense: como você programaria isso na sua linguagem predileta? Para quem conhece **Expressões Regulares**, basta usá-las num simples `grep`.

```
$ grep -E '[0-9]{2}\.?[0-9]{3}-?[0-9]{3}' ARQUIVO
```

Pronto! Tá resolvido!

Vamos desmembrá-la:

`[0-9]{2}`	Números (`[0-9]`) que ocorrem duas vezes (`{2}`);
`\.?`	Um ponto (`.`) opcional (`?`) a contrabarra (`\`) foi usado porque o ponto também é um metacaractere de **Expressões Regulares**, mas nesse caso, estava sendo usado como literal;
`[0-9]{3}`	Números (`[0-9]`) que ocorrem três vezes (`{3}`);
`-?`	Um traço (`-`) opcional (`?`);
`[0-9]{3}`	Números (`[0-9]`) que ocorrem três vezes (`{3}`).

Os parênteses formam grupos que permitem aplicar outros metacaracteres sobre o seu todo.

Exemplo

`pegadas?`	Casa com `pegada` e com `pegadas`;
`pega(das)?`	Casa com `pega` e `pegadas` porque o opcional (`?`) foi aplicado a todo o grupo.

15.2. E o que são os retrovisores?

Além do que acabamos de ver os parênteses (grupos) também retêm o texto que casou com a **Expressão Regular** do seu interior e portanto podemos usá-lo em outra parte da mesma **Expressão Regular** e é a isso que chamamos de retrovisores (tradução livre de *back reference* ou referência anterior).

Exemplo

`(a)br\1c\1d\1br\1` casa com `abracadabra`, mas:

`(a)(br)\1c\1d\1\2\1` também casa.

Explicação

`(a)`	Grupo que casou com o texto `a`;
`br\1`	Literal `br` seguido do 1º retrovisor (`\1`);
`c\1`	Literal `c` seguido do 1º retrovisor (`\1`);
`d\1`	Literal `d` seguido do 1º retrovisor (`\1`);
`br\1`	Literal `br` seguido do 1º retrovisor (`\1`);

Como o literal `br` se repetia duas vezes, no 2º exemplo também criamos um grupo com ele e por ser o 2º grupo criado, foi batizado como `\2`.

Exemplo

```
$ sed -r 's/(.)/\1 /g' <<< abcdefgjij
a b c d e f g j i j
```

Observação: o comando acima é a forma mais rápida e mais limpa que:

```
$ echo abcdefgjij | sed -r 's/(.)/\1 /g'
```

O ponto é um coringa que casa com qualquer caractere e a flag `g` no final diz que a substituição é geral, assim sendo o ponto (`.`) casou com cada uma das letras que foi substituída pelo texto (a letra) casado (`\1`) seguida de um espaço em branco.

Olha esse mesmo exemplo (mal) escrito de outra forma:

```
$ sed -r 's/(.)(.)(.)(.)(.)(.)(.)(.)(.)(.)/\1 \2 ... \10/' <<< abcdefghij
a b ... a0
```

Nesse caso salvamos cada uma das 10 letras em um retrovisor e vimos que ele entendeu o `\10` como um `\1` seguido de um zero, isso porque só podemos usar até 9 retrovisores, mas lhes garanto que essa quantidade é mais que suficiente para todas as tarefas.

Para transformar datas no formato **DD/MM/AAAA** em **AAAAMMDD**:

```
$ sed -r 's|([0-9]{2})/([0-9]{2})/([0-9]{4})|\3\2\1|' <<< 05/04/1947
19470405
```

Onde usei as barras verticais (`|`) como os separadores do `sed` para não confundir com as barras (`/`) da data. Desmembrando a **Expressão Regular**, vem:

`([0-9]{2})`	Número `[0-9]` de 2 algarismos `{2}` - casa com o dia
`/`	O literal `/` entre o dia e o mês
`([0-9]{2})`	Número `[0-9]` de 2 algarismos `{2}` - casa com o mês
`/`	O literal `/` entre o mês e o ano
`([0-9]{4})`	Número `[0-9]` de 4 algarismos `{4}` - casa com o ano

Só mais um exemplo para podermos usar o `grep` com **Expressões Regulares**.

A disposição do arquivo `/etc/passwd` é a seguinte:

```
UserName:x:Uid:Gid:...
```

Pense na dificuldade que você teria para listar todos os usuário que tivessem o `Uid` idêntico ao `Gid`.

Com **Expressões Regulares**, fazemos isso em uma única linha:

```
$ grep -Eo '[[:alnum:]]+:x:([0-9]+):\1:' /etc/passwd
root:x:0:0:
daemon:x:1:1:
bin:x:2:2:
```

```
...
```

Desmembrando a **Expressão Regular**:

`[[:alnum:]]+:`	Todas as alfanuméricas até encontrar o dois pontos (`:`)
`x:`	O literal `x:`
`([0-9]+):`	Montando um grupo com algarismos (para casar com o `Uid`) e terminando com dois pontos (`:`)
`\1:`	Casará com um texto igual ao que foi salvo no `Uid`

É interessante explicar que sempre usei o dois pontos (`:`) como um delimitador. Se não o fizesse, `123` casaria com `1234`. Veja:

```
$ grep -oE '(123):\1:' <<< 123:1234: &&
    echo Casou ||
    echo Não casou
Não casou
```

Tirando o dois pontos após o retrovisor:

```
$ grep -oE '(123):\1' <<< 123:1234: &&
    echo Casou ||
    echo Não casou
123:123
Casou
```

Usei a opção `-o` (*Only match*) que mostra somente o texto casado, para que você possa ver na segunda forma o casamento capenga que foi realizado.

A sintaxe que usei no início deste exemplo foi a mais simples de explicar, mas também poderia (e deveria) ter feito da seguinte forma:

```
$ grep -Eo '[[:alnum:]]+:x:([0-9]+:)\1' /etc/passwd
```

Embutindo o delimitador dois pontos (`:`) no grupo, já que ele terá de aparecer após os dois números.

15.3. Um vetor interessante

A partir do Bash 4.0 o novo comando `test ([[ ... ]])` passou a aceitar **Expressões Regulares**, mas ele tem uma diferença interessante na forma como salva os retrovisores. Ele usa o vetor (*array*) `BASH_REMATCH`, que no seu índice zero tem o casamento total feito pela **Expressão Regular**, no índice um, o que seria o retrovisor `\1`, e assim sucessivamente.

Exemplo

```
$ cat explica_test
#!/bin/bash
clear
read -p "Infome um endereço IP: " IP
[[ $IP =~ ^([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})$ ]] ||
{
    echo Isso não tem formato de endereço IP
    exit 1
}
echo -e "
Casamento total:\t${BASH_REMATCH[0]}
Primeiro octeto:\t${BASH_REMATCH[1]}
Segundo octeto: \t${BASH_REMATCH[2]}
Terceiro octeto:\t${BASH_REMATCH[3]}
Quarto octeto:  \t${BASH_REMATCH[4]}"
```

Algumas observações sobre esse cara:

1. Para usarmos **Expressões Regulares**, temos de testar usando o operador =~
2. Os grupos montados eram desnecessários, foram criados só para poder demonstrar o uso do vetor
3. Essa **Expressão Regular**, nem de perto está otimizada, mas serve para fins didáticos
4. Ela não pretende ser uma crítica de um endereço IP. Serve unicamente para verificar se um dado lido tem formato de endereço IP. Ela dentro de um grep poderia listar todos os endereços IP de um arquivo (experimente usar esse grep na saída do ifconfig)
5. Se não usássemos os metacaracteres ^ e $, o 1? e o 4? octetos poderiam ter qualquer tamanho, casando 3 algarismos e dando o endereço como correto. Experimente...

Dois testes:

$ explica_test

Informe um endereço IP: 12.34.43.21

Casamento total:	12.34.43.21

Primeiro octeto: 12

Segundo octeto: 34

Terceiro octeto: 43

Quarto octeto: 21

$ explica_test

Informe um endereço IP: 12.34.43.2123

A forma otimizada de fazer esta **Expressão Regular** seria:

```
[[ $IP =~ ^([0-9]{1,3}\.){3}([0-9]{1,3})$ ]]
```

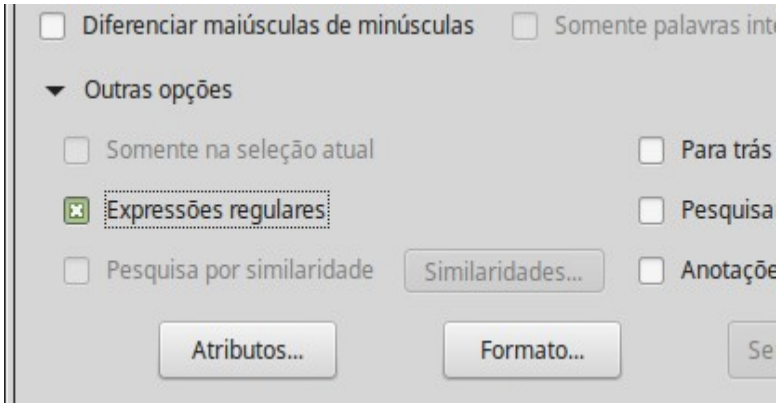
Onde:

`([0-9]{1,3}\.){3}` diz que o grupo formado por de `1` a `3` (`{1,3}`) algarismos (`[0-9]`) mais o ponto (`\.`) ocorre 3 vezes (`{3}`)

Mas nesse caso a demonstração estaria prejudicada, já que no primeiro retrovisor ficaria o último octeto casado por ele, ou seja o terceiro.

Apesar de ter abordado superficialmente as **Expressões Regulares**, já deu para dar uma boa introdução ao assunto.

Estou escrevendo esse artigo no LibreOffice e quando faço um `<CTRL>+H` nele e clico em **Outras opções**, veja o que aparece:



Se você é um cara que gosta de escrever código ou se vive às voltas com regras de firewall, aproveite o tempo que você tem disponível para estudar e, o mais importante praticar **Expressões Regulares**, pois aprender é muito fácil, mas para usá-las é necessário muita prática.

Existem na internet alguns emuladores de **Expressões Regulares**, neles a medida que você vai montando a expressão, ele vai mostrando paulatinamente o casamento. Não acho esse um bom meio de aprender, porque o software pensa por você e por isso não te motiva a aprender. Aconselho praticar **Expressões Regulares** usando sites (existem diversos) que você treina fazendo palavras cruzadas (*crossword*) cujas dicas são dadas por **Expressões Regulares**. Você aprende e se diverte.

Para finalizar vou te dar um choque de realidade: eu não contrato programadores que não conhecem **Expressões Regulares**, porque considero-os profissionais pouco produtivos, ou seja, quem não conhece **Expressões Regulares** é um programador meia boca ;)

16. Duas novas facilidades no comando case

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.

Exemplo

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 20=1, 21=2, 22=4, 23=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.

```
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
```

17. Usos interessantes de algumas variáveis do sistema

O Shell utiliza inúmeras variáveis que já são predefinidas. Para você ter uma ideia, faça:

```
$ env | more
```

e aparecerão variáveis aos borbotões e note, todas com os nomes em caixa alta. **Por isso recomendo a todos não usarem variáveis dos seus aplicativos em letras maiúsculas**. Dificulta a vida de quem está depurando e pode te levar a um erro dificílimo de identificar, devido a você ter usado inadvertidamente um nome de variável do sistema.

A seguir mostro algumas, que têm duas particularidades:

1. Poucos conhecem;

2. São muito úteis quando bem utilizadas.

Vamulá:

Variável	Descrição
`IFS`	Entender o funcionamento desta variável é muito importante, mas como é um tema extenso, este artigo ficaria muito longo no caso de entrar em muitos detalhes, Mas a grosso modo podemos dizer que diversos comandos (entre eles o `read`, `set`, ...) aceitam seus argumentos separados pelos caracteres definidos nesta variável, cujos *defaults* são espaço, `<TAB>` e `<ENTER>`.
`CDPATH`	Contém os caminhos que serão pesquisados para tentar localizar um diretório especificado. Apesar desta variável ser pouco conhecida, seu uso deve ser incentivado por poupar muito trabalho, principalmente em instalações com estrutura de diretórios com muitos níveis.
`PIPESTATUS`	É uma variável do tipo vetor (*array*) que contém uma lista de valores de código de retorno do último *pipeline* executado, isto é, um vetor que abriga cada um dos `$?` de cada instrução do último *pipeline*.
`PROMPT_COMMAND`	Se esta variável receber uma instrução, toda vez que você der um `<ENTER>` direto no *prompt* principal (`$PS1`), este comando será executado. É útil quando se está repetindo muito uma determinada instrução.
`REPLY`	Quando fazemos uma leitura com o comando `read` e não associamos nenhuma variável para receber a entrada, esses dados serão guardados nessa variável. Normalmente a usamos para leituras sem interesse futuro.

Veja isso:

```
$ read a b c <<< "1 2 3"
$ echo $a - $b - $c
1 - 2 - 3
```

Como a entrada veio separada por espaços em branco, que faz parte dos valores default da `$IFS`, pudemos ler seu conteúdo para diversas variáveis em um só `read`. Vou fazer o mesmo com `<ENTER>` que também é *default*.

```
$ Var=$(echo -e "1\n2\n3")
$ echo "$Var"
1
2
3
$ read a b c <<< $Var
$ echo $a - $b - $c
1 - 2 - 3
```

Agora vamos separar os campos por porralouquice:

```
$ IFS=-^ read a b c <<< 1^2-3
$ IFS=-:^ read a b c <<< 1^2-3
$ echo $a - $b - $c
1 - 2 - 3
```

Umas observações sobre o uso do `IFS`:

1. Nesse último exemplo, repare que entre a atribuição do `IFS` e o comando `read` não existe nada, sequer ponto e vírgula. Isso significa que ele foi alterado somente durante a execução do `read`. Só para te mostrar que isso pode ocorrer com outras duplas de comando+variável, veja a dupla `date+LANG`:

```
$ echo $LANG       # Idioma inicial
pt_BR.UTF-8
$ date
Dom Ago 18 14:56:52 -03 2019
$ LANG=C date      # Alterei o idioma só para o date
Sun Aug 18 14:57:29 -03 2019
$ echo $LANG       # Idioma final é o mesmo
pt_BR.UTF-8
```

2. Sempre que um Shell puder "enxergar" um `IFS`, ele o transformará num espaço em branco. Veja:

```
$ Var=$(grep root /etc/passwd)
```

```
$ echo $Var
root:x:0:0:root:bla-bla-bla:/root:/bin/bash
$ IFS=:
$ echo $Var
root x 0  root bla-bla-bla /root /bin/bash
$ echo "$Var"
root:x:0:0:root:bla-bla-bla:/root:/bin/bash
```

Como você pode notar, quando eu não protegi `$Var` da interpretação do *Shell*, os dois pontos (que era o `IFS` da vez) sumiram.

Quando coloquei `$Var` entre aspas para proteger seu conteúdo da interpretação do *Shell*, os dois pontos voltaram a aparecer.

3. Sempre que você alterar um `IFS`, é recomendável que você o restitua ao seu valor *default* após usá-lo para evitar erros difíceis de serem localizados.

17.2. CDPATH

Assim como a variável `$PATH` contém os caminhos nos quais o Shell procurará os arquivos, a `$CDPATH` possui os caminhos a serem seguidos para se fazer um `cd`.

```
$ cd bin
bash: cd: bin: Arquivo ou diretório não encontrado
$ CDPATH=$CDPATH:/usr/local     # Adiciona /usr/local nos caminhos
$ echo $CDPATH
.:..:~:/usr/local
$ pwd
/home/jneves/LM
$ cd bin
$ pwd
/usr/local/bin
```

Como `/usr/local` estava na minha variável `$CDPATH`, e não existia o diretório bin em nenhum dos seus antecessores (`.`, `..` e `~`), o `cd` foi executado para `/usr/local/bin`.

17.3. PIPESTATUS

Um vetor (*array*) em que cada elemento possui o código de retorno (`$?`) de cada uma uma das instruções que compõem um *pipeline*.

```
$ who
jneves   pts/0        Apr 11 16:26 (10.2.4.144)
jneves   pts/1        Apr 12 12:04 (10.2.4.144)
```

```
$ who | grep ^botelho
$ echo ${PIPESTATUS[*]}
0 1
```

Neste exemplo mostramos que o usuário `botelho` não estava "logado", em seguida executamos um *pipeline* que procurava por ele. Usa-se a notação `[*]` (ou `[@]`) em um vetor para listar todos os seus elementos, e desta forma vimos que a primeira instrução (`who`) foi bem sucedida (código de retorno 0) e a seguinte (`grep`), não foi (código de retorno 1).

17.4. PROMPT_COMMAND

Nunca dei muita bola para essa variável porque nunca havia encontrado nenhuma aplicabilidade para ela. Por outro lado, diversas vezes já dei um `<CTRL>+r` para pesquisar no arquivo de histórico de comandos uma instrução muito longa ou alguma que já havia esquecido e não encontrei o que queria porque este arquivo é circular, isto é, depois de um determinado número de registros gravados (essa quantidade é definida pela variável `$HISTSIZE`) começa a haver superposição.

Para ter os meus preciosos comandos por mais tempo, comecei tentando aumentar o valor de `$HISTSIZE` até o limite, mas, mesmo assim, após meses, pesquisava o histórico e a linha de comando que me interessava não estava mais lá.

Foi então que li um artigo que me chamou a atenção sobre a variável `PROMPT_COMMAND` e veja só o que inseri no `~/.bashrc`:

```
PROMPT_COMMAND='echo "$(history 1)" >> history4ever'
```

Como o comando `history 1` lista a última linha executada, e como a função de `$PROMPT_COMMAND` é executar o seu conteúdo antes de devolver o *prompt*, todo comando executado é gravado no arquivo `history4ever` e, quando preciso de um comando, basta pesquisá-lo usando todas as facilidades que o `grep` provê.

Ainda sobre o `history`, sempre que estou dando um treinamento, logo no primeiro minuto de aula aconselho que os treinandos ponham no seu `~/.bashrc` a seguinte linha:

```
$ PROMPT_COMMAND="echo -n \[$(date +%d/%m-%H:%M)\] "
```

Supondo que o $PS1 na minha máquina seja um cifrão ($), a cada vez que eu der um <ENTER>, ele me devolverá um *prompt* com a data e a hora, seguido pelo cifrão ($). Assim:

```
$ PROMPT_COMMAND="echo -n \[$(date +%d/%m-%H:%M)\]"
[12/09-02:56]$
```

Faço isso porque ao final do curso, junto com todo o material que entrego, o aluno ainda levará um pendrive com o histórico de tudo que ele fez ao longo das 40 horas de treinamento, com a data e a hora devidamente registradas.

Uma forma fácil de você ver a $PROMPT_COMMAND em ação é fazendo, por exemplo:

```
$ PROMPT_COMMAND=ls
```

Agora a saída de cada comando que você executar será seguida pela lista dos arquivos do diretório. Ficou um saco, né? Para desfazer faça:

```
$ unset PROMPT_COMMAND
```

Eu disse que seria executado antes do *prompt*, mas isso é o mesmo que dizer que será executado após cada comando. Então poderíamos fazer:

```
$ PROMPT_COMMAND='(( $? )) && {
    tput bold
    echo instrução errada
    tput sgr0
    }'
$ ./nãoexiste
bash: ./nãoexiste: Arquivo ou diretório não encontrado
instrução errada
```

Ou seja, após cada instrução, seu código de retorno é testado e, sendo diferente de zero, a mensagem `instrução errada` será dada em ênfase e logo após será restaurado o modo normal do terminal (sem negrito).

17.5. REPLY

Quando você lê algo com importância somente pontual para seu programa, use esta variável para receber o dado lido.

```
read -n1 -p "Deseja sair? "; [[ ${REPLY^} == S ]] && exit
```

Repare que não armazenei em nenhuma variável o que foi teclado, pois após esse local, ele não teria nenhum interesse para o meu programa. Usei então o `$REPLY` dentro de uma Expansão de Parâmetros que passa seu conteúdo para caixa alta.

O uso clássico dela é para coisas do tipo:

```
rread -n1 -p "Deseja sair? "
grep -q ${REPLY^} <<< YS &&
    echo Que pena que você vai abandonar o barco…
```

Diga de lá que eu ovo de cá: pra que serve guardar a saída dessa pergunta numa variável específica? Pra nada! A resposta será útil somente para/nesse ponto do programa.

Aqui usamos outros macetes: a *Expansão de Parâmetros* `${REPLY^}` coloca a opção lida (que só pode ter 1 caractere por causa do `read -n1`) em letra maiúscula e o `grep` testa se ela é `Y` ou `S`, dando então a mensagem.

18. Script contador de palavras

Relacionamos a seguir um contador de palavras completo e documentado.

Pode ter 3 tipos de saída:

1. Saída texto
2. Saída gráfica com zenity
3. Saída gráfica com YAD

Pode receber os dados como parâmetro:

```
$ ContaPalavras.sh arq*
```

Ou via *pipe*:

```
$ echo arq* | ContaPalavras.sh
$ ls arq* | ContaPalavras.sh
```

Em caso de erro, a mensagem respectiva será mandada para a saída de erros, então se você fizer:

```
$ ContaPalavras.sh ArqQueNaoExiste 2> /dev/null
```

ou:

```
$ ContaPalavras.sh ArqQueNaoExiste 2>&-
```

Não haverá saída de erro, mas o código de retorno (`$?`) será `1`.

Código do Script

```
#!/bin/bash
declare -A TodasPalavras
declare -i Minimo=4
trap "rm /tmp/Arquivao 2>&-; exit" 0 2 3 15
function Uso  # Dá msg de erro, ensina como usar e aborta
{
    echo "
$0: Faltou nome(s) válido(s) de arquivo(s)
Os nome(s) do(s) arquivo(s) pode(m) ser passado(s) das seguintes maneiras:
    echo ARQUIVO | $0
    $0 ARQUIVO
    Metacaracteres podem ser usados. P. ex: $0 ARQ*" >&2
    # A msg de erro é enviada para std err (>&2)
    exit 1
}
```

```
### De onde está vindo o dado, pipe ou parâmetro?
if  [[ -t 0 ]]     # Testa se o dado não está em stdin.
                   # Zero é o identificador da entrada primária (|, < ou <<<)
then
    (($#)) || {    # Testa se tem parâmetro
        echo Passe os dados via pipe, arquivo ou passagem de parâmetros >&2
        exit 1
    }
    Parms="$@"      # Coloca em Parms os parâmetros passados
else
    Parms=$(cat -)  # Coloca em $Parms o conteúdo de stdin
fi
set $Parms   # O conteúdo de $Parms se torna os parâmetros do prg
ls $Parms &>- || Uso # Não passou parâmetro ou arq. não existe.
                     # Obs: &>- fecha as saídas primaria e de erro
Parms=$(file $Parms | grep text | cut -f1 -d:) # retira da lista arquivos
                                               # que não são de texto puro

### Início da rotina que conta palavras ###
cat $Parms > /tmp/Arquivao   # Junta o texto de todos os arquivos num só arquivão
# A linha a seguir pega cada palavra do arquivo passado após, apagar pontuação e
números e pôr tudo em caixa baixa
for Palavra in $(tr [:punct:][:digit:] ' ' < /tmp/Arquivao | tr A-Z a-z)
do
    ((${#Palavra} < Minimo)) && continue  # Despreza palavras menores
                                          # que o minimo definido
    let TodasPalavras[$Palavra]++
done

### Pulo do gato! Concatena o vetor que contém
### as contagens com seus índices (palavras),
### ordena numéricamente e divide em colunas para facilitar consulta
paste <(tr ' ' '\n' <<< ${TodasPalavras[@]}) <(tr ' ' '\n' <<< ${!
TodasPalavras[@]}) | # Concatena as qtdes com as palavras
    sort -nr | # Classifica numéricamente em ordem reversa (maiores primeiro)
    column -c80 -x
### Fim da rotina que conta palavras ###

### Se quiser uma saída gráfica, comente as 3 linhas acima
### e descomente as 3 linhas a seguir:
#  paste <(tr ' ' '\n' <<< ${TodasPalavras[@]}) <(tr ' ' '\n' <<< ${!
TodasPalavras[@]}) |
#+    sort -nr |
#+    zenity --list --column Col1 --column Col2 --column Col3 --hide-header --
height 700 --width 400

### A saída das linhas a seguir também é gráfica,
### porém preciso do yad - recomendo muito instalar pq é bom demais
#  paste <(tr ' ' '\n' <<< ${TodasPalavras[@]}) <(tr ' ' '\n' <<< ${!
TodasPalavras[@]}) |
#+     sort -nr |
#+     column -c80 -x |
#+     yad --text-info --fontname "monospace 10" --height 700 --width 600
```

19. Uma longa discussão sobre o printf

Essa é uma dica longa porque foi uma *thread* que começou com uma dúvida e durou diversos dias em uma lista de discussão de Shell. O nosso colega de lista queria fazer preenchimento à esquerda com o símbolo `#` para fazer bloqueio de um campo. Para tal, ele fez um `for` cheio de álgebra, para calcular quantas voltas daria o *loop*, já que o tamanho total do campo era 15, mas a quantidade de algarismos de `$Valor` era variável.

A minha proposta foi:

```
$ Valor=123,45
$ Valor=$(printf "%15s\n" $Valor)
$ tr ' ' '#' <<< "$Valor"
#########123,45
```

Ou seja, o `printf` preencheu com espaços completando as 15 posições predeterminadas e o `tr` trocou os espaços pelos caracteres de bloqueio (`#`). Fique atento: se você não proteger com aspas a variável `$Valor`, os espaços em branco inseridos pelo `printf` serão perdidos durante a execução do `tr`.

Mas o `printf` possui a opção `-v`, com a qual podemos indicar o nome de uma variável que receberá a saída do comando. Assim sendo, obteríamos o mesmo resultado fazendo:

```
$ Valor=123,45
$ printf -v Valor "%15s\n" $Valor
$ printf '%s\n' ${Valor// /#}
#########123,45
```

Isso por si só, já aceleraria bastante, pois evitaria o *fork* produzido pela substituição de comandos (`$(..)`), mas também usamos uma *Expansão de Parâmetros*, onde `${Valor// /#}` que atua como um comando `tr` só que muito mais veloz, ela serve para trocar todos os espaços em branco por `#`.

Já que falamos nessa *Expansão de Parâmetros*, olha uma forma legal de se corrigir a máscara de um endereço de hardware (*mac address*) em maiúsculas ou minúsculas, a seu gosto:

```
$ Mac=0:13:ce:7:7a:ad
$ printf -v Mac "%02x:%02x:%02x:%02x:%02x:%02x" 0x${Mac//:/ 0x}
```

```
$ echo $Mac
00:13:ce:07:7a:ad
$ printf -v Mac "%02X:%02X:%02X:%02X:%02X:%02X" 0x${Mac//:/ 0x}
$ echo $Mac
00:13:CE:07:7A:AD
```

Nesse exemplo, repare o que a *Expansão de Parâmetros* fez:

```
$ Mac=0:13:ce:7:7a:ad
$ echo 0x${mac//:/ 0x}
0x0 0x13 0xce 0x7 0x7a 0xad
```

Agora que todos viraram hexadecimais, foi só meter um `printf` para formatar.

Seguindo a discussão da lista de Shell da qual falávamos antes da *Expansão de Parâmetros* interromper o nosso papo, alguém começou a falar em fazer uma linha tracejada com 20 caracteres traço (`-`) e aí apareceram as seguintes propostas:

- O processo caretão, estilo "NPF" (Não Pense, Faça!):

```
$ printf '%s\n' --------------------
```

- Da forma interativa e lenta:

  Basta um *loop* (feito com o `for`) para imprimir um hífen (com `printf`) vinte vezes e no final saltar linha (`echo`).

```
$ for ((x = 0; x < 20; x++)); do
>     printf %s -
> done; echo
```

- Usando o processo que acabamos de ver no exemplo de bloqueio de valores, só que agora não usaremos o `#`, e sim o `-`:

```
$ printf -v Espacos %20s
$ echo "${Espacos// /-}"
--------------------
```

Assim procedendo, a opção `-v` jogou a saída do primeiro `printf` (vinte espaços em branco) na variável `$Espacos` e a *Expansão de Parâmetros* substituiu os espaços em branco por hifens, gerando o resultado desejado.

- Para passar uma linha por todo o terminal (Dica 1)

```
$ printf "%$(tput cols)s\n" ' ' | tr ' ' -
```

Nesse caso o `tput cols` devolveu a quantidade de colunas da janela e o `printf`, juntamente com o `tr`, executou o que desejávamos, como já vimos nos outros exemplos.

- Para passar uma linha por todo o terminal (Dica 2):

  Não é tão boa como a primeira dica, mas serve porque é didática.

```
$ Tracos="----------------------------------------\
----------------------------------------\
----------------------------------------\
----------------------------------------"
$ printf '%s\n' "${Tracos:0:$(tput cols)}"
```

  Primeiramente criei uma linha com muitos hifens (mais do que suficientes para fazer o tracejado que desejamos), em seguida usei uma *Expansão de Parâmetros* que significa "extraia da variável `$Tracos` uma subcadeia que começa na posição `0` (início) e tem `tput cols` (quantidade de colunas da sessão corrente) caracteres".

  Poderíamos evitar a *Expansão de Parâmetros* usando o comando `cut`, porém ficaria mais lenta e mais complicada:

```
$ printf '%s\n' $(cut -c 1-$(tput cols) <<< $Tracos)
```

  Nesse caso o `cut` cortou `Tracos` desde o 10 caractere (`-c 1`) até (`-`) o caractere na posição do tamanho da tela (`$(tput cols)`).

  De repente um colega mandou a seguinte dica para a nossa lista:

  "Só compartilhando uma funçãozinha que fiz aqui para desenhar caixas de mensagem (só funciona para mensagens com uma linha. Se alguém quiser alterar, fique à vontade)"

  E nos brindou com esse código:

  function DrawBox

```
{
   string="$*";
   tamanho=${#string}
   tput setaf 4; printf "\e(0\x6c\e(B"
   for i in $(seq $tamanho)
      do printf "\e(0\x71\e(B"
   done
   printf "\e(0\x6b\e(B\n"; tput sgr0;
   tput setaf 4; printf "\e(0\x78\e(B"
   tput setaf 1; tput bold; echo -n $string; tput sgr0
   tput setaf 4; printf "\e(0\x78\e(B\n"; tput sgr0;
   tput setaf 4; printf "\e(0\x6d\e(B"
   for i in $(seq $tamanho)
      do printf "\e(0\x71\e(B"
   done
   printf "\e(0\x6a\e(B\n"; tput sgr0;
}
```

Seu uso seria da seguinte forma:

$ DrawBox Qualquer frase que caiba no terminal

Só que essa caixa é azul (`tput setaf 4`) e as letras são vermelhas, com ênfase (`tput setaf 1; tput bold`).

Mas observe a tabela a seguir. Ela explica os códigos gerados por esse monte de `printf` estranho:

O `printf`	Produz
`\e(0\x6c\e(B`	+
`\e(0\x71\e(B`	-
`\e(0\x6b\e(B`	+
`\e(0\x78\e(B`	|
`\e(0\x6d\e(B`	+
`\e(0\x6a\e(B`	+

Como eu tinha acabado de escrever os exemplos que vimos antes e ainda estava com eles na cabeça, sugeri que o `for` que ele usou para fazer as linhas horizontais fosse trocado; assim, substituiríamos:

```
for i in $(seq $tamanho)
    do printf "\e(0\x71\e(B"
done
```

Por:

```
printf -v linha "%${tamanho}s" ' '
printf -v traco "\e(0\x71\e(B"
echo -n ${linha// /$traco}
```

Tudo Shell puro, pois o `for`, o `printf` e o `echo` são *built-ins*, mas como o `printf` dentro do `for` seria executado tantas vezes quantos traços horizontais houvessem, imaginei que a minha solução fosse um pouco mais veloz e pedi-lhe para testar os tempos de execução, não sem antes apostar um chope. Ele criou dois *scripts*, um que executava mil vezes a função que ele havia proposto e outro que fazia o mesmo com a minha solução. Não vou dizer qual foi a forma mais veloz, porém adianto que o colega ainda está me devendo um chope... ;)

O código otimizado ficaria assim:

```
function DrawBox
{
    string="$*";
    tamanho=${#string}
    tput setaf 4; printf "\e(0\x6c\e(B"
    printf -v linha "%${tamanho}s" ' '
    printf -v traco "\e(0\x71\e(B"
    echo -n ${linha// /$traco}
    printf "\e(0\x6b\e(B\n"; tput sgr0;
    tput setaf 4; printf "\e(0\x78\e(B"
    tput setaf 1; tput bold; echo -n $string; tput sgr0
    tput setaf 4; printf "\e(0\x78\e(B\n"; tput sgr0;
    tput setaf 4; printf "\e(0\x6d\e(B"
    printf -v linha "%${tamanho}s" ' '
    printf -v traco "\e(0\x71\e(B"
    echo -n ${linha// /$traco}
    printf "\e(0\x6a\e(B\n"; tput sgr0;
}
```

Então agora, voltando aos nossos exemplos de passar uma linha por todo o terminal, posso sugerir uma outra (e mais bonita) solução:

- Para passar uma linha por todo o terminal (Dica 3)

```
printf -v linha "%$(tput cols)s" ' '
printf -v traco "\e(0\x71\e(B"
echo ${linha// /$traco}
```

20. Substituição de Processos

Eu quero listar todos os arquivos com os nomes começados por `arq` de um diretório, contando-os, então eu faço:

```
ls arq* | while read Arq
do
    echo $((i)) $Arq
done; echo Eu tenho :$i: arquivos
```

E a saída disso foi a seguinte:

```
1 arq
2 arq.err
3 arq.err1
4 arq.limpo
5 arq.lixo
6 arq.log
7 arq10
Eu tenho :: arquivos
```

Ou seja, funcionou tudo beleza, mas na hora de totalizar deu caca. Você sabe por quê?

Explico: o `while` inteiro rodou num *subshell* provocado pelo *pipe* (`|`) e como você pode ver pela contagem, a variável `$i` foi incrementada, mas ao terminar o *subshell*, todo seu ambiente foi embora com ele e `$i` não escapou.

Como fazer então? Se fosse um arquivo que eu estivesse listando, bastaria redirecionar a entrada do *loop* com um `< ARQUIVO` logo após o `done`. Mas o `ls` não é um arquivo, é um comando, então o que fazer? É para isso que existe a substituição de processos, então você deveria fazer:

```
while read Arq
do
    echo $((i)) $Arq
done < <(ls); echo Eu tenho :$i: arquivos
```

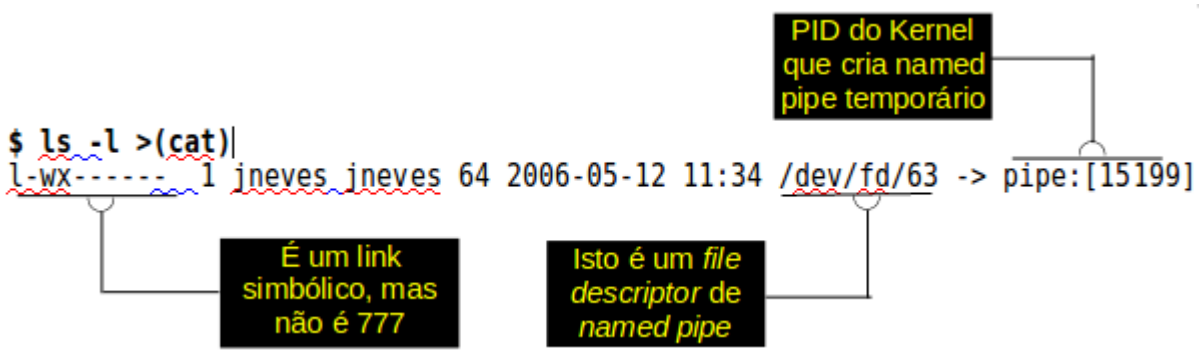
onde o 1º menor (`<`) é o redirecionamento de entrada (*stdin*) e o `<(ls)` é a Substituição de Processos.

Veja isso:

```
cat <(ls arq*)
arq
```

```
arq.err
arq.err1
arq.limpo
arq.lixo
arq.log
arq10
```

Se o `cat` listou é porque o `<(ls arq*)` é um arquivo. Então vamos inverter isso para que ver que arquivo é esse:



Disso tudo que vimos até agora, podemos resumir dizendo que a Substituição de Processos é uma técnica na qual se emula a saída de um comando como vindo de um arquivo temporário do tipo *FIFO* ou *Named Pipe* e serve para uso em instruções que precisam de um arquivo como entrada.

Como um exemplo final, vejamos o comando `comm`, muito pouco usado, porém excelente no que se propõe. Sua sintaxe é:

```
comm [OPCOES]... ARQ1 ARQ2
```

e ele é usado para comparar linha a linha `ARQ1` e `ARQ2`, desde que estejam classificados

Sem nenhuma opção ele produz uma saída com 3 colunas:

- Coluna 1 - Linhas únicas no `ARQ1`
- Coluna 2 - Linhas únicas no `ARQ2`
- Coluna 3 - Linhas comuns aos 2 arquivos.

Mas as suas principais opções são:

`-1`	Suprime a coluna 1
`-2`	Suprime a coluna 2
`-3`	Suprime a coluna 3

Uma vez entendido o comando, voltemos à Substituição de Parâmetros: o problema era excluir todos os arquivos que não tivessem uma determinada palavra. Veja o *one-liner* solução:

```
$ rm -i $(comm -13 <(grep -li 'PALAVRA' *) <(ls))
```

Nesta linha o comando `rm -i` removerá interativamente a saída do comando:

```
comm -13 <(grep -li 'PALAVRA' *) <(ls)
```

Que nada mais é que a instrução `comm` comparando 2 arquivos do tipo *Named Pipe*:

`<(grep -li 'PALAVRA' *)`	Que devolve o nome (`-l`) de todos os arquivos que contêm `PALAVRA`
`<(ls)`	e o nome de todos os arquivos.

Com a opção `-13`, são suprimidas as colunas 1 e 3, sobrando unicamente a coluna 2 que é referente às linhas únicas do arquivo 2, isto é, aqueles nos quais não foi encontrado `PALAVRA`.

Espero que este artigo tenha sido útil para entender o funcionamento da Substituição de Processos e, de quebra, os macetes do utilíssimo comando `comm`.

21. Opções úteis do cut mas não muito usadas

Mais três opções do `cut` que, apesar de pouco usadas, são muito úteis porque em casos específicos evitam um grande trabalho. São elas:

Diretiva	Descrição
`-s`	Não lista as linhas que não possuem o delimitador especificado.
`--complement`	Lista tudo menos o campo definido.
`--output-delimiter`	Especifica o delimitador que virá na saída do comando.

Veja os exemplos:

```
$ ls arq*
arq1 arq2.txt arq3.sh
```

Para pegar somente as extensões (mas repare que `arq1` não tem extensão), devemos fazer:

```
$ ls arq* |cut -f2 -d.
arq1
txt
sh
```

EPA! `arq1` não tinha que estar aí! O que houve? O problema foi ocasionado pelo padrão (*default*) do `cut`. Caso o alvo não tenha o separador definido, ele mandará tudo para a saída. Para evitar que isso aconteça, ou seja, para termos como resposta somente os campos que especificamos, existe a opção `-s`. Veja-a em uso:

```
$ ls arq* |cut -sf2 -d.
txt
sh
```

Veja agora o uso das opções `--complement` e `--output-delimiter`:

```
$ seq -s: 10
1:2:3:4:5:6:7:8:9:10
$ seq -s: 10 | cut -f5 -d: --complement
1:2:3:4:6:7:8:9:10          # Não listou o quinto campo
$ seq -s: 10 | cut -f5 -d: --complement --output-delimiter \|
1|2|3|4|6|7|8|9|10          # Substituiu, na saída, o delimitador : por |
$ seq -s: 10 | cut -f5 -d: --complement --output-delimiter $'\n'
1
2
3
4
```

```
6
7
8
9
10
```