SHELL SCRIPTING

Capítulo 1

Recordar é Viver...

1.1 - Usando Aspas, Apóstrofos e Barra Invertida

Para usar literalmente um caracter especial sem que o Shell interprete seu significado, coloque o caracter entre aspas ou entre apóstrofos ou coloque uma barra invertida antes dele. Seus usos característicos são melhores definidos assim:

- Aspas Duplas - Quando se coloca um caracter especial entre aspas, o Shell ignora o seu significado, exceto no caso deste caracter ser um cifrão (**$**), uma crase (**`**), ou umaa barra invertida(****);
- Apóstrofos - Os apóstrofos são mais restritivos. Todos os caracteres entre apóstrofos são ignorados;
- Barra Invertida - O Shell ignora um e somente um caracter que segue a barra invertida. Quando colocada ao final da linha, o Shell interpreta a barra invertida como um aviso de continuação da linha, devolvendo um prompt secundário (`PS2`) na linha seguinte da tela.

Exemplos:

```
$ echo *
 Arq1 Arq2 ....
$ echo \\
\
$ echo \
>  <^c>
$ echo "\"
> <^c>
$ echo Estou escrevendo uma linha compacta.
Estou escrevendo uma linha compacta
$ echo Assim     não    se         escreve    uma    linha    espacejada.
Assim não se escreve uma linha espacejada.
$ echo  "Estou     escrevendo   uma       linha     espacejada."
Estou     escrevendo   uma       linha     espacejada.
```

1.2 - Crase e Parênteses Resolvendo Crise entre Parentes

As crases são usadas para avisarmos ao Shell que o que está entre elas é um comando e para darmos prioridade em sua execução. Às vezes, é necessário priorizarmos um comando para que o seu resultado seja utilizado por outro.

Exemplos:

Supondo que o nome do computador que estamos trabalhando seja *comput1*, e fazendo:

```
$ echo "O nome deste computador é `uname`"
O nome deste computador é comput1
```

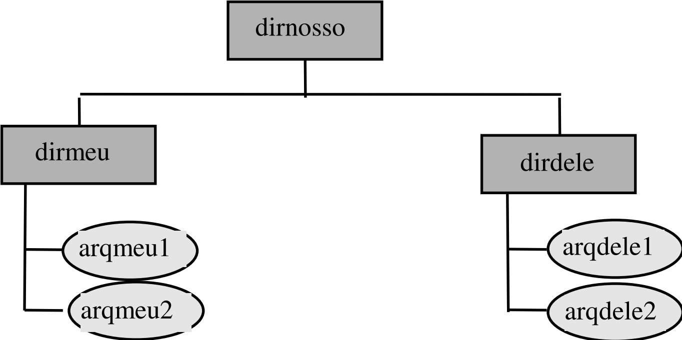
Caso o comando `uname` não estivesse entre crases o resultado teria sido:

```
O nome deste computador é uname.
```

Talvez por causa da matemática, temos tendência a usar os parênteses para priorizar esta execução de comandos. Está errado! No Shell, quando se usa um comando ou um agrupamento de comandos (que se consegue separando-os por ponto-e-vírgula), o que se esta fazendo é chamar um Shell secundário para executar este(s) comando(s).

Exemplos:

Suponhamos que você está situado em *dirmeu* na seguinte estrutura de diretórios:



Preste atenção nos comandos abaixo:

```
$ ( cd ../dirdele; ls )                    O ponto-e-vírgula separa 2 comandos na linha
arqdele1
arqdele2
$ pwd
dirmeu
```

Repare que nós fomos ao diretório *dirdele* e listamos os arquivos lá contidos, porém ao final do comando estávamos novamente no diretório *dirmeu*. Se, após esta seqüência de comandos, você sentir-se o David Coperfield, desista. No duro seu Shell nunca saiu do diretório *dirmeu,* o que aconteceu foi que os parênteses invocaram um novo Shell que, este sim, foi para o 2º diretório, listou seu conteúdo e morreu...

1.3 - Direcionando os Caracteres de Redirecionamento

A maioria dos comandos tem uma entrada, uma saída e pode gerar erros. Esta entrada é chamada *Entrada Padrão* e seu *default* é o teclado do terminal. Analogamente, a saída do comando é chamada *Saída Padrão* e seu *default* é a tela do terminal. Para a tela também são enviadas por *default* as mensagens de erro oriundas do comando que neste caso é chamada *Saída de Erro Padrão*.

Para que a execução de comandos não obedeçam aos seus respectivos *default*s, podemos usar os caracteres de redirecionamento como nas tabelas:

	Redirecionamento de Saída
>	Redireciona a saída de um comando para um arquivo especificado, inicializando-o caso não exista ou destruindo seu conteúdo anterior.
>>	Redireciona a saída de um comando para um arquivo especificado, anexando-o ao seu fim. Caso este arquivo não exista, será criado.
2>	Redireciona os erros gerados por um comando para o arquivo especificado. Caso não ocorra erro na execução do comando, o arquivo não será criado.

	Redirecionamento de Entrada
<	Avisa a um comando que a entrada padrão não será o teclado, mas sim o arquivo especificado.
<<	Também chamado de *here document*. Serve para indicar ao Shell que o escopo de um comando começa na linha seguinte e termina quando encontra na coluna 1 o label que segue o sinal <<.

	Redirecionamentos Especiais
|	Este é o famoso *pipe*, serve para direcionar a saída de um comando para a entrada de outro. É utilíssimo; não tenha parcimônia em usá-lo, pois, normalmente otimiza a execução do comando.
tee	Captura a saída de um comando com *pipe,* copiando o que está entrando no *tee* para a *saída padrão* e outro comando ou arquivo.

Exemplos:

```
$ ftp -ivn remocomp << FimFTP >> /tmp/$$ 2>> /tmp/$$
>    user fulano segredo
>    binary
>    get arqnada
>FimFTP
$
```

*Os sinais **>** são prompts secundários do UNIX. Enquanto não surgir o label FimFTP o **>** será o prompt (PS2), para indicar que o comando não terminou.*

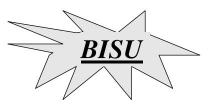
No exemplo acima fazemos um FTP (*File Transfer Protocol* - que serve, basicamente, para transmitir ou receber arquivos entre computadores remotos) para *remocomp*. Vamos analisar estas linhas de código:

Linha 1 - O trecho `<< FimFTP` avisa ao Shell que até que o *label* `FimFTP` seja encontrado na coluna 1 de alguma linha, todas as linhas intermediárias pertencem ao comando `ftp` e não deve ser interpretado pelo Shell.

O trecho `>> /tmp/$$` significa que as mensagens do `ftp` deverão ser anexadas ao arquivo `/tmp/<Num. do Processo>`[1] e `2>> /tmp/$$` o mesmo deverá ser feito com as mensagens de erro provenientes do comando.

Linhas 2 a 4 - Estas 3 linhas são o escopo do comando `ftp`. Nelas informamos o *LoginName* e a *Password* do usuário, em seguida avisamos que a transmissão será binária (sem interpretação do conteúdo) e ordenamos que nos seja transmitido o arquivo arqnada.

Linha 5 - Finalmente o Shell encontrou o término do programa. A partir dai, novamente começará a interpretação.



A variável **$$** deve sempre ser usada para compor nomes de arquivos temporários gerados por *scripts* de uso público, evitando conflitos de permissões de diferentes usuários sobre o referido arquivo.

```
$ mail fulano < blablabla
```

No caso acima, estou mandando um *e-mail* para o usuário `fulano` e o conteúdo deste *e-mail* foi previamente editado no arquivo `blablabla` que esta sendo redirecionado como entrada do comando `mail`.

```
$ rm talvez 2> /dev/null
```

Para não poluir a tela do meu *script* e por não ter certeza se o arquivo *talvez* existe, eu uso esta construção porque caso o arquivo não exista, a mensagem de erro do `rm talvez: no such file or directory` não será exibida na tela, mas sim enviada para `/dev/null`, ou seja, para o buraco negro, onde tudo se perde e nada volta.

```
$ echo "Atualmente existem `who | wc -l` usuarios conectados"
Atualmente existem         4 usuarios conectados
```

`who` - Lista os usuários conectados;
`wc -l` - Conta linhas;
`|` - Pega a lista gerada pelo `who` e a entrega ao `wc -l` para contá-las.

O comando acima poderia ficar melhor se tirássemos as aspas:

[1] A variável $$ representa o PID (**P**rocess **Id**entification).

```
$ echo Atualmente existem `who | wc -l` usuarios conectados
Atualmente existem 4 usuarios conectados
```

Exercícios

Vejamos o conteúdo do nosso diretório corrente:

```
?ls
2ehbom      bdb         listdir     param3      quequeisso  teles
DEADJOE     bronze      listdir1    param4      rem         testchar
DuLoren     confusao    lt          param5      tafechado   testsex
Param       erreeme     medieval    param6      talogado    tputcup
aa          hora        param1      pedi        tavazio     tr
add         kadeo       param2      pp          telefones
```

Como se comportariam os seguintes comandos:

```
ls *
ls [Pp]*
ls [c-ms-z]*o
ls param?
ls ?aram?
ls [!lpt]*o
ls [apt][ae][dls]*
ls *[!4-6]
ls *[aeiou]*
ls te[!s]*
```

O que aconteceria na execução destas seqüencias de comandos:

```
ls | wc -l
ls > /tmp/$$ 2> /tmp/x$$
ls NuncaVi >> /tmp/$$ 2>> /tmp/x$$
echo Nome do Sistema: uname
echo Nome do Sistema: `uname`
mail procara < mala
(cd ; pwd)
cat quequeisso | tee qqisso
mail procara << !
cat /etc/passwd | sort | lp
```

Capítulo 2

Comandos Que Não São do Planeta

Neste capítulo estudaremos comandos e/ou opções de comandos que não são usados com muita freqüência nas tarefas do dia-a-dia da operação do UNIX, porém são utilíssimos na elaboração e desenvolvimento de *scripts*.

Vale a pena como introdução a este capítulo, dar uma relembrada no *ed,* já que algumas das instruções que veremos a seguir, usam abundantemente (ops!) os conceitos, parâmetros e comandos deste editor.

Para tal, temos um arquivo chamado `quequeisso` com o seguinte formato:

```
$ cat quequeisso

ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!
O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:
Interpretador de comandos;
Controle do ambiente UNIX;
Redirecionamento de entrada e saída;
Substituicao de nomes de arquivos;
Concatenacao de pipe;
Execucao de programas;
Poderosa linguagem de programacao.
```

- Indicador de pesquisa: **/**

A barra serve para indicar uma pesquisa no arquivo, assim:

```
/vasco
```

Procurará a seqüência de caracteres `vasco` em qualquer lugar da linha

Exemplos:

```
$ ed quequeisso
416                                          416 caracteres no arquivo
1,$p                                         Listar(p) da 1ª(1) a última($) linha
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!
O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:
Interpretador de comandos;
Controle do ambiente UNIX;
Redirecionamento de entrada e saída;
Substituicao de nomes de arquivos;
```

```
Concatenacao de pipe;
Execucao de programas;
Poderosa linguagem de programacao.
/EH                                          Pesquisa a ocorrência da cadeia EH
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
/                                            Repete a mesma pesquisa
EH UMA LAVAGEM CEREBRAL!!!
```

- Início da linha: **^**

  Quando nossa intenção for pesquisar uma cadeia de caracteres no início da linha e somente no início, usamos o circunflexo **^**.

Assim a expressão regular:

```
^flamengo
```

Procurará a existência dos caracteres `flamengo` somente se eles ocorrerem no início da linha.

Exemplos:

```
$ ed quequeisso
416
1,$p
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!
O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:
Interpretador de comandos;
Controle do ambiente UNIX;
Redirecionamento de entrada e saída;
Substituicao de nomes de arquivos;
Concatenacao de pipe;
Execucao de programas;
Poderosa linguagem de programacao.
/^C                                          Procura C no início da linha
Controle do ambiente UNIX;
/                                            Procura próximo C no início da linha
Concatenacao de pipe;
```

- Fim da linha: **$**

  Da mesma forma que usamos o **^** para pesquisar no início da linha, usamos um cifrão (que alguns preferem chamar dólar. Eu prefiro usar cifrão e ter o dólar no bolso) para pesquisar no final da linha.

Assim:

```
fluminense$
```

procurará os caracteres `fluminense` somente se eles forem os últimos (coisa que não é difícil) da linha.

Exemplos:

Antes do exemplo a seguir, vou passar o conceito de outro caracter, o ponto (**.**), cuja finalidade é servir como uma espécie de curinga posicional, isto é, na posição em que o ponto (**.**) se encontra, qualquer caracter é válido.

```
ca.eta                                  Aceita caneta, capeta, careta,...
c.u                                     Aceita céu, cru, cpu, c7u...
```

vejamos então:

```
$ ed quequeisso
416
/UNIX.$                                 Termina com UNIX  seguido de  outro caracter
O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,   Depois da cadeia UNIX o outro caracter = ,
/                                       Repete a pesquisa
Controle do ambiente UNIX;              Outro caracter  =  ;
```

- Substituição: **s / /**

  Para trocarmos uma cadeia de caracteres por outra, usamos um **s** seguido da cadeia inicial entre duas barras (**/**) e da cadeia final.

Fazendo:

```
s /alhos/bugalhos                       O espaço entre o s e a / é facultativo
```

Estaremos trocando alhos por bugalhos. MAS ATENÇÃO, somente a primeira linha que ocorresse alhos seria alterada. Para alterarmos em todas as linhas deveríamos fazer:

```
1,$ s/alhos/bugalhos
```

Exemplos:

```
1,$ s /de/xx                            Troque de por xx em todas as linhas
Poxxrosa linguagem de programacao.
1,$p
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!
O Shell alem xx analisar cada dado entrado a partir do prompt do UNIX,
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:
Interpretador xx comandos;
Controle do ambiente UNIX;
Redirecionamento xx entrada e saída;
Substituicao xx nomes de arquivos;
Concatenacao xx pipe;
Execucao xx programas;
Poxxrosa linguagem de programacao.
```

Aí, você que é esperto pra chuchu, vai me perguntar:

-Poxa, a linha 8 ficou "`Substituicao xx nomes de arquivos;`". Se o **de** permanece é sinal que esta expressão regular não funciona! Não deveria ter um `xx` no lugar daquele maldito `de`?

E eu vou te responder:

-Não campeão, o ***s*** sozinho substitui somente a primeira ocorrência da primeira linha, se fizermos `1,$ s` será substituída a primeira ocorrência de todas as linhas. Note que o `1,$`

quer dizer da primeira à última linha, em nenhum lugar dissemos que a pesquisa deveria ser global.

Então vamos desfazer esta alteração:

```
u                                                       O u (undo) desfaz a última alteração
1,$p
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!
O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:
Interpretador de comandos;
Controle do ambiente UNIX;
Redirecionamento de entrada e saída;
Substituicao de nomes de arquivos;
Concatenacao de pipe;
Execucao de programas;
Poderosa linguagem de programacao.
```

Vamos finalmente globalizar esta substituição. Para tal, basta acrescentarmos ao final da expressão regular a letra `g` de global. Vale acrescentar que a expressão regular `g` não está atrelada ao comando de substituição. Outros comandos também a usam para cumprir a mesma finalidade.

Então devemos fazer:

```
1,$ s/de/xx/g
1,$p
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!
O Shell alem xx analisar cada dado entrado a partir do prompt do UNIX,
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:
Interpretador xx comandos;
Controle do ambiente UNIX;
Redirecionamento xx entrada e saída;
Substituicao xx nomes xx arquivos;
Concatenacao xx pipe;
Execucao xx programas;
Poxxrosa linguagem xx programacao.
```

Vamos agora restaurar o texto para podermos seguir em frente usando o que aprendemos sobre o ed. Para isto vamos desfazer a última alteração (***u***ndo).

```
u                                                       O u desfaz a última alteração
```

2.1 - O comando sed

Até no nome o comando `sed` se parece com o `ed`. sua sintaxe geral é:

```
sed expressão regular [arquivo]
```

Onde expressão regular nada mais é senão uma das expressões regulares do `ed` como as vistas acima, que obedecem ao seguinte formato geral:

```
[<endereço-1>, [<endereço-2>]] <função> [argumento]
```

Onde <endereço-1>, <endereço-2> definem o escopo de abrangência do comando. Se ambos forem omitidos, a interação será sobre todas as linhas do `arquivo` especificado. Se somente um for eleito, o comando só atuará sobre a linha referida

Se nenhum arquivo for especificado, é assumida a *entrada padrão* (o teclado, lembra-se?).

As funções são várias e semelhantes ao editor `ed`, como veremos a seguir:

- Função substitui: **`s`**

  Como vimos no ed, o **`s`** substitui a cadeia de caracteres que está entre o primeiro par de barras[2] pela cadeia contida no segundo par.

Usando mais uma vez o nosso arquivo `quequeisso`, caso desejássemos destacar as palavras UNIX do texto, poderíamos fazer:

```
$ sed 's/UNIX/UNIX <- ACHEI!!!!!/' quequeisso       Substitui a cadeia UNIX por UNIX <- ACHEI!!!!!
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!
O Shell alem de analisar cada dado entrado a partir do prompt do UNIX <-
ACHEI!!!!!,
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:
Interpretador de comandos;
Controle do ambiente UNIX <- ACHEI!!!!!;
Redirecionamento de entrada e saída;
Substituicao de nomes de arquivos;
Concatenacao de pipe;
Execucao de programas;
Poderosa linguagem de programacao.
```

No entanto, se fizéssemos:

```
$ cat quequeisso
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!
O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:
Interpretador de comandos;
Controle do ambiente UNIX;
Redirecionamento de entrada e saída;
Substituicao de nomes de arquivos;
Concatenacao de pipe;
Execucao de programas;
Poderosa linguagem de programacao.
```

-Ué! eu não acabei de trocar `UNIX` por `UNIX <- ACHEI!!!!!` ? Porque voltou tudo ao que era antes?

Você trocou, mas não especificou a saída da alteração, então o texto alterado foi para a *saída padrão* (tela do terminal), não ficando fisicamente gravado em nenhum lugar. Para que as alterações sejam definitivas, a saída do `sed` deve ser direcionada para um arquivo. CUIDADO! Não use como saída o arquivo de entrada, pois desta forma você estaria perdendo todo o seu conteúdo, e sem chance de arrependimento.

No exemplo abaixo, da linha **1** à **2**, vamos **s**ubstituir as letras maiúsculas(**`[A-Z]`** lê-se de A até Z) por nada (no exemplo não foi colocado nada entre **//**) e finalmente vamos **g**lobalizar esta pesquisa por todas as ocorrências nas duas linhas. A saída será

[2] Par de barras é uma simplificação e seu uso é generalizado mas não obrigatório, pode ser substituído por \?expressão?, onde ? é qualquer caracter. Note que quando fazemos \#123\#456# a barra invertida serve para inibir o efeito de delimitador do segundo #, sendo então a expressão interpretada como 123#456.

redirecionada para um arquivo temporário para, caso queira, salvar as alterações executadas.

```
$ sed '1,2s/[A-Z]//g' quequeisso > /tmp/ex        Da linha 1 a 2 troque maiúscula por nada
$ cat /tmp/ex                                     Lista /tmp/ex que é o arquivo de saída do sed
,      ,                                          Da 1ª linha sobraram uma , 6 espaços e outra ,
   !!!                                            Da 2ª linha sobraram 3 espaços e 3 !
O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:
Interpretador de comandos;
Controle do ambiente UNIX;
Redirecionamento de entrada e saida;
Substituicao de nomes de arquivos;
Concatenacao de pipe;
Execucao de programas;
Poderosa linguagem de programacao.
```

O arquivo não será alterado para manter o exemplo didático e para lhe fazer, conforme prometido, uma lavagem cerebral. Mas, se você quisesse salvar as alterações, bastava:

```
$ mv /tmp/ex quequeisso
```

Outro exemplo:

```
$ cat quequeisso | sed 's/ .*//'                  Observe que existe 1 espaço entre a 1ª / e o .
ATENCAO,
EH
O
interfaceando
Interpretador
Controle
Redirecionamento
Substituicao
Concatenacao
Execucao
Poderosa
```

O comando *cat* gerou uma listagem do arquivo *quequeisso* que foi entregue ao comando `sed` pelo *pipe* (**|**). Este comando `sed` pede para **s**ubstituir tudo após o primeiro espaço (**/ .*/**) por nada (**//**), ou seja todos os caracteres após o primeiro espaço são deletados.

- Função imprime (print): ***p***

  Esta função serve para reproduzir linhas de um endereço especificado ou que atendam a determinado argumento de pesquisa.

Exemplos:

```
$ sed '/UNIX/p' quequeisso                        O p imprime a linha que atenda à expressão
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!
O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,
O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:
Interpretador de comandos;
Controle do ambiente UNIX;
Controle do ambiente UNIX;
Redirecionamento de entrada e saida;
Substituicao de nomes de arquivos;
Concatenacao de pipe;
Execucao de programas;
```

```
Poderosa linguagem de programacao.
Além disto, o comando sed tambem possui diversas opções que apresentaremos
a seguir.
```

Repare que as linhas que continham a cadeia UNIX (argumento de pesquisa) foram duplicadas. Vamos fazer um pouco diferente:

```
$ sed '/UNIX/!p' quequeisso                          Repare o ! negando o p
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!
EH UMA LAVAGEM CEREBRAL!!!
O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:
Interpretador de comandos;
Interpretador de comandos;
Controle do ambiente UNIX;
Redirecionamento de entrada e saida;
Redirecionamento de entrada e saida;
Substituicao de nomes de arquivos;
Substituicao de nomes de arquivos;
Concatenacao de pipe;
Concatenacao de pipe;
Execucao de programas;
Execucao de programas;
Poderosa linguagem de programacao.
Poderosa linguagem de programacao.
```

Neste caso, o ponto de espantação (ou exclamação, como preferir) serve para negar o ***p***, isto é, serão impressas as linhas que não atenderem ao argumento de pesquisa (possuir a cadeia UNIX). Repare portanto que as linhas que possuem a cadeia UNIX não são repetidas, ao passo que as outras o são.

- Função deleta linha: ***d***

  Deleta as linhas referenciadas ou que atendem ao argumento de pesquisa.

Exemplos:

```
$ sed '1,4d' quequeisso                              Delete da linha 1 até a 4
Interpretador de comandos;
Controle do ambiente UNIX;
Redirecionamento de entrada e saida;
Substituicao de nomes de arquivos;
Concatenacao de pipe;
Execucao de programas;
Poderosa linguagem de programacao.
$ sed '/UNIX/d' quequeisso                           Delete linhas que contenham UNIX
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:
Interpretador de comandos;
Redirecionamento de entrada e saida;
Substituicao de nomes de arquivos;
Concatenacao de pipe;
Execucao de programas;
Poderosa linguagem de programacao.
```

Observe que nos exemplos acima foram deletadas as linhas seguintes: no primeiro caso as que foram apontadas pelos seus endereços e, no segundo, as que satisfizessem uma condição (possuírem a cadeia UNIX).

- Função ***a***crescenta: ***a***

  Acrescenta após o endereço informado, uma nova linha cujo conteúdo vem a seguir.

A sintaxe do comando usando a função fica assim:

```
$ sed '<endereço>a\                    A \ é obrigatória e serve para anular o new-line
> <texto a ser inserido>' arquivo
```

A primeira linha do comando deve ser interrompida após o `a`, para tal usa-se a barra invertida. O primeiro > das linhas seguintes é o *prompt secundário* (`PS2`) do UNIX.

Exemplos:

```
$ sed '2a\
> 3 - Como era de se esperar a linha 3 fica após a segunda linha'\
>quequeisso
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!
3 - Como era de se esperar a linha 3 fica após a segunda linha.
O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:
Interpretador de comandos;
Controle do ambiente UNIX;
Redirecionamento de entrada e saida;
Substituicao de nomes de arquivos;
Concatenacao de pipe;
Execucao de programas;
Poderosa linguagem de programacao.
```

- Função ***i***nsere: ***i***

  Idêntica à função **a**crescenta, porém não insere após o endereço especificado mas sim antes deste endereço.

- Função troca: ***c*** (***c***hange)

  Idêntica à função ***a***crescenta, porém não insere após o endereço especificado mas sim troca o conteúdo deste endereço pelo informado no comando.

- Função finaliza: ***q*** (***q***uit)

  Serve para marcar o ponto de término de execução do *sed,* o que em alguns casos é necessário.

Exemplos:

Para listarmos o nosso velho e bom quequeisso até a primeira ocorrência da cadeia UNIX, poderíamos fazer:

```
$ cat quequeisso | sed '/UNIX/q'              q encerra o comando quando encontrar UNIX
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!
O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,
```

ou então:

```
$ sed '/UNIX/q' quequeisso                    q encerra o comando quando encontrar UNIX

ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!
O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,
```

A opção -n

Muito foi falado que o `sed` sempre transcrevia todas as linhas da entrada para a saída, este é o seu *default*. Para que possamos listar somente as linhas que atendem ao(s) critério(s) de pesquisa ou ao(s) endereço(s) especificado(s), devemos usar a opção **-n**, que fala ao `sed`:

- Não (**n**ot) mande nada para a saída, a não ser que algo explicitamente o mande fazê-lo.

E, é claro, o `sed` obedece. E como você já vai perceber, a opção **-n** vem sempre acompanhada da função **p** que é quem explicita a necessidade de mandar os dados para a saída.

Exemplos:

Vamos separar as linhas de `quequeisso` que contenham pelo menos um " `de` " (um `de` compreendido entre 2 espaços), das que não contenham.

Linhas que contêm a cadeia:

```
$ sed -n '/ de /p'  quequeisso          Observe que entre as / e o de existe um espaço
O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,
Interpretador de comandos;
Redirecionamento de entrada e saida;
Substituicao de nomes de arquivos;
Concatenacao de pipe;
Execucao de programas;
Poderosa linguagem de programacao.
```

Linhas que não contêm a cadeia:

```
$ sed -n '/ de /!p'  quequeisso         Observe o ! negando o p
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:
Controle do ambiente UNIX;
```

2.2 - A Família de Comandos GREP

O comando `grep` é um comando bastante conhecido, mas o que normalmente não sabemos é que existem mais dois irmãos na sua família, além de diversas opções.

A finalidade básica dos comandos é localizar cadeias de caracteres em uma entrada definida, que pode(m) ser arquivo(s), a saída de um comando passada através de um *pipe* (**|**), ou a entrada padrão se nenhuma outra for especificada. Na verdade, esta localização dá-se por meio de expressões regulares sofisticadas que obedecem ao padrão *ed*, dai o nome do comando *(**G**lobal **R**egular **E**xpression **P**rint).*

Exemplos:

```
$ grep UNIX quequeisso                    Entrada do grep é  um arquivo
O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,
Controle do ambiente UNIX;
```

Neste exemplo, o `grep` foi usado para listar as ocorrências da cadeia UNIX no famigerado quequeisso.

```
$ grep grep *.sh                          Entrada do grep são os arqs terminados em .sh
leol.sh:     if  [ `echo $FrameOL | grep -c "$OL"` -ne 0 ]
pegasub.sh:  grep "D.SUB.GER.*.Z" "/tmp/Dir$$" > "/tmp/dir$$"
sub.sh:      if [ `grep -c "530 Login attempt failed" "/tmp/$$"` -ne 0 ]
transsub.sh: Linha=`grep '^'$OLM /usr/local/var/ArqOLs`
transsub.sh: grep ' ON PKBENEF.$' DIR.$OL > DIR
```

Neste exemplo, o `grep` lista todas as linhas, de todos arquivos terminados em `.sh`, no diretório corrente (note que arquivo = `*.sh`), que contenham a cadeia `grep`. Note que as linhas são precedidas pelo nome do arquivo onde se encontravam.

```
$ ps -ef | grep julio                     Entrada do grep é um comando
 root  20120  20118  14  09:22:14  pts/2  0:00  login -h TERM=vt220 julio
julio   7827  20121  12  15:02:20  pts/2  0:00 ps -ef
julio  20121  20120  80  09:22:26  pts/2  0:01 -ksh
julio   7828  20121   1  15:02:20  pts/2  0:00 grep julio
```

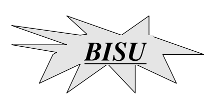
Neste exemplo, o `grep` foi usado como saída do comando `ps` (***p***rogram ***s***tatus) para localizar os processos em execução pelo usuário `julio`. Note que a 2ª linha é referente ao `ps` e a última ao `grep`.

Até aí tudo bem, você já sabia tudo isto. Vamos agora incrementar estes comandos, que são fundamentais na elaboração de *scripts* usando a linguagem Shell. Antes, porém, vamos conhecer sua família, que possui três membros:

- Comando `grep` - Que pesquisa cadeias de caracter a partir de uma entrada definida, podendo ou não usar expressões regulares.

- Comando `egrep` - Idêntico ao `grep`, porém mais poderoso e mais lento. Este comando, além de permitir o uso de expressões regulares, permite-nos *envenená-las* com o uso de parênteses para agrupá-las (somente neste ambiente), bem como com o operador lógico **|** (lógico somente no contexto do `egrep` já que aqui ele representa *ou).*

- Comando `fgrep` - O rapidinho da família, seu uso é indicado para expressões regulares que não possuam metacaracteres. Seu uso, nos casos possíveis, deve ser encorajado, uma vez que é 30% mais veloz que o `grep` e até 50% mais que o `egrep`.



CUIDADO ao usar as expressões regulares nestes comandos, pois os caracteres $, *, [, ^, |, (,), e \ são significantes para o Shell e podem desvirtuar totalmente sua interpretação. Lembre-se de colocar as expressões regulares entre apóstrofos.

Com as informações que acabamos de adquirir, agora podemos afirmar que nos 3 últimos exemplos, o uso do `grep` não é o mais indicado. Em ambos os casos deveria ter sido usado o comando `fgrep`, já que nenhum deles necessita o uso de metacaracter.

Exemplos:

Veja só issso:

```
$ egrep '^(i|I)' quequeisso                    Localizar linhas começadas(^) por i ou I
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:
Interpretador de comandos;
```

e isso:

```
$ grep '^(i|I)' quequeisso                     Localizar linhas começadas(^) por (i|I)
$                                              A pesquisa não devolveu nem uma linha sequer
```

Note que ao tentarmos localizar as linhas do arquivo começadas por um *i ou I* com o `grep`, nada retornou, sequer erro, pois este comando não interpreta parênteses nem o `ou` (**|**). Desta forma, foi executada uma pesquisa para localizar a cadeia *(i|I)* no *início da linha*. Caso tivéssemos usado o `fgrep` estaríamos tentando localizar a cadeia ^*(i|I)* em *qualquer ponto da linha*.

Para localizarmos todos os arquivos comuns do nosso diretório:

```
$ ls -la | grep '^-'                           Estou procurando um traço(-) na 1ª posição
-rw-r--r--   1 julio    dipao          624 Dec 10  1996 case.sh
-rw-r--r--   1 julio    dipao          416 Sep  3 10:53 quequeisso
-rw-r--r--   1 julio    dipao          415 Aug 28 16:40 quequeisso~
-rwsr-xr-x   1 root     sys            179 Dec 10  1996 shell1.sh
-rwxr--r--   1 julio    dipao          164 Dec 10  1996 shell1.sh~
```

Para localizar os arquivos executáveis pelo seu dono devemos na primeira posição de um `ls -l` procurar um `-` e na quarta um `x` *ou um* `s` (arquivos com *SUID*, como veremos mais tarde).

```
$ ls -la | egrep '^-..(x|s)'                   Começa(^) com - qq. 2 carac(..) um x ou s(x|s)
-rwsr-xr-x   1 root     sys            179 Dec 10  1996 shell1.sh
-rwxr--r--   1 julio    dipao          164 Dec 10  1996 shell1.sh~
```

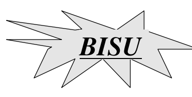
Resultado idêntico ao anterior, porém tempo de execução inferior:

```
$ ls -la | grep '^-..[xs]'
-rwsr-xr-x   1 root     sys            179 Dec 10  1996 shell1.sh
-rwxr--r--   1 julio    dipao          164 Dec 10  1996 shell1.sh~
```

Para apimentar um pouco mais estes três comandos, eles podem ser usados com diversas opções. Vejamos as mais importantes:

- A opção ***-c*** (**c**ount ou contar)

  Esta opção devolve a quantidade de linhas em que foi encontrada a cadeia alvo da pesquisa. É extremamente útil quando estamos desenvolvendo rotinas de crítica.



CUIDADO! a opção *-c* conta a quantidade de linhas que contêm uma determinada cadeia de caracteres e não quantas vezes esta cadeia foi encontrada.

Exemplos:

```
$ echo Existem `grep -c '^Maria' CadPess` Marias trabalhando na Empresa
Existem 224 Marias trabalhando na Empresa
```

No exemplo acima, foi considerado que o nome era o 1º campo do cadastro de pessoal por isso usamos o **^** . As crases foram colocadas para priorizar a execução do comando `grep`, já que quando a mensagem fosse ecoada, já seria necessária a quantidade de Marias encontradas pelo `grep`.

- A opção ***-l***

  Algumas vezes você quer saber quais os arquivos que possuem uma determinada cadeia de caracteres, sem se interessar por nenhum detalhe destes arquivos. Para tal usa-se a opção ***-l***. Esta opção é utilíssima para descobrir programas que necessitam manutenção.

Exemplos:

Suponhamos que você queira saber quantos programas terão de ser alterados caso se renomeie um arquivo que atualmente chama-se *ArqOLs*. A primeira, mais óbvia e tradicional, porém menos esperta é a seguinte:

```
$ cd dirscripts                                   Diretório onde residem os scripts Shell
$ fgrep ArqOLs *.sh                               Pesquisa os arquivos com extensão .sh (scripts)
listadir.sh:                   Maq=`grep "^$OL" /usr/local/var/ArqOLs |`
listadir.sh:     if [ `grep -c "^$OL" /usr/local/var/ArqOLs` -eq ]
listadir.sh:           Maq=`grep "^$OL" /usr/local/var/ArqOLs | cut -f2`
listusu.sh:    cat /usr/local/var/ArqOLs |
transcdb.sh:         *) NomeOL=`grep "^$Linha" $DirVar/ArqOLs | cut -f2`
transcdb.sh:         *) NomeOL=`grep "^$OL" $DirVar/ArqOLs | cut -f2`
transcdb.sh:cat $DirVar/ArqOLs |
transcdb01.sh:       *) NomeOL=`grep "^$Linha" $DirVar/ArqOLs | cut -f2`
transcdb01.sh:       *) NomeOL=`grep "^$OL" $DirVar/ArqOLs | cut -f2`
transcdb01.sh:cat $DirVar/ArqOLs |
transsub.sh:    Linha=`grep '^'$OLM /usr/local/var/ArqOLs`
transsub.sh:    Site=`echo "$Linha"  | cut -f2`   #####  ArqOLs no Format
uparau.sh:        Maq=`grep "^$OL" /usr/local/var/ArqOLs | cut -f2`
uparau.sh:            Erro "Nao encontrei OL $OL em ArqOLs"
uparau.sh:} < /usr/local/var/ArqOLs
uparaug.sh:        Maq=`grep "^$OL" /usr/local/var/ArqOLs | cut -f2`
uparaug.sh:            Erro "Nao encontrei OL $OL em ArqOLs"
uparaug.sh:} < /usr/local/var/ArqOLs
```

Ou então de uma forma mais esperta e limpa:

```
$ cd dirscripts
$ fgrep -l ArqOLs *.sh
listadir.sh
listusu.sh
```

```
transcdb.sh
transcdb01.sh
transsub.sh
uparau.sh
uparaug.sh
```

Ou ainda para sabermos somente a quantidade de programas a serem alterados:

```
$ cd dirscripts
$ fgrep -l ArqOLs *.sh | wc -l                    wc -l conta linha e wc -c conta caracteres
7
```

- A opção ***-v***

Esta opção da família `grep` é particularmente útil para excluir registros de um arquivo, uma vez que ela devolve todas as linhas que não possuem a cadeia pesquisada.

Exemplos:

Neste exemplo, vamos remover um usuário chamado `kara`. Existem 3 formas de faze-lo: pelo `sysadm`, usando-se o comando `userdel -r kara` (a opção `-r` é para remover também o *home directory*) ou então:

```
$ rm -r ~kara                                Removeu o diretório home do Kara
$ grep -v 'kara' /etc/passwd > /tmp/passwd   Exclui o Kara, temporariamente do /etc/passwd
$ grep -v 'kara' /etc/shadow > /tmp/shadow   Ídem do /etc/shadow
$ mv /tmp/passwd /etc/passwd                 Efetivou a exclusão do Kara em /etc/passwd
$ mv /tmp/shadow /etc/shadow                 Ídem em /etc/shadow
```

2.3 - Os Comandos para Cortar e Colar

- **Cortando cadeias de caracteres - *cut***

Agora vamos falar em um comando utilissímo na elaboração de *scripts* - o `cut`.

Este comando é usado quando se deseja extrair campos ou pedaço de dados de arquivos, ou de qualquer outra entrada. Seu formato geral é:

```
cut -ccaracteres arquivo
```

Onde *`caracteres`* é a porção que se deseja cortar de cada registro de *arquivo*. Isto pode ser um simples número ou uma faixa. Estas atribuições podem ser:

```
cut -cposição arq              Retorna todos os caracteres de posição
cut -cposini-posfim arq        Retorna todas as cadeias entre posini e posfim
cut -cposini- arq              Retorna todas as cadeias a partir de posini
cut -c-posfim arq              Retorna todas as cadeisa do início até posfim
```

Exemplos:

```
 $ who
ciro       pts/0       Sep  8 09:02
norberto   pts/2       Sep  8 15:57
ney        pts/3       Sep  8 14:51
luis       pts/4       Sep  8 16:23
hudson     pts/5       Sep  8 10:33
```

```
julio      pts/6        Sep  8 11:19
```

```
$ who | cut -c-8                                  Extrair da saída do who até o 8º caracter
ciro
norberto
ney
luis
hudson
julio
```

```
$ who | cut -c12-17                               Extrair da saída do who do 12º até o 17º carac.
pts/0
pts/1
pts/2
pts/3
pts/4
pts/5
```

```
$ who | cut -c25-                                 Extrair da saída do who a partir do 25º caracter
Sep  8 09:02
Sep  8 15:57
Sep  8 14:51
Sep  8 16:23
Sep  8 10:33
Sep  8 11:19
```

Neste exato momento, aparentemente chegamos a um impasse porque você vai me dizer:

-Esse tal de `cut` nem de longe serve pra mim. O que um programador mais precisa não é de extrair posições predefinidas, mas sim de determinados campos que nem sempre ocorrem nas mesmas posições do mesmo arquivo.

E eu vou lhe responder:

- Você esta coberto de razão. Precisamos muito extrair campos predefinidos dos arquivos; por isso vou lhe contar como se faz isso.

Existem 2 opções do *cut* que servem para especificar o(s) campos dos registros que desejamos extrair. São elas:

A opção **-f** (**f**ield) - Serve para especificar os campos (**f**ields) que desejamos extrair. Obedece às mesmas regras do **-c**caracter. Isto é:

```
cut -fcampo arq                       Retorna o campo campo
cut -fcampoini-campofim arq           Retorna campos de campoini a campofim
cut -fcampoini- arq                   Retorna todos os campos a partir de campoini
cut -f-campofim arq                   Retorna todos os campos do início ao campofim
```

Bem tudo o que foi falado sobre a opção **-f** só serve se o delimitador de campos for o caracter `<TAB> (/011)`, que é o delimitador *default*. Para podermos generalizar seu uso devemos conhecer a opção:

A opção **-d** (**d**elimitador) - Ao usarmos esta opção, descrevemos para o *cut* qual será o separador de campos do arquivo. No uso desta opção é necessário tomarmos cuidado para proteger (com aspas ou apóstrofos) os caracteres que o Shell possa interpretar como metacaracteres. Seu formato geral é:

```
cut -fcampos [-ddelimitador] arquivo
```

Onde *delimitador* é o caracter que delimita todos os campos de *arquivo*.

Exemplos:

```
$ tail -4 /etc/passwd                                   Listando os últimos 4 registros de /etc/passwd
aluno4:x:54084:17026:Curso Unix Basico:/dsv/usr/aluno4:/usr/bin/ksh
aluno5:x:54085:17026:Curso Unix Basico:/dsv/usr/aluno5:/usr/bin/ksh
aluno6:x:54086:17026:Curso Unix Basico:/dsv/usr/aluno6:/usr/bin/ksh
aluno7:x:54087:17026:Curso Unix Basico:/dsv/usr/aluno7:/usr/bin/ksh
$ tail -4 /etc/passwd | cut -f1 -d:                     Extraindo da saída do tail somente o 1º campo
aluno4
aluno5
aluno6
aluno7
```

O arquivo `telefones` foi criado com 2 campos separados por um `<TAB>`, e com o seguinte conteúdo:

```
$ cat telefones
Ciro Grippi      (021)555-1234
Ney Gerhardt     (024)543-4321
Enio Cardoso     (023)232-3423
Claudia Marcia   (021)555-2112
Paula Duarte     (011)449-0989
Ney Garrafas     (021)988-3398
```

Para extrair somente os nomes:

```
$ cat telefones | cut -f1                               Delimitador <TAB> é o default. Não especifiquei
Ciro Grippi
Ney Gerhardt
Enio Cardoso
Claudia Marcia
Paula Duarte
Ney Garrafas
```

Quero somente o primeiro nome:

```
$ cat telefones | cut -f1 -d                            Depois do -d eu coloquei um espaço
cut: option requires an argument -- d                   ZEBRA!!!!
cut: ERROR: Usage: cut [-s] [-d<char>] {-c<list> | -f<list>} file ...
```

ZEBRA!!!! Eu queria pegar o nome que estivesse antes do delimitador *espaço* mas esqueci que *espaços*, em Shell, devem estar entre aspas. Vamos tentar novamente:

```
$ cat telefones | cut -f1 -d" "                         O d" " indica que o delimitador é branco
Ciro
Ney
Enio
Claudia
Paula
Ney
```

Para extrair o código de DDD:

```
$ cat telefones | cut -f2 -d"(" | cut -f1 -d")"        Repare que o "(" e o ")" estão entre aspas
021
024
023
021
011
021
```

Se não colocássemos os parênteses entre aspas, olha só a encrenca:

```
$ cat telefones | cut -f2 -d( | cut -f1 -d)
ksh: syntax error: `(' unexpected
```

- **Colando cadeias de caracteres - *paste***

O `paste` funciona como um `cut` de marcha à ré, isto é, enquanto o `cut` separa pedaços de arquivos, o `paste` junta-os. Seu formato genérico é:

```
paste arquivos
```

Onde cada linha dos *arquivos* especificados serão coladas entre si, de maneira a formar linhas únicas que serão mandadas para a *saída padrão*. Vejamos os exemplos:

Exemplos:

```
$ cat /etc/passwd
bolpetti:x:54000:1001:Joao Bolpetti Neto - DISB.O 821-
moswaldo:x:54001:1001:Marco Oswaldo da Costa Freitas  diteo 2338
jneves:x:54002:1001:Julio Cezar Neves 821-6339
cgrippi:x:54003:1001:Ciro Grippi Barbosa Lima - ramal (821)2338
$ cat /etc/passwd | cut -f1 -d: > /tmp/logins     1º campo (-f1) mandado para /tmp/logins
$ cat /etc/passwd | cut -f3 -d: > /tmp/uid        3º campo (-f3) mandado para /tmp/uid
$ paste /tmp/logins /tmp/uid                      Juntando as linhas de /tmp/logins e /tmp/uid
bolpetti        54000
moswaldo        54001
jneves  54002
cgrippi 54003
```

Neste exemplo, tiramos um pedaço do `/etc/passwd` e dele extraímos campos para outros arquivos. Finalmente, como exemplo didático, juntamos os dois arquivos dando a saída na tela. Note que o `paste` inseriu entre os 2 campos, o separador `<TAB>`, seu *default*.

- A opção **-d** (**d**elimitador)

Analogamente ao `cut`, a sintaxe do `paste` permite o uso de delimitador, que também deve ser protegido (com aspas ou apóstrofos) se porventura o Shell puder interpreta-lo como um metacaracter.

No exemplo anterior, caso quiséssemos alterar o separador para, digamos, **:** teríamos que fazer:

```
$ paste -d: /tmp/logins /tmp/uid
bolpetti:54000
moswaldo:54001
jneves:54002
cgrippi:54003
```

- A opção **-s**

A opção **-s** serve para transformar linhas de um arquivo em colunas, separadas por `<TAB>`:

Exemplos:

```
$ paste -s /tmp/logins
bolpetti        moswaldo        jneves  cgrippi
```

Se quiséssemos os campos separados por espaço e não por `<TAB>` faríamos:

```
$ paste -s -d" " /tmp/logins
bolpetti moswaldo jneves cgrippi
```

2.4 - O *tr* *tr*aduz, *tr*anscreve ou *tr*ansforma Cadeias de Caracteres?

Avalie você mesmo a sua finalidade! No duro o `tr` recebe dados da *entrada padrão* (que a esta altura você já sabe que pode ser *redirecionado*), convertendo-os mediante um padrão especificado. Sua sintaxe é a seguinte:

```
tr <dos-caracteres> <para-os-caracteres>
```

onde `<dos-caracteres> e <para-os-caracteres>` são um ou mais caracteres. Qualquer caracter vindo da entrada definida, que coincida com um caracter de `<dos-caracteres>` será convertido para o seu correspondente em `<para-os-caracteres>`. O resultado desta conversão será enviado para a saída.

Exemplos:

Vamos pegar de volta o nosso velho `quequeisso`.

```
$ cat quequeisso
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,
EH UMA LAVAGEM CEREBRAL!!!
O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:
Interpretador de comandos;
Controle do ambiente UNIX;
Redirecionamento de entrada e saída;
Substituicao de nomes de arquivos;
Concatenacao de pipe;
Execucao de programas;
Poderosa linguagem de programacao.
```

Na sua forma mais simples podemos usar o `tr` para fazer simples substituições. Assim, se quiséssemos trocar toda letra *a* por letra *z* faríamos:

```
$ cat quequeisso |                              Estou listando o quequeisso para ...
> tr a z                                        converter a letra a na letra z
ATENCAO, O TEXTO ABAIXO NAO EH TREINAMENTO,     O A não sofreu alteração porque é maiúsculo
EH UMA LAVAGEM CEREBRAL!!!
O Shell zlem de znzliszr czdz dzdo entrzdo z pzrtir do prompt do UNIX,
interfzceando com os usuzrios, tem tzmbem zs seguintes ztribuicoes:
Interpretzdor de comzndos;
Controle do zmbiente UNIX;
Redirecionzmento de entrzdz e szmdz;
Substituiczo de nomes de zrquivos;
Conczteńzczo de pipe;
Execuczo de progrzmzs;
Poderosz linguzgem de progrzmzczo.
```

As duas linhas acima têm um bom uso, servem para desenvolver linguagens com grau de complexidade semelhante à lingua do *PÊ*...

Poderíamos também fazer a conversão, usando a entrada de outra forma:

```
$ tr A a < quequeisso
aTENCaO, O TEXTO aBaIXO NaO EH TREINaMENTO,
EH UMa LaVaGEM CEREBRaL!!!
O Shell alem de analisar cada dado entrado a partir do prompt do UNIX,
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:
Interpretador de comandos;
Controle do ambiente UNIX;
Redirecionamento de entrada e saida;
Substituicao de nomes de arquivos;
Concatenacao de pipe;
Execucao de programas;
Poderosa linguagem de programacao.
```

Neste último exemplo vimos como converter os **A** (maiúsculos) em **a** (minúsculos).

Nos dois casos anteriores a entrada dos dados foi feita, ou por um *pipeline* de comandos ou redireciondo a entrada. Isto deve-se ao fato do `tr` estar esperando os dados pela *entrada primária*.

Um exemplo mais lógico do `tr`. Suponha que queiramos tabelar os *login names* e os *home directories* dos dez primeiros registros de */etc/passwd*. Observe a execução da instrução abaixo:

```
$ head -10 /etc/passwd |
> cut -f1,6 -d:
daemon:/
bin:/usr/bin
sys:/
adm:/var/adm
uucp:/usr/lib/uucp
lp:/var/spool/lp
nuucp:/var/spool/uucppublic
listen:/usr/net/nls
sync:/
```

Separo os 10 primeiros registros de /etc/passwd para pegar o 1º e 6º campos

Da execução do comando acima, resultaram o 1º e 6º campos dos 10 primeiros registros de /etc/passwd. O separador permanece 2 pontos

Então para realizarmos o proposto, deveríamos:

```
$ head -10 /etc/passwd |
> cut -f1,6 -d: |
> tr : '	'
root	/
daemon	/
bin	/usr/bin
sys	/
adm	/var/adm
uucp	/usr/lib/uucp
lp	/var/spool/lp
nuucp	/var/spool/uucppublic
listen	/usr/net/nls
sync	/
```

Idêntico ao anterior, porém troco : por ***<TAB>***

O `tr` também pode usar caracteres octais tanto para especificar o que estamos pesquisando, quanto o resultado que esperamos.

A representação octal é passada para o `tr` no seguinte formato:

```
\nnn
```

Suponha agora que eu tenho um arquivo com comandos separados por ponto-e-virgula e eu queira colocar cada comando em uma linha.

Vejamos como está este arquivo:

```
$ cat confusao
cd $HOME;pwd;date;ls -la;echo $LOGNAME ${SHELL}x
```

Para coloca-lo de forma legível, podemos fazer:

```
$ cat confusao | tr ";" "\012"                    Converte : em <ENTER>
cd $HOME
pwd
date
ls -la
echo $LOGNAME x${SHELL}x$
```

O maior uso deste comando é no entanto transformar maiúsculas em minúsculas e vice-versa. Vejamos como converter as letras maiúsculas de *quequeisso* em letras minúsculas:

```
$ cat quequeisso |
> tr '[A-Z]' '[a-z]'
atencao, o texto abaixo nao eh treinamento,
eh uma lavagem cerebral!!!
o shell alem de analisar cada dado entrado a partir do prompt do unix,
interfaceando com os usuarios, tem tambem as seguintes atribuicoes:
interpretador de comandos;
controle do ambiente unix;
redirecionamento de entrada e saida;
substituicao de nomes de arquivos;
concatenacao de pipe;
execucao de programas;
poderosa linguagem de programacao.
```

2.5 - Exprimindo o *expr* de Forma Expressa

O `expr` é o nosso comando bom-bril, isto é, tem 1001 utilidades. Vejamos as principais:

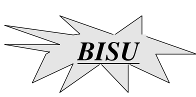
- Execução de operações aritméticas -

As operações aritméticas são executadas com auxílio do comando `expr`. Os operadores são:

+	Soma
-	Subtração
*	Multiplicação
/	Divisão
%	Resto da divisão

Exemplos:

```
$ expr 2+7
2+7                                     Que foi que houve?...
$ expr 2 + 7
9                                       Agora sim, com espaço separando operadores
$ expr 2 - 7
-5
$ expr 2 * 7
expr: syntax error                      Zebra, falta proteger o asterisco

$ expr "2 * 7"
2 * 7                                   Agora eu protegi a expressão toda...
$ expr 2 \* 7
14                                      Agora sim...
$ expr 2 % 7
2
$ expr 7 % 2
1
```



Precisamos cautela na execução deste comando, já que o mesmo trabalha somente com números inteiros, podemos, neste caso, dizer que: *a ordem dos tratores altera o pão duro*. Vejamos:

```
$ expr 7 / 5 \* 10
10                                      7 dividido por 5 = 1 (inteiro), vezes 10 = 10
$ expr 7 \* 10 / 5
14                                      7 vezes 10 = 70 dividido por 5 = 14
```

- Para medir o tamanho de uma cadeia de caracteres

Com a sintaxe:

```
expr length cadeia
```

Exemplos:

Assim:

```
$ expr length 5678
4                                       4 é o tamanho da cadeia 5678
```

- Para extrair uma subcadeia de uma cadeia de caracteres

Com a sintaxe:

```
expr substr cadeia <a partir da posição> <qtd. caracteres>
```

Exemplos:

Assim:

```
$ expr substr 5678 2 3                  Extrair a partir da 2ª posição 3 caracteres
678
$ expr substr "que teste chato" 11 5    Extrair a partir da 11ª posição 5 caracteres
chato
```

- Para encontrar uma subcadeia em uma cadeia

Com a sintaxe:

```
expr index cadeia subcadeia
```

Exemplos:

Assim:

```
$ expr index 5678 7                    Em que posição de 5678 está o 7?
3
```

Exercícios

Descubra o resultado da execução dos comandos abaixo:

```
sed 's#UNIX#unix#' quequeisso          sed -n /UNIX/p quequeisso

sed '1,2s/[A-Z]//'                     grep '^(P|S)' quequeisso

sed '1,2s/[a-z]//g'                    egrep '^(E|C)' quequeisso

sed 's/ .*//' quequeisso               egrep -v '^(E|C)' quequeisso | wc -l

cat quequeisso | sed 's/ .*//'         fgrep -l ou *

expr 15 / 2 * 5                        grep ecardoso /etc/passwd | tr : -

expr 15 * 5 / 2                        expr length "C qui sabe"

finger | cut -c10-30 | cut -f1 -d " " | tr "[a-z]" "[A-Z]"
```

Capítulo 3

E Nós Viemos Aqui Para Falar ou Para Programar?

Depois de todo este blablablá vamos finalmente ver como funciona esta tal de programação em Shell. Eu não quero te enganar não, mas você já fez vários programas, uma vez que cada um dos comandos que você usou nos exemplos até aqui, é um mini programa.

O Shell, por ser uma linguagem interpretada, pode receber os comandos diretamente do teclado e executa-los instantaneamente, ou armazená-los em um arquivo (normalmente chamado de *script*) que, posteriormente, pode ser processado tantas vezes quanto necessário. Até aqui o que vimos enquadra-se no primeiro caso. Veremos agora o que fazer para criarmos os nossos arquivos de programas.

3.1 - Executando um Programa (Sem Ser na Cadeira Elétrica)

Vamos mostrar o conteúdo do arquivo DuLoren:

```
$ cat DuLoren                                   Do 1º script a gente nunca se esquece...
#
#  Meu Primeiro Script em Shell
#

echo Eu tenho `cat telefones | wc -l` telefones cadastrados
echo "Que sao:"
cat telefones
```

Ih! É um programa! Vamos então executá-lo:

```
$ DuLoren
ksh: DuLoren: cannot execute
```

Ué, se é um programa porque não posso executá-lo?

```
$ ls -l DuLoren
-rw-r--r--   1 julio    dipao           90 Sep 29 16:19 DuLoren
```

Para ser executável, é necessário que aqui tenha um ***x***

Então, devemos antes de tudo fazer:

```
$ chmod 744 DuLoren                              Ou $ chmod +x DuLoren
$ ls -l DuLoren
-rwxr--r--   1 julio    dipao           90 Sep 29 16:19 DuLoren
```

Agora sim, vamos ver:

```
$ DuLoren
Eu tenho 6 telefones cadastrados
Que sao:
Ciro Grippi     (021)555-1234
Ney Gerhardt    (024)543-4321
Enio Cardoso    (023)232-3423
Claudia Marcia  (021)555-2112
Paula Duarte    (011)449-0989
Ney Garrafas    (021)988-3398
```

3.2 - Usando Variáveis

O Shell, como qualquer outra linguagem de programação, trabalha com variáveis. O nome de uma variável é iniciado por uma letra ou um sublinha (_), seguido ou não por quaisquer caracteres alfanuméricos ou caracteres sublinha.

- **Para criar variáveis:**

Para armazenar ou atribuir valor a uma variável basta colocar o nome da variável, um sinal de igual (**=**) colado no nome escolhido e colado ao sinal de igual, o valor estipulado. Assim:

Exemplos:

```
$ variavel=qqcoisa        Armazena qqcoisa em variavel
$ contador=0              Coloca zero na variável contador
$ vazio=                  Cria a variável vazio com o valor nulo (NULL)
```

Repare nos exemplos acima que ao atribuírmos valor a uma variável, não colocamos espaços em branco no corpo desta atribuição, sob pena de ganharmos um erro de sintaxe.

- **Para exibir o conteúdo das variáveis:**

Para exibirmos o valor de uma variável, devemos preceder o seu nome por um cifrão (**$**). Assim se quisermos exibir o conteúdo das variáveis criadas acima, faríamos:

Exemplos:

```
$ echo variavel=$variavel, contador=$contador, vazio=$vazio
variavel=qqcoisa, contador=0, vazio=
```

Se nosso desejo for listar o conteúdo da variável `contador` seguido do número 1 e fizermos:

```
$ echo $contador1
```

O resultado será nulo, porque o Shell interpretará o comando acima como uma solicitação para que seja exibido o conteúdo da variável *contador1*. Como tal variável não existe, o valor retornado será nulo.

Como fazer então? O Shell possui o recurso de delimitar variáveis colocando-as entre chaves(**{ }**). Desta forma, para solucionar o problema acima, poderíamos fazer:

Exemplos:

```
$ echo ${contador}1
01                                          O contador tem 0, e colocamos o 1 a seguir
```

3.3 - Passando e Recebendo Parâmetros:

Suponha que você tenha um programa chamado `listdir`, assim:

```
$ cat listdir
echo Os Arquivos do Diretorio Corrente Sao:
ls -l
```

Que quando executado geraria:

```
$ listdir
Os Arquivos do Diretorio Corrente Sao:
-rwxr--r--   1 julio    dipao           90 Sep 29 16:19 DuLoren
-rwxr--r--   1 julio    dipao           51 Sep 29 17:46 listdir
-rw-r--r--   1 julio    dipao          416 Sep  3 10:53 quequeisso
-rw-r--r--   1 julio    dipao          137 Sep  9 11:33 telefones
```

Ora, este programa seria estático, só listaria o conteúdo do diretório corrente. Para tornar o programa mais dinâmico nosso *script* deveria receber o nome do diretório que desejássemos listar. Dizemos que estamos executando um *script* passando parâmetros, e sua forma geral é a seguinte:

```
progr parm1 parm2 parm3 . . . parmn
```

Isto é chama-se o programa da forma habitual e, em seguida, separados por espaços em branco vêm os parâmetros.

Sobre esta chamada de `progr` acima, devemos nos ater a diversos detalhes:

- Os parâmetros passados (`parm1, parm2...parmn`) dão origem, dentro do programa que as recebe, às variáveis `$1, $2, $3....$n`. É importante notar que a passagem de parâmetro é posicional, isto é, o primeiro parâmetro é o `$1`, o segundo `$2` e assim sucessivamente, até `$9`.

Exemplos:

```
$ cat param1                                Programa didático sobre passagem de parâmet.
echo $1                                     Lista 1º parâmetro
echo $2                                     Lista 2º parâmetro
echo $11                                    Lista 11º parâmetro
$ param1 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. Executa programa passando 11 parâmetros
1.                                          Listou 1º parâmetro
2.                                          Listou 2º parâmetro
1.1                                         Você esperava 11. e deu 1.1! Dá pra entender?
```

A explicação para o que aconteceu com o 11º parâmetro é que, apesar do Shell não limitar a quantidade de parâmetros que podem ser passados, só é possível endereçar diretamente até o 9º parâmetro.

- A variável `$0` conterá o nome do programa

Exemplos:

```
$ cat param2                                    Programa exemplo da variável $0
echo $0                                         Linha alterada em relação ao exemplo anterior
echo $2
echo $11
$ param2 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11.
param2                                          O parâmetro $0 devolveu o nome do programa
2.
1.1
```

- A variável `$#` conterá a quantidade de parâmetros recebida pelo programa.

Exemplos:

```
$ cat param3
echo O Programa $0 Recebeu $# Parâmetros        Observe o $0 e o $# incluídos nesta linha
echo $1
echo $2
echo $11
$ param3 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11.
O Programa param3 Recebeu 11 Parâmetros         $0 gerou nome do prg e $# o tot. de parâmetros
1.
2.
1.1
```

- A variável `$*` contém o conjunto de todos os parâmetros separados por espaços em branco;

Exemplos:

```
$ cat param4
echo O Programa $0 Recebeu $# Parametros Listados Abaixo:
echo $*
$ param4 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11.
O Programa param4 Recebeu 11 Parametros Listados Abaixo:
1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11.              O 10º e o 11º parâmetros foram listados
```

- O comando `shift n` mata os `n` primeiros parâmetros, o valor *default* de `n` e´1. Este é um dos comandos que nos permite passar mais que 9 parâmetros.

Exemplos:

```
$ cat param5                                    Modificando o programa param3...
echo O Programa $0 Recebeu $# Parâmetros
echo $1
echo $2
shift 10                                        Matei os 10 primeiros parâmetros e ...
echo $1                                         O 11º parâmetro se transforma no 1º
$ param5 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11.
O Programa param5 Recebeu 11 Parâmetros
1.
2.
11.                                             Agora sim...
```

Observe o uso das aspas no mesmo exemplo, a diferença que faz:

```
$ param5 "1. 2. 3. 4. 5. 6. 7." 8. 9. 10. 11.
O programa param5 Recebeu 5 Parametros
1. 2. 3. 4. 5. 6. 7.                          1º parâmetro porque as aspas agrupam
8.
param5[4]: shift: bad number                  Agora não existem 10 parâmetros para o shift
```

Apesar de podermos trabalhar com os parâmetros de ordem superior a nove das formas mostradas acima, existem outras maneiras, até mais intuitivas de fazê-lo, que serão apresentadas mais adiante.

Olha só este *script* para saber desde quando as pessoas se *logaram* (ARGH!!!).

```
$ cat kadeo
echo "$1 esta logado desde \c"                $1 é o 1º parm. O \c é para não saltar linha
who | grep $1 | cut -c25-                     Na saída do who pesquiso usuário e corto data
$ kadeo enio                                  Executando, $1 conterá enio
enio esta logado desde Sep 30 16:49
```

Alterando o `listdir`, que está no início da seção 3.3 para `listdir1`, de forma a listar o conteúdo de um diretório especificado, e não somente o diretório corrente como fazia o `listdir`, teríamos:

```
$ cat listdir1
cd $1                                         cd para o diretório que será passado
echo Os Arquivos do Diretorio $1 Sao:
ls -l
$ listdir1 /tmp                               Na execução passando o diretório /tmp
Os Arquivos do Diretorio /tmp Sao:
total 92
-rw-r--r--   1 enio     ssup           310 Sep 30 15:50 23009
-rw-------   1 root     root         10794 Sep 30 17:29 799.debug
-rw-r--r--   1 bolpetti ssup             0 Sep 30 08:40 csa
-rw-rw-r--   1 root     sys            592 Sep 29 20:07 sa.adrfl
-rw-rw-rw-   1 root     root            20 Sep 30 14:51 xx
```

Esta parte de passagem de parâmetros é muito importante, até agora, o que vimos foi muito superficial. Mais à frente veremos outros exemplos e aplicações que farão uma abordagem mais detalhada no tema.

3.4 - Desta Vez Vamos...

Você se lembra do arquivo de telefones dos exemplos anteriores?

```
$ cat telefones
Ciro Grippi     (021)555-1234
Ney Gerhardt    (024)543-4321
Enio Cardoso    (023)232-3423
Claudia Marcia  (021)555-2112
Paula Duarte    (011)449-0989
Ney Garrafas    (021)988-3398
```

Vamos colocá-lo em ordem para facilitar o seu uso a partir deste ponto:

```
$ sort -o telefones telefones                 O -o indica a saída do sort. O próprio telefones
$ cat telefones
Ciro Grippi     (021)555-1234
Claudia Marcia  (021)555-2112
Enio Cardoso    (023)232-3423
```

```
Ney Garrafas    (021)988-3398
Ney Gerhardt    (024)543-4321
Paula Duarte    (011)449-0989
```

- **Programa Para Procurar Pessoas no Arquivo de Telefones**

Você a esta altura já sabe que deve usar o comando `grep` para achar uma determinada pessoa neste arquivo:

```
$ grep Car telefones
Enio Cardoso    (023)232-3423
```

E também sabe que para procurar por um nome composto, você deve colocá-lo entre aspas:

```
$ grep "Ney Ge" telefones
Ney Gerhardt    (024)543-4321
```

Vamos então listar o "*bacalho"* para procurar uma pessoa que será passada como parâmetro e listar o seu registro.

```
$ cat pp
#
#    Pesquisa Pessoa no Catalogo Telefonico
#

grep $1 telefones
```

Vamos usá-lo:

```
$ pp Ciro
Ciro Grippi     (021)555-1234
$ pp "Ney Ge"                                  Entre aspas como manda o figurino
grep: can't open Ge                            Quequeisso minha gente???
telefones:Ney Gerhardt  (024)543-4321
telefones:Ney Garrafas  (021)988-3398
```

No último exemplo, cuidadosamente, coloquei *Ney Ge* entre aspas. O que houve? Onde foi que errei? Olhe novamente para o `grep` executado pelo programa `pp`:

```
grep $1 telefones
```

Mesmo colocando *Ney Ge* entre aspas, para que seja encarado como um único argumento, quando o `$1` foi passado pelo Shell para o comando `grep`, transformou-se em dois parâmetros. Desta forma o conteúdo final da linha, que o `grep` executou foi o seguinte:

```
grep Ney Ge telefones
```

Como a sintaxe do `grep` é:

```
grep <cadeia de caracteres> arq1, [arq2, arq3,...arqn]
```

O Shell entendeu que deveria procurar a cadeia de caracteres *Ney* nos arquivos *Ge* e *telefones* e por não existir o arquivo *Ge*, gerou o erro.

Para resolver este problema, basta colocar a variável `$1` entre aspas, da seguinte forma:

```
$ cat pp
#
#    Pesquisa Pessoa no Catalogo Telefonico - versao 2
#

grep "$1" telefones
$ pp Ciro
Ciro Grippi     (021)555-1234                   Continua funcionando...
$ pp "Ney Ge"                                   Agora é que são elas...
Ney Gerhardt    (024)543-4321                   Viu só? Não te disse?
```

- **Programa Para Inserir Pessoas no Arquivo de Telefones**

Vamos continuar o desenvolvimento da série de programas para manipular o arquivo de telefones. Você, provavelmente, desejará adicionar pessoas a este arquivo. Para tal vamos ver um programa chamado `add` que recebe dois argumentos: o nome da pessoa que será adicionada e o seu telefone:

```
$ cat add
#
# Adiciona Pessoas ao Arquivo de Telefones
#

echo "$1        $2" >> telefones
```

Observe que entre o `$1` e o `$2` do exemplo acima existe um caracter `<TAB>` que deve ser colocado entre aspas, para evitar que o Shell o transforme em um espaço simples.

Vamos executar este programa:

```
$ add "Joao Bolpetti" (011)224-3469
ksh: syntax error: `(' unexpected               Ops!!
```

Não se esqueça! *Os parênteses por si só têm significado para o Shell e devendo, portanto, sempre serem protegidos da sua interpretação*. Vamos tentar de outra forma:

```
$ add "Joao Bolpetti" "(011)224-3469"           Com os parênteses protegidos por aspas...
$ pp Bolp                                       Só para ver se achamos a nova entrada
Joao Bolpetti   (011)224-3469
$ cat telefones                                 Vamos ver o que houve
Ciro Grippi     (021)555-1234
Claudia Marcia  (021)555-2112
Enio Cardoso    (023)232-3423
Ney Garrafas    (021)988-3398
Ney Gerhardt    (024)543-4321
Paula Duarte    (011)449-0989
Joao Bolpetti   (011)224-3469
```

Quase!!! O programa só não est'a 100% porque o arquivo `telefones` não está mais classificado. Vamos aprimorar este programa:

```
$ cat add
#
# Adiciona Pessoas ao Arquivo de Telefones - Versao 2
#

echo "$1        $2" >> telefones
sort -o telefones telefones                     Classifica o arquivo. Saída nele mesmo
```

Vamos ver sua execução:

```
$ add "Luiz Carlos" "(021)767-2124"
$ cat telefones
Ciro Grippi     (021)555-1234
Claudia Marcia  (021)555-2112
Enio Cardoso    (023)232-3423
Joao Bolpetti   (011)224-3469
Luiz Carlos     (021)767-2124
Ney Garrafas    (021)988-3398
Ney Gerhardt    (024)543-4321
Paula Duarte    (011)449-0989
```

Então, cada vez que incluímos um registro, o arquivo `telefones` é classificado.

- **Programa Para Remover Pessoas do Arquivo de Telefones**

  Não existe sistema que permita consultar um arquivo, incluir dados e não possibilite a remoção de registros. Portanto, dando prosseguimento a esta série de *scripts*, desenvolveremos um programa chamado `rem` que terá como argumento o nome da pessoa a ser removida. Isto deverá ser feito usando-se a opção `-v` do comando `grep` (Veja na seção 2.2).

```
$ cat rem
#
# Remove Pessoas do Arquivo de Telefones
#

grep -v "$1" telefones > /tmp/$$
mv /tmp/$$ telefones
```

No programa acima o `grep` lista para `/tmp/$$`[3] o conteúdo de `telefones`, exceto os registros que tenham a cadeia de caracteres especificada em `$1`. Depois da execução do comando `grep`, o velho `telefones` será substituído pelo novo `/tmp/$$`.

Vejamos sua execução:

```
$ rem Joao Bolpetti
$ cat telefones
Ciro Grippi     (021)555-1234
Claudia Marcia  (021)555-2112
Enio Cardoso    (023)232-3423
Luiz Carlos     (021)767-2124
Ney Garrafas    (021)988-3398
Ney Gerhardt    (024)543-4321
Paula Duarte    (011)449-0989
$ rem Ney
$ cat telefones
Ciro Grippi     (021)555-1234
Claudia Marcia  (021)555-2112
Enio Cardoso    (023)232-3423
Luiz Carlos     (021)767-2124
Paula Duarte    (011)449-0989
```

No primeiro caso, *João Bolpetti* foi removido a contento. No segundo, no entanto, as entradas `Ney Garrafas` e `Ney Gerhardt` foram removidas, já que ambas satisfazem ao padrão de remoção informado. Vamos usar o programa `add` para cadastra-los de volta no `telefones`.

```
$ add "Ney Gerhardt" "(024)543-4321"
$ add "Ney Garrafas" "(021)988-3398"
```

[3] O diretótio /tmp é liberado para qualquer usuário poder gravar e o $$ representa o PID. Veja também nota de rodapé da página 1.4.

Então vimos, conforme já havia antecipado, que para fazermos programas em Shell, basta justapormos comandos em um arquivo, torna-lo executável e fim de papo. Vejamos a partir de agora, aprender instruções mais complexas que nos ajudarão na programação.

Exercícios

1. Fazer o programa para procura, pelo sobrenome, pessoas no arquivo telefones.

2. Fazer um programa para listar todas as pessoas de um determinado DDD.

3. Listar os usuários que estão "logados" há mais de um dia.

DICA: ver como funcionam os comandos *date* e *who*.

Capítulo 4

Liberdade Condicional !!

A partir de agora, você terá liberdade para executar *comandos condicionais*, sem os quais nenhuma linguagem de programação pode sobreviver. Eles possibilitam testar situações corriqueiras dentro dos programas, de forma a permitir tomada de decisões e contra-decisões cabíveis em cada caso, mudando o fluxo de execução das rotinas.

Todas as 134.532 linguagens de programação que eu conheço (vide *curriculum vitae* em anexo) se utilizam fartamente dos *comandos condicionais*. Porque o Shell seria diferente? Ele também usa estes comandos, porém de uma forma não muito ortodoxa. Enquanto as outras linguagens testam direto uma condição, o Shell testa o código de retorno do comando que o segue.

O código de retorno de uma instrução está associado à variável `$?`, de forma que sempre que um comando é executado com sucesso, `$?` é igual a *zero*, caso contrário o valor retornado será diferente de *zero*.

Exemplos:

```
$ ls -l quequeisso                                      Este nós conhecemos de outros carnavais...
-rw-r--r--   1 julio    dipao          416 Sep  3 10:53 quequeisso
$ echo $?                                               Que tal a execução da instrução anterior??
0                                                       Beleza...
$ ls -l nadadisso                                       Deste eu nunca ouvi falar...
nadadisso: No such file or directory
$ echo $?                                               Que tal a execução da instrução anterior??
2                                                       Zebra...
```

Nos exemplos acima, pudemos ver que, no primeiro caso, quando verificamos a existência de `quequeisso`, o código de retorno (`$?`) foi zero, indicando que o comando `ls` foi bem sucedido. O oposto acontece no caso do `nadadisso`.

4.1 - O Bom e Velho *if*

Qual é o programador que não conhece o comando `if`? Como todas as outras linguagens, o Shell também tem o seu comando `if` que na sua forma geral tem a seguinte sintaxe:

```
if  <comando>                    Se <comando> for bem sucedido ($? = 0)...
then                             Então...
    <comando1>                   Execute
    <comando2>                   Estes
    <...>                        Comandos...
else                             Senão ($? ≠ 0)...
    <comando3>                   Execute
    <comando4>                   Os outros
    <...>                        Comandos....
fi                               Fim do teste condicional.
```

Exemplos:

Vamos escrever um *script* que nos diga se uma determinada pessoa, que será passada por parâmetro, fez *login* no seu computador:

```
$ cat talogado
#
#  Verifica se determinado usuario esta "logado"
#

if  who | grep $1
then
    echo $1 esta logado
else
    echo $1 nao esta logado
fi
```

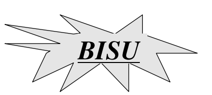
No trecho de programa acima o `if` executa o `who | grep` e testa o *código de retorno* do `grep`. Se for zero (o `grep` achou o usuário) o *bloco de comandos* "pendurado" abaixo do `then` serão executados. Caso contrário, estes comandos são saltados e, existindo um `else`, sua seqüência de instruções será então executada.

Observe o ambiente e a execução do `talogado`:

```
$ who                                        Quem esta logado?
enio       pts/0       Oct  9 22:07
ney        pts/2       Oct 10 14:42
ciro       pts/4       Oct 10 16:03
julio      pts/9       Oct  9 10:27
$ talogado enio
enio       pts/0       Oct  9 22:07          Esta linha indesejável é a saída do who...
julio esta logado                            Esta sim é a resposta que esperávamos
$ talogado ZeNinguem
ZeNinguem nao esta logado
```

Observe ainda os testes abaixo, que foram feitos no mesmo ambiente dos exemplos anteriores:

```
$ who | grep julio
julio      pts/9       Oct  9 10:27
$ echo $?
0                                            Comando grep executado com êxito
$ who | grep ZeNinguem
$ echo $?
1                                            Comando grep falhou...
```



Repare que a execução do *talogado enio* gerou uma linha correspondente à saída do `who`, que não traz proveito algum para o programa; diria até que o ideal seria que ela não aparecesse. Para tal, existe um *buraco negro* do UNIX que todas as coisas que lá são colocadas desaparecem sem deixar vestígio. Este *buraco negro*, chamado */dev/null,* é um arquivo especial do sistema, do tipo *device,* no qual todos estão aptos a gravar ou ler (neste último caso, ganhando imediatamente um `EOF`).

Exemplos:

Vamos redirecionar para o *buraco negro* a saída do `grep`, de forma a conseguir o resultado esperado no exercício anterior:

```
$ cat talogado
#
#  Verifica se determinado usuario esta "logado" - versao 2
#

if  who | grep $1 > /dev/null                    Incluí /dev/null redirecionando a saída do who
then
    echo $1 esta logado
else
    echo $1 nao esta logado
fi
$ talogado enio
enio esta logado                                  Não apareceu a linha indesejada
```

Homessa[4]!! Alcançamos o desejado. Esse tal de Shell é fogo...

Para saber se o conteúdo de uma variável é numérico ou não, poderíamos fazer:

```
if  expr $1 + 0 2> /dev/null                      Se $1 ≠ número, erro que irá para /dev/null
then
    echo Eh um numero
else
    echo Nao eh um numero
fi
```

Neste exemplo, somando zero ao conteúdo de uma variável, se ela for numérica seu conteúdo não será alterado, caso contrário, a instrução `expr` gerará um *código de retorno* diferente de zero que será capturado pelo `if`.

4.2 - Testando o *test*

A essa altura dos acontecimentos você irá me perguntar: ora, se o comando `if` só testa o conteúdo da variável `$?`, o que fazer para testar condições?

Para isso o Shell tem o comando `test`, que na sua forma geral obedece a seguinte sintaxe:

```
test <expressão>
```

[4] Para quem não sabe, *homessa* é uma interjeição de espanto formada por *homem* + *essa.* Shell também é cultura...

Sendo `<expressão>` a condição que se deseja testar. O comando `test` avalia `<expressão>` e se o resultado for verdadeiro gera o *código de retorno* (`$?`) zero, caso contrário será gerado um *código de retorno* diferente de zero.

- Esse negócio de *código de retorno* para lá, `$?` para cá está me cheirando a `if`...

Bom faro, realmente é normal a saída do comando `test` ser usada como entrada do comando `if`, como veremos abaixo:

Exemplos:

Temos uma rotina que recebe um parâmetro e analisa se é *S* para *S*im ou *N* para *N*ão. Abaixo, fragmento de sua codificação:

```
$ cat pedi
#
#  Testa a resposta a um pedido. Deve ser (S)im ou (N)ao
#

resp=$1
if  test $resp = N                               Saída do test é entrada do if
then
    echo Ela nao deixa...
else
    if  test $resp = S                           Um if dentro do outro dificulta a legibilidade
    then
        echo Oba, ela deixou!!!
    else
        echo Acho que ela esta na duvida.
    fi
fi
$ pedi S
Oba, ela deixou!!!
$ pedi N
Ela nao deixa...
$ pedi A
Acho que ela esta na duvida.
$ pedi                                           Xiii, esqueci de passar o parâmetro...
pedi[6]: test: argument expected                 Ué, o que houve?
pedi[10]: test: argument expected                Ídem...
Acho que ela esta na duvida.
```

Vamos fazer os mesmos testes direto no prompt do UNIX:

```
$ resp=N
$ test $resp = N
$ echo $?
0                                                Se fizéssemos um if o then seria executado
$ test $resp = S
$ echo $?
1                                                Se fizéssemos um if o else seria executado
```

Dos exemplos acima devemos realçar duas situações:

- Sempre que possível deve-se evitar um `if` dentro de outro, de forma a não dificultar a legibilidade e a lógica do programa. O UNIX permite isto com o uso de `elif` substituindo o `else`...`if`..., além de otimizar a execução.

- Quando estivermos testando uma variável, seu nome deve sempre estar entre aspas para evitar erros, no caso desta variável estar vazia.

A rotina acima poderia (e deveria) ser escrita da seguinte forma:

```
$ cat pedi
#
#  Testa a resposta a um pedido. Deve ser (S)im ou (N)ao - Versao 2
#

resp=$1
if  test "$resp" = N                        Repare que a variável foi colocada entre aspas
then
    echo Ela nao deixa...
elif  test "$resp" = S                      Repare uso do elif e a variável entre aspas
then
    echo Oba, ela deixou!!!
else
    echo Acho que ela esta na duvida.
fi
$ pedi                                      Não foi informado parâmetro.
Acho que ela esta na duvida.                Não indicou erro de execução
```

Mas, para o programa ficar supimpa, que tal fazê-lo da forma abaixo:

```
$ cat pedi
#
#  Testa a resposta a um pedido. Deve ser (S)im ou (N)ao - Versao 3
#

if  test $# = 0                             Estou recebendo parâmetro ou não?
then
    echo Faltou informar a resposta         Não recebi o tal do parâmetro.
    exit 1                                  Encerro o programa com código de retorno ≠0
fi
resp=$1
if  test "$resp" = N
then
    echo Ela nao deixa...
elif  test "$resp" = S
then
    echo Oba, ela deixou!!!
else
    echo Acho que ela esta na duvida.
fi
$ pedi
Faltou informar a resposta
```

Veremos a seguir as principais opções para testes de condição de arquivos, usando-se o comando *test.*

Opções	Verdadeiro ($?=0) se arquivo existe e:
-r arquivo	tem permissão de leitura.
-w arquivo	tem permissão de gravação.
-x arquivo	tem permissão de execução.
-f arquivo	é um arquivo regular.
-d arquivo	é um diretório.
-u arquivo	seu bit set-user-ID está ativo
-g arquivo	seu bit set-group-ID está ativo
-k arquivo	seu sticky bit está ativo
-s arquivo	seu tamanho é maior que zero

Exemplos:

```
$ ls -l ta* que*                                                  listando conteúdo do diretório
-rw-r--r--   1 julio    dipao          416 Sep  3 10:53 quequeisso
----------   1 julio    dipao           86 Oct 14 14:25 tafechado
-rwxr--r--   1 julio    dipao          160 Oct 10 18:11 talogado
-rw-r--r--   1 julio    dipao            0 Oct 14 14:35 tavazio
$ test -f tafechado                                               tafechado é um arquivo?
$ echo $?
0                                                                 Condição verdadeira. O arquivo existe
$ test -f talogado                                                Existe arquivo chamado talogado?
$ echo $?
0                                                                 Condição verdadeira. Existe o arquivo talogado
$ test -r tafechado                                               Tenho direito de leitura sobre tafechado ?
$ echo $?
1                                                                 Condição falsa. Não pode ler tafechado
$ test -r talogado                                                Tenho direito de leitura sobre talogado?
$ echo $?
0                                                                 Condição verdadeira. Pode ler talogado
$ test -x talogado                                                Posso executar talogado?
$ echo $?
0                                                                 Condição verdadeira. talogado é executável
$ test -x quequeisso                                              Posso executar quequeisso?
$ echo $?
1                                                                 Condição falsa. Não pode executar quequeisso
$ test -s tavazio                                                 tavazio existe e tem tamanho maior que zero?
$ echo $?
1                                                                 Condição falsa. tavazio tem comprimento zero
$ test -s tafechado                                               tafechado existe e tamanho é maior que zero?
$ echo $?
0                                                                 Cond. verdadeira. tafechado é maior que zero
```

Vejamos, agora, como se usa o comando *test* com cadeia de caracteres:

opções	**É verdadeiro ($?=0) se:**
-z cad_1	o tamanho de cad_1 é zero.
-n cad_1	o tamanho da cadeia cad_1 é diferente de zero.
cad_1 **=** cad_2	as cadeias cad_1 e cad_2 são idênticas
cad_1	cad_1 é uma cadeia não nula.

Exemplos:

```
$ nada=                                                           Variável nada criada com valor nulo
$ test -z "$nada"                                                 Variável nada não existe ou esta vazia?
$ echo $?
0                                                                 Condição verdadeira
$ test -n "$nada"                                                 Variável nada existe e não esta vazia?
$ echo $?
1                                                                 Condição falsa
$ test $nada                                                      Variável nada não esta vazia?
$ echo $?
1                                                                 Condição falsa
```

```
$ nada=algo                                   Atribui valor a nada
$ echo $nada                                  Só para mostrar que agora nada vale algo
algo
$ test $nada                                  Variável nada não esta vazia?
$ echo $?
0                                             Condição verdadeira
```

CUIDADO!!! Olha só que "lama" que se pode fazer sem querer:

```
$ echo $igual                                 Liste o conteúdo da variável igual
=                                             Na variável igual esta armazenado o caracter =
$ test -z "$igual"                            Variável igual não existe ou esta vazia?
ksh: test: argument expected                  ZEBRA!!!
```

O operador = tem maior precedência que o operador -z, então o que foi executado seria para testar se a cadeia -z = . Ora, = a quê? Como não foi explicitado nada do lado direito do sinal de igual, o Shell avisou que estava faltando um argumento.

Este teste deveria ter sido feito mais ou menos assim:

```
test :"$igual" = :
```

O *test* acima retornará verdadeiro se a variável *igual* contiver valor nulo, caso contrário o retorno será falso.

Usamos o comando *test* com inteiros usando os seguintes operadores:

Operando	É verdadeiro ($?=0) se:	Significado:
int_1 **-eq** int_2	int_1 igual a int_2	equal to
int_1 **-ne** int_2	int_1 diferente de int_2	not equal to
int_1 **-gt** int_2	int_1 maior que int_2	greater than
int_1 **-ge** int_2	int_1 maior ou igual a int_2	greater or equal
int_1 **-lt** int_2	int_1 menor que int_2	less than
int_1 -**le** int_2	int_1 menor ou igual a int_2	less or equal

CUIDADO!!! Lembre-se que o Shell diferentemente da maioria das outras linguagens, não distingue tipos de dados armazenados nas variáveis. Portanto, que tipo de teste que deve ser feito? Você decide (mas por favor não conte com a maioria da TV Globo...). Exemplificando sempre fica mais fácil:

Exemplos:

```
$ echo $qqcoisa
010
$ test "$qqcoisa" -eq 10                      Fiz um teste entre inteiros...
$ echo $?
0                                             Retornou verdadeiro
$ test "$qqcoisa" = 10                        Fiz comparação entre cadeias de caracteres...
$ echo $?
1                                             Retornou falso
```

Dos exemplos acima podemos inferir que, caso o objetivo do teste fosse identificar se o conteúdo da variável era exatamente igual a um valor, este teste deveria ser executado com operandos característicos de cadeias de caracteres. Por outro lado, se seu desejo fosse testar a semelhança entre o valor e o conteúdo da variável, o operando deveria ser numérico.

Voltaremos aos exemplos daqui a pouco com o *test* de roupa nova.

4.2.1 - O *test* de Roupa Nova

Cá entre nós, o comando `if test` ... fica muito esquisito, não fica? O `if` com este formato foge ao usual de todas as linguagens. Para minorar este inconveniente e dar mais legibilidade ao programa, o comando `test` pode ser substituído por um par de colchetes (`[ ]`) abraçando o argumento a ser testado. Então, na sua forma mais geral o comando `test` que é escrito na seguinte forma:

```
test <expressão>
```

Pode ser, simplesmente, escrito:

```
[¦ <expressão>¦]
```

Note que os espaços colados nos colchetes estão hachurados. Isto é para mostrar que naqueles pontos é obrigatória a presença de espaços em branco.

Exemplos:

Então, o último exemplo:

```
$ test "$qqcoisa" = 10                    Fiz comparação entre cadeias de caracteres...
$ echo $?
1                                         Retornou falso
```

Poderia ter sido escrito com a seguinte roupagem:

```
$ [ "$qqcoisa" = 10 ]                     Fiz comparação entre cadeias de caracteres...
$ echo $?
1                                         Retornou falso
```

E se estivesse no bojo de um programa seria:

```
if  [ "$qqcoisa" = 10 ]
then
    echo "A variavel qqcoisa eh exatamente igual a 10"
fi
```

Que aumentaria enormemente a legibilidade.

A partir deste ponto, esta será na grande maioria dos casos, a forma geral de uso deste comando ao longo deste livro, e espero que também assim seja nos programas que você desenvolverá.

Diferentemente do padrão Shell, dentro de um `test`, o uso de parênteses indica precedência que podem ser usados ocasionalmente, porém não se esqueça que devem vir

protegidos, para que não sofram interpretação do Shell, e, sempre, devem estar isolados entre dois espaços em branco.

4.2.2 - Se Alguém Disser Que Eu Disse, Eu Nego...

Mas nego usando um ponto de exclamação (ou espantação como o Bolpa prefere), porque é assim que diversas linguagens de programação, ai incluído o Shell, representam o seu operador lógico de negação, como até já havíamos visto no comando *sed* descrito na seção 2.1

Exemplos:

Vamos fazer um programa bastante útil, que serve para salvar um arquivo no seu formato original antes de editá-lo pelo `vi`, de forma a poder recuperá-lo incólume, no caso de alguma barbeiragem no uso do editor (o que, cá entre nós, sabemos que é a maior moleza!).

```
$ cat vira
#
#  vira - vi resguardando arquivo anterior
#

if  [ "$#" -ne 1 ]
then
    echo "Erro -> Uso: vira <arquivo>"
    exit 1
fi

Arq=$1
if  [ ! -f "$Arq" ]
then
    vi $Arq
    exit 0
fi

if  [ ! -w "$Arq" ]
then
    echo "Nao perca seu tempo, você nao conseguira sobregravar $Arq"
    exit 2
fi

cp $Arq $Arq~
vi $Arq
exit 0
```

4.2.3 - Não Confunda *and* Com *The End*

O primeiro é um operador lógico, o segundo é um fim "holywoodiano", freqüentemente ilógico. Como nosso negócio é a lógica esmiuçaremos o primeiro e ignoraremos o segundo. Está combinado?

O operador *and* ou *e*, representado em Shell por **-a**, serve para testar duas ou mais condições, dando resultado verdadeiro somente se <u>todas</u> as condições testadas forem verdadeiras.

Como em um recenseamento em nível nacional não existe chance de mandar o recenseador de volta ao domicílio em que ele levantou um dado duvidoso, os programas de entrada manual de dados feitos para sistemas de recenseamento geralmente não têm críticas, o que pode gerar grandes ansiedades. Supondo-se, mesmo que inocentemente, que só existem dois sexos, vejamos o seguinte trecho de programa de entrada de dados do recenseamento:

```
if  [ "$sexo" = 1 ]
then
    Some 1 ao contador de homens
else                                        Qualquer coisa ≠ 1 será considerado mulher
    Some 1 ao contador de mulheres
fi
```

Por isso, conheço vários homens ansiosos, achando que outros estão com duas cotas de mulheres já que ele tem somente aquela antiiiiiga em casa. Não é nada disso gente, caso o dado tivesse sido criticado *a priori*, esta proporção de mulher para homem cairia bastante. Vejamos como:

```
if  [ "$sexo" != 1 -a "$sexo" != 2 ]        Se sexo ≠ 1 e sexo ≠ 2...
then
    echo "Sexo Invalido"
fi
```

Agora sim, podemos inserir a rotina de incremento dos contadores, descrita acima.

4.2.4 - *or* ou *ou* Disse o Cão Afônico

O operador lógico *or* ou *ou*, representado em Shell por **-o**, serve para testar duas ou mais condições, dando resultado verdadeiro se <u>pelo menos uma</u> dentre as condições testadas for verdadeira.

Observe que a crítica de sexo escrita no ítem anterior, também poderia ter sido feita, e no meu entender de forma mais fácil, da forma abaixo:

```
if  [ "$sexo" -lt 1 -o "$sexo" -gt 2 ]      Se sexo menor que 1 ou sexo maior que 2
then
    echo "Sexo Invalido"
fi
```

CUIDADO!! Preste muita atenção com a mistura explosiva dos três operadores lógicos descritos acima. Tenha sempre em mente que <u>*não ou*</u> vale <u>*e*</u> e da mesma forma <u>*não e*</u> vale <u>*ou*</u>.

Qual é a forma correta de fazer a pergunta: *se sexo não igual a 1* ***e*** *sexo não igual a 2* ou *se sexo não igual a 1* **ou** *sexo não igual a 2* ????

O operador **-a** tem precedência sobre o operador **-o** desta forma, se quisermos priorizar a execução do *or* em detrimento ao *and*, devemos priorizar a expressão do *or* com o uso de parênteses.

Se existisse um comando *if* construído da seguinte forma:

```
if [ $sexo -eq 1 -o $sexo -eq 2 -a $nome = joao -o $nome = maria ]
```

A primeira expressão a ser resolvida seria:

```
$sexo -eq 2 and $nome = joao
```

Bagunçando totalmente a lógica, porque o *and* tem prioridade sobre o *or*. Para escrevermos de forma correta deveríamos:

```
if [ \( $sexo -eq  1 -o $sexo -eq 2 \) -a \          A última contrabarra informa que linha continua
     \( $nome = joao -o $nome = maria \) ]          As outras inibem interpretação dos parênteses
```

4.3 - Disfarçando de *if*

Existem em Shell dois operadores, que podem ser usados executando tarefas do comando `if`. Apesar de, geralmente, gerarem comandos mais leves e mais otimizados, o programa perde em legibilidade. Desta forma, creio só ser vantagem usá-los em linhas de programa repetidas com muita freqüência. Estes operadores são:

- **&&** (*and* ou *e* lógico)

  De acordo com a "Tabela Verdade" para que um teste tipo `<cond1>` *e* `<cond2>` seja verdadeiro, é necessário que ambas as condições sejam verdadeiras. Assim, se `<cond1>` for verdadeira, `<cond2>` obrigatoriamente será executada para testar se é verdadeira ou falsa. Veja os exemplos a seguir:

  Exemplos:

```
$ a="Eu sou a variavel a"
$ b="Sou a outra"
$ echo $a && echo $b
Eu sou a variavel a                    1ª instrução foi executada com sucesso...
Sou a outra                            Portanto a 2ª também foi executada
$ ls -l xxx && echo $a
xxx: No such file or directory         Já que 1ª instrução falhou, 2ª não foi executada
```

Agora um exemplo com mais utilidade:

```
[ "$DDD" = 084 ] && Cidade=Natal       Faço Cidade=Natal se DDD for igual a 084
```

- **||** (*or* ou *ou* lógico)

  De acordo com a "Tabela Verdade", para que um teste tipo `<cond1>` *ou* `<cond2>` seja verdadeiro, é necessário que qualquer uma das condições seja verdadeira. Assim, `se` `<cond1>` for verdadeira, `<cond2>` não será executada, pois certamente o resultado final será verdadeiro. Veja os exemplos abaixo:

```
$ echo $a || echo $b
Eu sou a variavel a
$ ls -l xxx || echo $a
```

```
xxx: No such file or directory
Eu sou a variavel a
```

Agora um exemplo com mais (um pouco) de utilidade:

```
[ "$DDD" = 084 ] && Cidade=Natal || Cidade=Rio        Se DDD ≠ 084, faço Cidade=Rio
```

4.4 - O Caso Que o *case* Casa Melhor

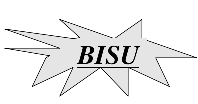
O comando *case* deve ser usado caso a quantidade de `ifs` sucessivos seja maior que três, pois além de agilizar a execução, aumenta a legibilidade e diminui o tamanho do código.

Este comando permite verificar um padrão contra diversos outros, e executa diversos comandos quando o critério avaliado é alcançado.

A forma geral do comando é a seguinte:

```
case valor in
padr1)                                                1º padrão de comparação
     <comando1>
     <...>
     <comandon>
     ;;                                               Fim do 1º bloco de comandos
padr2)
     <comando1>
     <...>
     <comandon>
     ;;
<...>
padrn)                                                Enésimo padrão de comparação
     <comando1>
     <...>
     <comandon>
     ;;                                               Fim do enésimo bloco de comandos
esac                                                  fim do case
```

O `valor` é comparado a cada um dos *padrões* ($padr_1$, $padr_2$, ..., $padr_n$), até que satisfaça a um deles, quando, então, passará a executar os *comandos* subseqüentes até que dois ponto-e-vírgula (*;;*) sucessivos sejam encontrados.



Caso o valor informado não satisfaça nenhum dos padrões, o comando `case` não reportará erro, e, simplesmente não entrará em *bloco de comando* algum, o que provavelmente redundará em "furo" de lógica no decorrer do programa. Para evitar isto, é freqüente colocar * como sendo o último padrão de comparação. Assim, qualquer que seja o valor, este padrão será satisfeito.

Exemplos:

Vamos mostrar um programa que recebendo um caracter como parâmetro identifica o seu tipo:

```
$ cat testchar
#
#  Testa de que tipo eh um caracter recebido por parametro
#
```

```
#####              Teste da Quantidade de Parametros              #####
erro=0
if  [ "$#" -ne 1 ]
then
    echo "Erro -> Uso: testchar caracter"
    erro=1
fi

#####     Testa se o 1o. parametro tem o tamanho de um caracter     #####
case $1 in
    ?)  ;;                                       Se tiver somente um caracter ...
    *)  echo "Erro -> Parametro passado tem mais de um caracter"
        erro=2
        ;;
esac

#####  Se houve erro o programa termina, passando o codigo do erro  #####
if  [ "$erro" -ne 0 ]
then
    exit $erro
fi

case $1 in
[a-z])  echo Letra Minuscula
        ;;
[A-Z])  echo Letra Maiuscula
        ;;
[0-9])  echo Numero
        ;;
    *)  echo Caracter Especial                   Veja o asterisco sendo usado como "o resto"
        ;;                                       Os ponto-e-vírgula antes do esac são opcionais
esac
```

Repare a rotina em que se testa o tamanho do parâmetro recebido: não se esqueça que o *ponto de interrogação* (*?*) substitui qualquer caracter naquela posição, isto é, qualquer caracter, desde que seja único, será aceito pela rotina, caso contrário (*), será considerado errado.

```
$ testchar
Erro -> Uso: testchar caracter
Erro -> Parametro passado tem mais de um caracter
$ testchar aaa
Erro -> Parametro passado tem mais de um caracter
$ testchar A B
Erro -> Uso: testchar caracter
$ testchar ab ab
Erro -> Uso: testchar caracter
Erro -> Parametro passado tem mais de um caracter
$ testchar 1
Numero
$ testchar a
Letra Minuscula
$ testchar A
Letra Maiuscula
$ testchar %
Caracter Especial
```

Exercícios

1. Fazer um programa que imprima horas e minutos no formato 12 horas (ex. 7:00 am/pm)

2. Escreva um programa que execute o comando `sed` tendo o 1º parâmetro como argumento deste comando e o 2º como o arquivo a ser alterado. Se e somente se o `sed` for bem sucedido, o arquivo original deve ser trocado pelo arquivo modificado.

Capítulo 5

De Lupa no Loop

Todo programa tem uma entrada do(s) dado(s), um processamento e uma saída. Este processamento em 110% das vezes é feito em *ciclos* que normalmente chamamos de *loop*. Neste capítulo, veremos em *"close"* (e não é a Roberta) como se deve trabalhar os comandos que produzem este efeito.

Antes de mais nada, gostaria de passar o conceito de *bloco de programa*. Chamamos *bloco de programa* ou *bloco de instruções* o agrupamento de instruções compreendido entre um sinal de abre chaves (*{*) e um de fecha chaves (*}*). Repare no fragmento de programa abaixo:

```
[ -d "$Diretorio" ] ||
{
    mkdir $Diretorio
    cd $Diretorio
}
```

O operador lógico **||** obriga a execução da instrução seguinte, caso a anterior tenha sido mal sucedida. Então, no caso de não existir o diretório contido na variável `Diretorio`, o mesmo será criado e, então, faremos um `cd` para dentro dele.

Um bloco de programas também pode ser aberto por um `do`, por um `if`, por um `else` ou por um `case` e fechado por um `done`, um `else`, um `fi`, ou por um `esac`.

5.1 - O Forró do *for*

Se em português disséssemos:

```
para var em valor1 valor2 ... valorn
faça
    <comando1>
    <comando2>
    <...>
    <comandom>
feito
```

Entenderíamos que a variável *var* assumiria os valores $valor_1$, $valor_2$, ..., $valor_n$ durante as *n* execuções dos comandos $comando_1$, $comando_2$, ..., $comando_m$. Pois é meu amigo, a sintaxe original do comando `for` comporta-se, exatamente, da mesma maneira. Assim, vertendo-se para o ingles, fica:

```
for var in valor1 valor2 ... valorn
do
    <comando1>
    <comando2>
    <...>
    <comandom>
done
```

Que funciona da maneira descrita acima.

Vejamos alguns exemplos do uso do comando `for`:

Exemplos:

Para escrever os múltiplos de 11 a partir de 11 até 99 basta:

```
$ cat bronze
#
#  Lista multiplos de onze a partir de 11 ate 99
#

for i in 1 2 3 4 5 6 7 8 9
do
    echo  $i$i
done

$ bronze
11
22
33
44
55
66
77
88
99
```

Nesta série de exemplos, veremos uma grande quantidade de formas diferentes de executar a mesma tarefa:

```
$ ls param*
param1  param2  param3  param4  param5
$ for i in param1 param2 param3 param4 param5      O valor de i foi 100% especificado
> do
>     ls $i
> done
param1
param2
param3
param4
param5
$ for i in param[1-5]                              Fizemos i=param e variamos seu final de 1 a 5
> do
>     ls $i
> done
param1
param2
param3
param4
param5
$ for i in `ls param*`                             O ls gera nome dos arq. separados por branco
> do
>     ls $i
> done
```

```
param1
param2
param3
param4
param5
```

Repare que os comandos que envolvem *blocos de instruções,* quando são diretamente digitados via terminal, só terminam quando o UNIX vê o fechamento destes blocos, mandando um prompt secundário (**>**) para cada nova linha.

No programa abaixo, que é uma continuação dos exemplos das páginas 3.3, 3.4 e 3.5, devemos notar que :

- Na linha do `for` não existe o `in`. Neste caso, a variável do `for` tomará o valor dos parâmetros passados.
- Na discussão sobre passagem de parâmetros (seção 3.3), falamos que só usando alguns artifícios conseguiríamos recuperar parâmetros de ordem superior a 9. Pois bem, o `for` é o mais importante destes processos. Repare que no exemplo abaixo os onze parâmetros serão listados.

```
$ cat param6
echo O programa $0 Recebeu $# Parametros
echo "Que sao: \c"                                   O \c é para continuar listagem na mesma linha
for i
do
    echo "$i \c"
done
$ param6 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11.
O programa param6 Recebeu 11 Parametros
Que sao: 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. $       O prompt ($) após o 11. é devido ao último \c
```

Para o exemplo seguinte, servem as observações acima sobre a não presença do `in` e sobre a quantidade de parâmetros superior a nove. Leia atentamente este exemplo; é muito útil. Para ser utilizado, basta implantar no seu *.profile* uma rotina de remoção dos arquivos por ele gerados com data de criação inferior a um período previamente estipulado (sugiro quatro dias) e, no mesmo *.profile,* pode-se colocar também uma linha:

```
alias rm=erreme
```

para coloca-lo no lugar do comando `rm`.

```
#
#  Salvando Copia de Arquivo Antes de Remove-lo
#

if  [ $# -eq 0 ]                                  Deve ter um ou mais arquivos p/ remover
then
    echo "Erro -> Uso: erreme arq [arq] ... [arq]"
    echo "        O uso de metacaracteres eh permitido. Ex. erreme arq*"
    exit 1
fi

MeuDir="/tmp/$LOGNAME"                            Variável do sistema. Contém o nome do usuário
if  [ ! -d $MeuDir ]                              Se não existir o meu diretório sob o /tmp...
then
    mkdir $MeuDir                                 Vou cria-lo.
fi

if  [ ! -w $MeuDir ]                              Se não posso gravar no diretório...
then
    echo Impossivel salvar arquivos em $MeuDir. Mude permissao e tente
novamente...
```

```
    exit 2
fi

Erro=0                                          Variável para indicar o código de retorno do prg
for Arq
do
    if  [ ! -f $Arq ]                           Se este arquivo não existir...
    then
        echo $Arq nao existe.
        Erro=3
        continue                                Volta para o comando for
    fi

    DirOrig=`dirname $Arq`                      Comando dirname informa nome do diretório
    if  [ ! -w $DirOrig ]                       Verifica permissão de gravação no diretório
    then
        echo Sem permissao de remover no diretorio de $Arq
        Erro=4
        continue                                Volta para o comando for
    fi

    if  [ "$DirOrig" = "$MeuDir" ]              Se diretório que vou remover=diretório de cópia
    then
        echo $Arq ficara sem copia de seguranca
        rm -i $Arq                              Pergunto se deseja remover
        [ -f $Arq ] || echo $Arq removido
        continue
    fi

    mv $Arq $MeuDir                             Salvo e deleto.
    echo $Arq removido
done
exit $Erro                                      Passo Nº do erro para o código de retorno
```

Existe uma variável intrínseca ao sistema chamada de **IFS** (***I****nter* ***F****ield* ***S****eparator* - Separador entre os campos) que como o nome diz, trata-se do separador *default* entre dois campos. Assume-se que este *default* seja espaço, uma vez que é assim que ele se comporta. Senão vejamos:

```
$ echo $IFS                                     Nada aparece quando listamos a variável

$ echo "$IFS" | hd                              hd lista conteúdo da variável em hexadecimal
0000    20 09 0a 0a                             ...
0004
```

No entanto, quando passamos o conteúdo da variável para o comando *hd* (que gera a listagem em hexadecimal) vemos que é composto por três *bytes* que significam:

`20` - Espaço

`09` - <TAB>

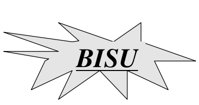
`0a` - Line feed

O último `0a` foi produzido pela tecla <ENTER> no fim da linha (após o `hd`). Veja abaixo a seqüência de instruções dadas diretamente a partir do teclado:

```
$ grep fernando /etc/passwd                     Procuro o usuário fernando em /etc/passwd
fernando:x:110:1002:Fernando Guimaraes:/dsv/usr/fernando:/usr/bin/ksh
$ OldIFS=$IFS                                   Salvo o IFS antigo
$ IFS=:                                         O novo IFS passa a ser dois-pontos (:)
$ linha=`grep fernando /etc/passwd`             Salvo na variável linha o registro do Fernando
$ echo $linha                                   Repare que não aparece mais o dois-pontos(:)
fernando x 110 1002 Fernando Guimaraes /dsv/usr/fernando /usr/bin/ksh
$ for tasco in `grep fernando /etc/passwd`
```

```
> do
>     echo $tasco
> done
fernando
x
110
1002
Fernando Guimaraes                              Aqui o espaço não separou o campo
/dsv/usr/fernando
/usr/bin/ksh
$ IFS=$OldIFS                                   Tudo como dantes no quartel de Abrantes
```

Do exercício anterior, notamos que o registro cujo separador é *dois-pontos* (`:`), após a alteração do `IFS` foi gerado na variável `linha` sem que o mesmo aparecesse, **apesar de ainda constar do seu conteúdo** como pode ser demonstrado pelo fato da cadeia `Fernando Guimaraes` permanecer unida.



Quando for alterar o conteúdo da variável `IFS`, não se esqueça de salvá-lo antes, uma vez que só conheço duas formas de restaurar seu conteúdo original: salvá-lo e recuperá-lo ou desconectar-se e conectar-se novamente.

5.2 - Perguntaram ao Mineiro: o Que é *while*? *while* é *while*, Uai!

Creio ser o `while` o comando de *loop* mais usado em programação Shell, portanto, o mais importante deste capítulo. Se fôssemos escrevê-lo em português sua sintaxe seria mais ou menos assim:

```
enquanto <condição>
faça
    <comando1>
    <comando2>
    <...>
    <comandon>
feito
```

Onde `condição` é executada e seu código de retorno é testado, caso seja igual a zero, só então $comando_1$, $comando_2$, ... $comando_n$ (o *bloco de comandos* entre o `faça` e o `feito`) são executados e ao encontrar o `feito`, reinicia-se todo o ciclo novamente. Isto continua até que a *condição* devolva um código de retorno diferente de zero, quando, então, a execução do programa salta para a instrução que segue o `feito`.

Mostrando agora a sintaxe deste comando em Shell, como manda o figurino e aproveitando as definições acima:

```
while <condição>
do
    <comando1>
    <comando2>
    <...>
    <comandon>
done
```

Um exemplo fala melhor que mil palavras:

Exemplos:

Vamos refazer o programa `bronze` usando o comando `while` no lugar do `for`:

```
$ cat bronze
#
#  Lista multiplos de 11 a partir de 11 ate 99 - Versao 2
#

i=1
while [ $i -le 9 ]                           Repare que a frente do while temos um test
do
    echo  $i$i
    i=`expr $i + 1`
done
$ bronze
11
22
33
44
55
66
77
88
99
```

Agora cabe a você estipular o melhor processo para desenvolver o `bronze`. Eu, provavelmente, faria usando o `while`.

O Ciro é meu vizinho e trabalha em uma sala distante da minha. Se eu soubesse o momento em que ele se desconecta do UNIX, ligaria para ele lembrando a minha carona. Para isso, eu faço assim:

```
while who | grep ciro > /dev/null
do
    sleep 30                                 Espera 30 seg. fazendo absolutamente nada
done
echo "O Ciro DEU EXIT. NAO HESITE, ligue logo para ele."
```

Enquanto o Ciro estiver "logado", o "pipeline" `who | grep ciro` gerará código de retorno igual a zero levando o programa a dormir por trinta segundos. Assim que o Ciro se desconectar, na próxima vez que a condição for analisada, resultará um código de retorno diferente de zero, de forma que o fluxo de execução saia do *loop*, gerando a mensagem de alerta.

Obviamente este programa bloquearia o terminal não permitindo executar mais nada, até que o Ciro se desconectasse. Se quiséssemos liberar o terminal, deveríamos executar o programa em *background*, o que se faz colocando um **&** imediatamente após o seu nome, quando se comanda a sua execução. Supondo que o nome deste programa seja *cirobye*, deveríamos fazer:

```
$ cirobye &                                  "Startei" o programa em background
2469                                         Shell devolveu PID
$                                            Recebi o prompt para continuar trabalhando
```

5.3 - O *until* Não Leva Um ~ Mas é Útil.

Se plantarmos uma bananeira vendo o mundo de cabeça para baixo (ou de ponta cabeça, como preferem os paulistas), quando olharmos para o `while` estaremos vendo o `until`. Entendeu? Nem eu.

O blablablá acima foi para mostrar que o `until` é igual ao `while`, porém ao contrário. Se este fosse um livro de física, diríamos com a mesma direção e sentidos opostos.

Como já fizemos em diversas oportunidades, vejamos qual seria a sua sintaxe se o Shell entendesse português:

```
até que <condição>
faça
    <comando1>
    <comando2>
    <...>
    <comandon>
feito
```

Onde *condição* é executada e seu *código de retorno* é testado, caso ***não*** seja igual a zero, $comando_1$, $comando_2$, ... $comando_n$ (os comandos entre o `faça` e o `feito`) são executados e ao encontrar o `feito`, reinicia-se todo o ciclo novamente. Isto continua até a *`condição`* devolver um *código de retorno* igual a zero, quando a execução do programa salta para a instrução que segue o `feito`.

O formato geral desta sintaxe em Shell, é o seguinte:

```
until <condição>
do
    <comando1>
    <comando2>
    <...>
    <comandon>
done
```

O comando `until` é particularmente útil quando o nosso programa necessita esperar que um determinado evento ocorra.

Exemplos:

Para provarmos que o `until` é o `while` ao contrário, vejamos um exemplo que seja o inverso do anterior: vamos fazer um programa que nos avise quando uma determinada pessoa se conectou. Ainda ao contrário, esta pessoa não é o Ciro mas, sim a Bárbara, (que como diz o nome, é realmente bárbara...). Eu já havia feito o programa *talogado (descrito na página 4.2)* que eu poderia usar para saber quando a Bárbara se conectasse. O *talogado* era assim:

```
$ cat talogado
#
#  Verifica se determinado usuario esta "logado" - versao 2
#

if  who | grep $1 > /dev/null
then
    echo $1 esta logado
else
    echo $1 nao esta logado
fi
```

Ora, para saber se a Bárbara já havia se conectado eu tinha que fazer:

```
$ talogado barbara
barbara nao esta logado
```

E até que ela se conectasse eu tinha que executar este comando diversas vezes (já que minha ansiedade me obrigava a executar o programa a cada minuto). Resolvi então desenvolver um programa específico para estes deliciosos momentos matinais, que chamei de *bdb* (Bom Dia Bárbara em código).

```
$ cat bdb
#
#  Avisa que determinada usuaria se conectou
#

if  [ "$#" -ne 1 ]
then
    echo "Erro -> Uso: bdb usuario"
    exit 1
fi
until who | grep $1 > /dev/null
do
    sleep 30
done
echo $1 se logou
```

Esta forma de programa está quase boa, mas o programa iria prender a tela do meu terminal até que a pessoa ansiosamente aguardada se conectasse.

Ora, isto é muito fácil! Basta executar o programa em *background*. E foi o que fiz, até que um dia estava no *vi* editando um *script* quando ela se logou e a mensagem vindo das profundezas do *background* estragou a minha edição. Outra vez foi pior ainda, estava listando na tela um arquivo grande, a mensagem do programa saiu no meio deste `cat` e com o *scroll* da tela eu não vi e não telefonei para dar-lhe bom dia.

A partir de então resolvi reescrever o programa dando uma opção de mandar o aviso para onde escolhesse: direto para a tela ou para o meu *mail*, isto é, se na execução do programa fosse passado o parâmetro *-m*, o resultado iria para o mail, caso contrário, para a tela.

Veja só o que fiz:

```
$ cat bdb
#
#  Avisa que determinada usuaria se conectou - versao 2
#

MandaMail=                                 Inicializa variável vazia
if  [ "$1" = -m ]                          A seguir verifico se foi passado o parâmetro -m
then
    MandaMail=1
    shift                                  Já sinalizei com MandaMail. Jogo fora o -m
fi
if  [ "$#" -ne 1 ]                         Vamos verificar se recebi o nome do usuário
then
    echo "Erro -> Uso: bdb [-m] usuario"
    echo "Usa-se -m para avisar via mail"
    exit 1
fi
until who | grep $1 > /dev/null            Até que seu nome apareça no comando who...
do
    sleep 30                               Aguarde 30 segundos
done
if  [ "$MandaMail" ]                       Se a variável MandaMail não estiver vazia...
then
    echo "$1 se logou" | mail julio
```

```
else
    echo "$1 se logou"
fi
```

No exemplo acima, primeiro testamos se a opção `-m` foi usada. Se foi, colocamos a variável `MandaMail` igual a um e fizemos um `shift` para jogar fora o primeiro argumento (movendo o nome do usuário para `$1` e decrementando `$#`). O programa então prossegue como na versão anterior, até que sai do ciclo de espera, quando faz um teste para verificar se a opção `-m` foi utilizada.

```
$ bdb barbara -m
Erro -> Uso: bdb [-m] usuario
•            -m para mandar aviso via mail
$ bdb -m barbara &                                Mandei executar em background
[1]     28808                                     recebi o PID do programa
...                                               Continuo o meu trabalho...
you have mail                                     Sistema me avisa que chegou um mail
```

O programa anterior esta genérico quanto ao usuário que esperamos conectar-se pois, sempre o passamos por parâmetro, porém, só eu posso usá-lo porque o mail será endereçado para mim. Se quisermos que o mail seja enviado para qualquer pessoa que esteja executando o programa, devemos fazer:

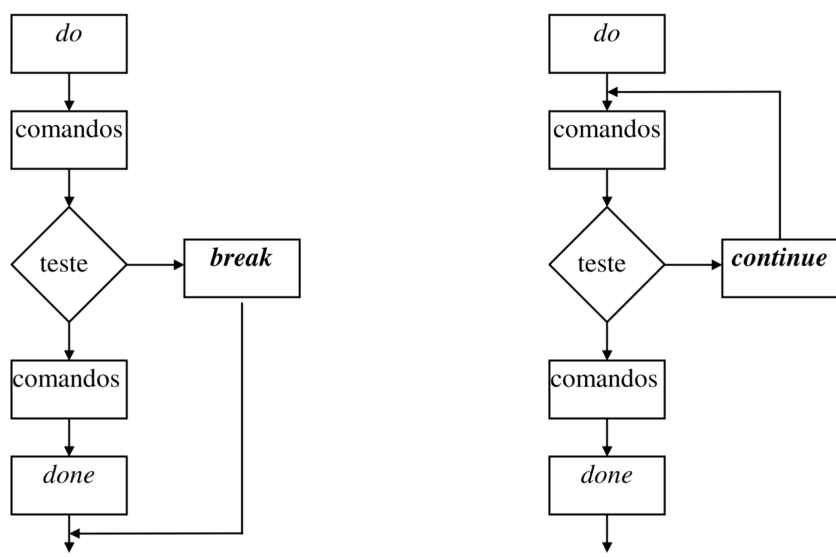
```
$ cat bdb
#
#  Avisa que determinada usuaria se conectou - versao 3
#

MandaMail=
if  [ "$1" = -m ]
then
    Eu=`who am i | cut -f1 -d" "`                 Variável Eu recebe  nome do usuário  ativo.
    MandaMail=1
    shift
fi
if  [ "$#" -ne 1 ]
then
    echo "Erro -> Uso: bdb [-m] usuario"
    echo "Usa-se -m para avisar via mail"
    exit 1
fi
until who | grep $1 > /dev/null
do
    sleep 30
done
if  [ "$MandaMail" ]
then
    echo "$1 se logou" | mail $Eu                 Mail vai para quem esta executando programa
else
    echo "$1 se logou"
fi
```

A linguagem de programação do Shell é muito rica e cheia de recursos, isto por vezes faz com que um programa que idealizamos, após um pouco de reflexão, torne-se um elefante branco. Poderíamos ter obtido o mesmo resultado acima, sem fazer todas estas alterações se executássemos o programa, em sua primeira versão, assim:

```
$ bdb barbara | mail <nome destinatário> &
[1]     28808
```

5.4 - *continue* Dançando o *break*

Nem sempre um ciclo de programa, compreendido entre um `do` e um `done`, sai pela porta da frente. Em algumas oportunidades, temos que colocar um comando que aborte de forma controlada este *loop*. De maneira inversa, algumas vezes desejamos que o fluxo de execução do programa volte antes de chegar ao `done`. Para isto, temos respectivamente, os comandos `break` e `continue` (que já vimos rapidamente na explicação do comado `for`) e funcionam da seguinte forma:



Uso do comando *break*   Uso do comando *continue*

Nas suas sintaxes genéricas eles aparecem da seguinte forma:

```
break [qtd loop]
```

e

```
continue [qtd loop]
```

Onde `qtd loop` representa a quantidade dos *loops* mais internos sobre os quais os comandos irão atuar.

Exemplos:

Vamos fazer um programa que liste os números múltiplos de dois, a partir de zero até um valor informado pelo teclado.

```
$ cat 2ehbom
Conta=1
while [ "$Conta" -le "$1" ]
```

```
do
    Resto=`expr $Conta % 2`
    if  [ "$Resto" -eq 1 ]                  Se resto da divisão por 2 for = 1. Não é múltiplo
    then
        Conta=`expr $Conta + 1`
        continue                            Volta ao comando do sem executar fim do loop
    fi
    echo $Conta eh multiplo de dois.
    Conta=`expr $Conta + 1`
done
```

```
$ 2ehbom
$ 2ehbom 1
$ 2ehbom 9
2 eh multiplo de dois.
4 eh multiplo de dois.
6 eh multiplo de dois.
8 eh multiplo de dois.
```

Para dar um exemplo mais genérico, poderíamos alterar o programa `2ehbom` para ficar com o seguinte formato:

```
$ cat 2ehbom
Conta=0
while true                              É assim que se faz um loop perpétuo
do
    Conta=`expr $Conta + 1`
    if  [ "$Conta" -gt "$1" ]           Se o contador for > limite superior informado
    then
        break                           Bye, bye loop
    fi
    Resto=`expr $Conta % 2`
    if  [ "$Resto" -eq 1 ]
    then
        continue                        Se não for múltiplo de 2 volta ao início do loop
    fi
    echo $Conta eh multiplo de dois.
done
```

Exercícios

1. Existe um arquivo chamado ArqOLs que tem cadastrados os operadores de todos os escritórios estaduais, responsáveis pela leitura de todos os e-mails recebidos, e pela tomada da atitude solicitada por cada e-mail. Este arquivo tem o lay out abaixo:

<Nº OL><TAB><Nome da Máquina do Escritório><TAB><Oper$_1$> <Oper$_2$>....<Oper$_n$>

Fazer um programa que mande um e-mail para todos os operadores de um determinado escritório (recebendo <Nº OL> ou <Nome da Máquina do Escritório> como parâmetro) com o conteúdo de um arquivo (nome recebido com 2º parâmetro).

Capítulo 6

Aprendendo a Ler...

A partir deste capítulo começaremos a nos soltar na programação Shell, já que poderemos montar rotinas com muita utilização e pouca embromação, e aprenderemos a receber dados oriundos do teclado ou de arquivos.

6.1 - Qual Posição Você Prefere?

Nesta seção aprenderemos, principalmente, a posicionar o cursor, além de outras facilidades, para quando necessitarmos receber os dados via teclado, possamos formatar a tela vizando melhorar a apresentação e o entendimento do que esta sendo pedido.

Neste ponto, já devemos saber que existe a instrução `clear`, cuja finalidade é limpar a tela e que deve ser o ponto inicial de qualquer rotina de recepção de dados via teclado.

Para formatação de tela, além do `clear` existe uma instrução multifacetada de uso geral, que é o `tput`. Veremos a seguir as principais faces desta instrução:

- `tput cup` - Cuja finalidade é posicionar o cursor na tela e cuja sintaxe é a seguinte:

  ```
  tput cup lin col
  ```

  Onde `lin` é a linha e `col` a coluna onde se deseja posicionar o cursor. É interessante e importante assinalar que a numeração das linhas e das colunas começa em zero.

- `tput bold` - Coloca o terminal no modo de ênfase, chamando a atenção sobre o que aparecerá na tela a seguir.

- `tput smso` - Coloca o terminal no modo de vídeo reverso, a partir daquele ponto até a seqüência de restauração de tela.

- `tput blink` - Coloca o terminal em modo piscante, a partir daquele ponto até a seqüência de restauração de tela.

- `tput sgr0` - Restaura o modo normal do terminal. Deve ser usado após um dos três comandos acima, para restaurar os atributos de vídeo.

- `tput reset` - Restaura todos os parâmetros do seu terminal ao voltando suas definições ao *default* do *terminfo* definido pela variável do sistema `$TERM` e dá um

clear no terminal. Sempre que possível deve ser usado no final da execução de programas que utilizam a instrução `tput`.

Exemplos:

```
$ cat tputcup
clear
tput cup 3 6
echo ".<-"
tput cup 2 10
echo "/"
tput cup 1 12
echo "/"
tput cup 0 14
echo "_________ Este eh o ponto (3, 6)"
```

Executando vem:

```
$ tputcup
              _________ Este eh o ponto (3, 6)
$           /
           /
      .<-
```

Note que no exemplo acima, propositalmente, a tela foi formatada de baixo para cima, isto é, da linha três para a zero. Isto explica a presença do prompt (**$**) na linha um, já que quando acabou a execução do programa, o cursor estava na linha zero.

Por curiosidade, vamos tirar a instrução *clear* da primeira linha, em seguida vamos listar o programa e executá-lo. Abaixo está a tela resultante destes passos:

```
$ cat tputcup _________ Este eh o ponto (3, 6)
$ ut cup 3 6/
echo ".<-" /
tput cup.<-10
echo "/"
tput cup 1 12
echo "/"
tput cup 0 14
echo "_________ Este eh o ponto (3, 6)"
$ tputcup
```

Nesta bagunça, vimos que a execução do programa foi feita por cima de sua listagem e, conforme dá para perceber, se não usarmos o `clear` no início do programa que trabalha com `tput cup`, sua tela formatada, normalmente fica comprometida.

Já que as outras formas de `tput` envolvem somente atributos de tela, fica difícil em uma publicação, apresentar exemplos que ilustrem seus usos. Podemos, no entanto, digitar para efeito de teste, uma rotina que faça crítica de sexo. Façamos assim:

```
$ cat > testsex
clear
sexo=$1
if  [ "$sexo" -lt 1 -o "(" "$sexo" -gt 2 ")" ]
then
    tput cup 21 20                          Vai para linha 21 coluna 20
    tput smso                               Coloca terminal em vídeo reverso
    echo "Uuuiiii, voce tambem, meu bem!!"
    sleep 5                                 Espera 5 segundos para ler mensagem de erro
    tput sgr0                               Restaura o modo normal do terminal
    tput reset                              Retorna tudo ao original e dá um clear
```

```
    exit 1
fi

tput bold                                   Coloca terminal em modo de realce
tput cup 10 35                              Vai para linha 10 coluna 35

if  [ "$sexo" -eq 1 ]
then
    echo sexo masculino
else
    echo sexo feminino
fi

sleep 5
tput sgr0
tput reset
exit
<^D>
```

Vamos testa-lo assim:

```
$ testsex 1
$ testsex 2
$ testsex 5
```

6.2 - Afinal Como É Que Se Lê?

Com o comando `read`, é claro. Você pode ler dados de arquivos ou diretamente do teclado com esta instrução. Sua sintaxe geral é:

```
read var1 [var2] ... [varn]
```

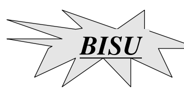
Onde var_1 [var_2] ... [var_n] são variáveis separadas por um ou mais espaços em branco e os valores informados também deverão conter um ou mais espaços entre si.

Exemplos:

```
$ read a b c
aaaaa bbbbbb ccccccc                        Cadeias de caracteres separadas por espaço
$ echo "$a\n$b\n$c"                         O \n significa new-line
aaaaa
bbbbbb
ccccccc
$ read a b
aaaaa bbbbbb ccccccc
$ echo "$a\n$b"
aaaaa                                       A variável $a recebe a 1ª porção
bbbbbb ccccccc                              A variável $b recebe o resto da cadeia
```

Vamos ver agora um trecho de programa que critica uma matrícula recebida pelo teclado. Caso este campo esteja vazio, daremos oportunidade ao operador para descontinuar o programa. Caso não seja numérico, será dado um aviso de que o campo digitado está incorreto, pedindo a sua redigitação:

```
$ cat tstmatr
while true                                          Só sai do loop se encontrar um break
do
    clear
    tput cup 10 15
    echo "Entre com sua matricula: \c"
    read Matric
    if  [ -z "$Matric" ]                            Se a variável Matric estiver vazia...
    then
        tput cup 12 15
        echo "Deseja abandonar? (N/s) \c"
        read sn
        if  [ "$sn" = S -o "(" "$sn" = s ")" ]     Testo se é s em caixa alta ou caixa baixa
        then
            exit
        fi
        continue
    fi
    Matric=`expr $Matric + 0 2> /dev/null`          Se houver erro, jogo a mensagem fora.
    if  [ "$?" -ne 0 ]                              Se último comando foi inválido...
    then
        tput cup 12 15
        echo "Matricula Nao Numerica"
        read a
        continue
    fi
    break
done
```



Caso o separador entre os campos não seja um espaço em branco (o que é muito comum nos arquivos do UNIX), basta trocarmos a variável `IFS` conforme vimos no capítulo 5. Vejamos o exemplo abaixo:

```
$ grep julio /etc/passwd                            O registro estava da forma abaixo
julio:x:60009:1022:Testa aplicativos:/prd1/usr/julio:/usr/bin/ksh
$ OldIFS=$IFS                                       Salvei o IFS
$ IFS=:                                             Fiz o novo IFS valer : como em /etc/passwd
$ grep julio /etc/passwd | read lname nada uid gid coment hdir shini
$ echo "$lname\n$uid\n$gid\n$coment\n$hdir\n$shini"
julio                                               Este é o login name
60009                                               Este é o User Id
1022                                                O Group Id
Testa aplicativos                                   Os comentários
/prd1/usr/julio                                     O Home Directory
/usr/bin/ksh                                        O Shell inicial
$ IFS=$OldIFS                                       Restauro o IFS, voltando tudo à normalidade
```

O exemplo abaixo é real, muito importante, e todos já conhecem:

Primeiramente vamos listar um pedaço do */usr/local/var/ArqOLs*, para que possamos conhecer o seu *lay-out*[5]. Vamos fazer isto listando os quatro primeiros (`head -4`) dos dezesseis últimos (`tail -16`) registros do arquivo:

```
$ tail -16 /usr/local/var/ArqOLs | head -4
13      dupbgp01        ttania tdaniel semanuel
14      duprtl01        areva itarouco oalves
15      dupedb01        alyra lribeiro mmelo hmelo lealobo
16      dupiss01        jroberto cgercira
```

[5] O conhecimento deste arquivo é muito importante já que cabe aos administradores regionais a sua manutenção, que se faz pelo programa *acesso.sh* sendo executado no *dot*.

No pedaço de arquivo acima, notamos que o primeiro campo é a OL, o segundo o nome da máquina e o terceiro os usuários que receberão *e-mail* quando for disponibilizado algum arquivo para a sua regional.

A seguir, a simplificação da rotina que envia *e-mail* para os operadores, extraída do programa */usr/local/bin/uparau.sh*

```
cat /usr/local/var/ArqOLs |                     Passa linha a linha para o while por causa do |
while read OL Maq Opers                         Opers recebe do 3º campo em diante
do
    for Oper in $Opers                          Separa cada um dos operadores das regionais
    do
    mail "$Oper@$Maq" << FimMail                Tudo até o label FimMail faz parte do comando

   Ref. Transferencia de Arquivos

   Informamos que:

O processamento de `date '+%d/%b/%y \`as %R Hr'` disponibilizou o arquivo
`echo "$Arq".Z` no seu diretorio de saida (/prd4/staout/$Site) do `uname`

Lembramos que a politica de backup nao inclui arquivos transitorios  como
o citado acima. E' portanto fundamental a presteza em captura-lo para sua
regional, o que pode  ser feito utilizando  o programa  pegapack.sh, que
que esta disponivel no diretorio /dsv/ftp/pub de durjcv01.

   Saudacoes.
FimMail
    done
done
```

Uma outra forma de ler dados de um arquivo é montar um bloco de programa que faça leitura e redirecionar a entrada primária para o arquivo que se deseja ler. Assim:

```
while read Linha
do
    OL=`echo "$Linha" | cut -f1`
    Arq=`echo "$Linha" | cut -f2`
    Opers=`echo "$Linha" | cut -f3`
done < /usr/local/var/ArqOLs
```

Apesar de correta, a forma acima deve ser evitada por dificultar a legibilidade do programa, já que o nome do arquivo que está sendo lido aparece somente no final do bloco e, mais importante, para manter a compatibilidade com as versões do UNIX mais antigas que a System V Release 2, quando foi introduzido o redirecionamento de E/S na instrução `read`.

BISU Todo cuidado é pouco quando, dentro de um ciclo de leitura de um arquivo, você desejar ler um dado oriundo da tela, como por exemplo: esperar que a tecla `<ENTER>` seja acionada, demonstrando que uma determinada mensagem que você mandou para a tela, já foi lida. Neste caso, a única forma de fazer isto, que conheço, é redirecionando a entrada para */dev/tty* que é o *terminal corrente*. Se este redirecionamento não for feito, o novo `read` (no caso esperando o `<ENTER>`) não lerá do teclado, mas sim do arquivo especificado no *loop* de leitura.

Isto está muito enrolado, vamos ver um exemplo para clarear:

```
$ cat lt
#
#  Lista o conteudo do arquivo de telefones
#

ContaLinha=0                                           Variável para fazer quebra de página
clear
echo "
            Nome                Telefone
"
cat telefones |
while read Linha
do
    if  [ $ContaLinha -ge 21 ]                          Se ContaLinha = 21 quebra a página
    then
        tput cup 24 28
        echo "Tecle <ENTER> para prosseguir ou X para terminar...\c"
        read a < /dev/tty                               Redirecionando a leitura para o teclado
        if  [ "$a" = X ]
        then
            exit                                        Se operador teclou X, termina programa
        fi
        clear
        echo "
            Nome                Telefone
"
        ContaLinha=0
    fi
    echo "            $Linha"
    ContaLinha=`expr $ContaLinha + 1`
done
tput cup 24 49
echo "Tecle <ENTER> para terminar...\c"
read a                                                  Esta fora do loop. Não redirecionei leitura
clear
exit
```

Quando eu fiz `read a` da primeira vez, estava dentro de um *loop* de leitura do arquivo *telefones* e então se não tivesse redirecionado a entrada, o conteúdo do próximo registro de *telefones* teria ido para a variável `$a`. Da segunda vez, por já ter saído do loop de leitura, não foi necessário fazer o redirecionamento.

Outro tipo de programa que o `read` nos permite fazer, é um menu orientado ao sistema que estamos desenvolvendo. Como exemplo, retornaremos aos nossos programas que tratam o arquivo *telefones,* que são: *add*, *pp*, *rem* e *lt*. Vamos reuni-los em um único programa que chamaremos de *teles* e que será responsável pelo controle da execução dos mesmos.

```
$ cat teles
#
#  Menu do cadastro de telefones
#

echo "

                Opcao   Acao
                =====   ====

                  1     Procurar Alguem
                  2     Adicionar Alguem ao Caderno de Telefones
                  3     Remover Alguem do Cadastro de Telefones
                  4     Listagem do Caderno de Telefones

                Escolha Uma Das Opcoes Acima (1-4): \c"
read Opcao
```

```
echo "\n"
case "$Opcao"
in
    1) echo "          Entre com o nome a pesquisar: \c"
       read Nome
       pp "$Nome"
       ;;
    2) echo "          Nome a ser adicionado: \c"
       read Nome
       echo "          Telefone de $Nome: \c"
       read Telef
       add "$Nome" "$Telef"
       ;;
    3) echo "          Nome a ser removido: \c"
       read Nome
       rem "$Nome"
       ;;
    4) lt
       ;;
    *) echo "Soh sao validas opcoes entre 1 e 4"
       exit 1
       ;;
esac
exit
```

Um simples comando `echo` é usado para mostrar todo o menu no terminal, tirando proveito das aspas para preservarem os caracteres *new line* (\010) presentes no texto. O comando `read` é usado para armazenar na variável *Opcao* a escolha do usuário.

Um `case` é então executado para definir que rumo tomar com a escolha. Um simples asterisco (*), representando nenhuma das escolhas anteriores, serve como crítica ao conteúdo da variável *Opcao*.

Vejamos a sua execução:

```
$ teles


                Opcao   Acao
                =====   ====

                  1     Procurar Alguem
                  2     Adicionar Alguem ao Caderno de Telefones
                  3     Remover Alguem do Cadastro de Telefones
                  4     Listagem do Caderno de Telefones

                Escolha Uma Das Opcoes Acima (1-4): 2


          Nome a ser adicionado: Juliana Duarte
          Telefone de Juliana Duarte: (024)622-2876
```

Temos duas maneiras de ver se tudo correu bem: verificando pelo nome da Juliana ou fazendo a listagem geral. Vejamos as duas execuções:

```
$ teles

                Opcao   Acao
                =====   ====

                  1     Procurar Alguem
                  2     Adicionar Alguem ao Caderno de Telefones
                  3     Remover Alguem do Cadastro de Telefones
                  4     Listagem do Caderno de Telefones

                Escolha Uma Das Opcoes Acima (1-4): 1


          Entre com o nome a pesquisar: Juliana
Juliana Duarte (024)622-2876
```

Já no segundo caso, após chamarmos *teles* e fazermos a opção **4** teremos:

```
            Nome               Telefone

            Ciro  Grippi       (021)555-1234
            Claudia Marcia     (021)555-2112
            Enio Cardoso       (023)232-3423
            Juliana Duarte     (024)622-2876 <--- Cá esta ela...
            Luiz  Carlos       (021)767-2124
            Ney Garrafas       (021)988-3398
            Ney Gerhardt       (024)543-4321
            Paula Duarte       (011)449-0989



                                             Tecle <ENTER> para terminar...
```

Vamos executar, novamente, passando uma opção inválida:

```
$ teles

                Opcao   Acao
                =====   ====

                  1     Procurar Alguem
                  2     Adicionar Alguem ao Caderno de Telefones
                  3     Remover Alguem do Cadastro de Telefones
                  4     Listagem do Caderno de Telefones

                Escolha Uma Das Opcoes Acima (1-4): 6


Soh sao validas opcoes entre 1 e 4
```

Neste caso, o programa simplesmente exibe `Soh sao validas opcoes entre 1 e 4` e é descontinuado. Uma versão mais amigável deveria continuar solicitando até que uma opção correta seja feita. Sempre que se fala *até* penso em `until` e efetivamente para que este nosso desejo aconteça, basta colocar todo o programa dentro de um ciclo de `until` que será executado *até* que uma opção correta seja feita.

A grande maioria das vezes que executarmos *teles,* será para consultar o nome de alguém, assim se na chamada do programa estivéssemos passando algum parâmetro, poderíamos supor que a opção desejada por *default* fosse a 1 e já executássemos o

programa *pp*. Assim, poderíamos escrever uma nova versão do programa, com a seguinte cara:

```
$ cat teles
#
#  Menu do cadastro de telefones - versao 2
#

if  [ "$#" -ne 0 ]
then
    pp "$*"                                              Mais que um parâmetro, executo pp
    exit
fi

OK=                                                      Enquanto opção for inválida, $OK estará vazia

until [ "$OK" ]
do
    echo "

                Opcao   Acao
                =====   ====

                  1     Procurar Alguem
                  2     Adicionar Alguem ao Caderno de Telefones
                  3     Remover Alguem do Cadastro de Telefones
                  4     Listagem do Caderno de Telefones

                Escolha Uma Das Opcoes Acima (1-4): \c"
    read Opcao
    echo "\n"
    OK=1                                                 Até que se prove ao contrário, a opção é boa
    case "$Opcao"
    in
        1) echo "          Entre com o nome a pesquisar: \c"
           read Nome
           pp "$Nome"
           ;;
        2) echo "          Nome a ser adicionado: \c"
           read Nome
           echo "          Telefone de $Nome: \c"
           read Telef
           add "$Nome" "$Telef"
           ;;
        3) echo "          Nome a ser removido: \c"
           read Nome
           rem "$Nome"
           ;;
        4) lt
           ;;
        *) echo "Soh sao validas opcoes entre 1 e 4"
           OK=                                           Opção incorreta. Esvazio $OK forçando o loop
           ;;
    esac
done
exit
```

Se a quantidade de parâmetros for diferente de zero então o *pp* é chamado diretamente, passando os argumentos digitados na linha de comando. A variável `$OK` foi criada para controlar o *loop* do `until`, enquanto ela permanecer vazia o programa continuará em *loop*, portanto, logo após receber a escolha, colocamos um valor em `$OK` (no caso colocamos 1, mas poderia ser true, verdadeiro, OK,....) para que servisse para qualquer opção correta. Caso a escolha fosse indevida, então `$OK` seria novamente esvaziada, forçando desta maneira o *loop*. Vejamos então seu comportamento:

```
$ teles Juliana
Juliana Duarte (024)622-2876
$ teles


                Opcao   Acao
                =====   ====

                  1     Procurar Alguem
                  2     Adicionar Alguem ao Caderno de Telefones
                  3     Remover Alguem do Cadastro de Telefones
                  4     Listagem do Caderno de Telefones

                Escolha Uma Das Opcoes Acima (1-4): 5


Soh sao validas opcoes entre 1 e 4


                Opcao   Acao
                =====   ====

                  1     Procurar Alguem
                  2     Adicionar Alguem ao Caderno de Telefones
                  3     Remover Alguem do Cadastro de Telefones
                  4     Listagem do Caderno de Telefones

                Escolha Uma Das Opcoes Acima (1-4): 1


          Entre com o nome a pesquisar: Ney
Ney Garrafas   (021)988-3398
Ney Gerhardt   (024)543-4321
```

1.Vamos engrossar o exercício do capítulo 5, mandando e-mail para todos os 1º operadores de todos os escritórios.

Capítulo 7

Várias Variáveis

Veremos, ao longo deste capítulo o uso das mais importantes[6] *variáveis especiais*. Entenda-se por *variável especial* as variáveis predefinidas do UNIX. Para conhecê-las melhor é necessário, primeiro, entender como funciona o ambiente de uma sessão UNIX. Veja só esta seqüência de comandos:

```
$ ls -l teste                                         Repare que o arquivo teste é executável
-rwxr--r--   1 julio    dipao             9 Nov  7 15:45 teste
$ cat teste
sleep 30                                              Dentro do arquivo tem somente um comando
$ ps -u julio                                         Estes são os meus processos em execução
   PID TTY        TIME COMD
 23160 pts/4      0:00 ps
 19079 pts/4      0:00 ksh
$ teste&                                              Teste terá 30 seg. de execução em background
[1]     23188                                         Nº do processo iniciado em background
$ ps -u julio                                         Novamente verifico os processos em execução
   PID TTY        TIME COMD
 23188 pts/4      0:00 ksh                            Novo processo
 23190 pts/4      0:00 ps
 19079 pts/4      0:00 ksh
 23189 pts/4      0:00 sleep                          Novo processo
```

Repare que o *script* `teste` foi feito para ficar 30 segundos parado e mais nada. Iniciamos este *script* em *background* para liberar a tela e, imediatamente, verificamos quais processos estavam em execução. Existiam dois novos processos: o primeiro, já era de se esperar, é do `sleep` porém, o segundo é um `ksh` que tem o mesmo `process id` que foi gerado quando o `teste` foi colocado em *background* (*PID*=23188). Isto significa que cada *script* inicia um novo Shell que vai interpretar as instruções nele contidas. Para efeito meramente didático, passaremos a nos referir a este novo Shell como um Subshell.

7.1 - Exportar é o que Importa

Vamos mudar totalmente o *script teste* para vermos as implicações dos fatos acima expostos. O teste agora tem esta cara:

[6] Veremos somente as mais importantes porque a quantidade de variáveis especiais é muito grande e no contexto desta publicação não cabe detalhá-las.

```
$ cat teste
echo "a=$a, b=$b, c=$c"
```

Vamos, via teclado, colocar valor nas variáveis `$a, $b, e $c`:

```
$ a=5
$ b=Silvina
$ c=Duarte
```

Vamos agora executar o nosso programa `teste`, já reformulado, para que liste estas variáveis recém criadas:

```
$ teste
a=, b=,c=
```

Ih!! O que foi que houve? Porque os valores das variáveis não foram listados?

Calma meu amigo. Você estava em um Shell, e nele você criou as variáveis, e o *script* `teste` chamou um Subshell para ser o seu intérprete. Ora, se você criou as variáveis em um Shell e executou o programa em outro, fica claro que um programa executado em um Shell não consegue ver as variáveis definidas em outro Shell, certo? Errado, apesar de ter sido exatamente isto que aconteceu, poderíamos ver as variáveis de outro Shell, bastando que as tivéssemos exportado, o que não fizemos.

Se fizermos:

```
$ export b                              Exportei somente a variável $b, sem usar $
```

E novamente:

```
$ teste                                 Executei novamente teste
a=, b=Silvina, c=                       Agora sim! O Subshell enxergou $b
```

E finalmente se fizermos:

```
$ export a c                            Exportei as variáveis $a e $c
$ teste
a=5, b=Silvina, c=Duarte                Agora o Subshell pode ver todas
```

Como a curiosidade é inerente ao ser humano, poderíamos executar o comando *export* sem argumentos para ver o que aconteceria:

```
$ export
EXINIT=set showmode number autoindent   Este foi meu .profile que exportou
HOME=/prd1/usr/julio
HZ=100
KRB5CCNAME=FILE:/tmp/krb5cc_60009_7616
KURL=NO
LOGNAME=julio
MAIL=/var/mail/julio
PATH=/usr/bin:/usr/local/bin:.          Estas variáveis especiais já haviam sido
PGPPATH=/usr/local/bin                  exportadas pelo Shell inicializado, no
PS1=$                                   momento em que foi aberta a sessão.
PWD=/prd1/usr/julio/curso
SHELL=/usr/bin/ksh
TERM=vt220
TERMCAP=/etc/termcap
TZ=GMT0
_=telefones
```

Vamos resumir o modo como as variáveis locais e exportadas trabalham:

- Qualquer variável que não é exportada é uma variável local cuja existência é ignorada pelos Subshells;
- As variáveis exportadas e seus valores são copiados para o ambiente dos Subshells criados, onde podem ser acessadas e alteradas. No entanto, estas alterações não afetam os conteúdos das variáveis dos Shells pais;
- Se um Subshell explicitamente exporta uma variável, então as alterações feitas nesta variável afetam a exportada também. Se um Subshell não exporta explicitamente uma variável, então estas mudanças feitas alteram somente a local, mesmo que a variável tenha sido exportada de um Shell pai;
- As variáveis exportadas retêm suas características não somente para Subshells diretamente gerados, mas também para Subshells gerados por estes Subshells (e assim por diante);
- Uma variável pode ser exportada a qualquer momento, antes ou depois de receber um valor associado.

7.2 - É **.** e Pronto

Suponha que todo dia de manhã logo após se conectar, você queira executar um *script* para preparar seu ambiente de trabalho e tenha a seguinte cara:

```
$ cat direts
BENEF=/prd2/usr/beneficio
ARREC=/prd3/usr/legal/arrecadacao
DB=/prd3/usr/bencdb
TRANS=/prd4/staout/durjcv01
```

O intuito deste *script* seria que após sua execução bastaria fazer:

```
cd $DB
```

Que eu já estaria no diretório `/prd3/usr/bencdb`[7]. Vamos então executar este Shell e vejamos o que acontece:

```
$ direts
$ echo $ARREC

$
```

Ora, isto era de se esperar, pois o meu *script* em sua execução chamou um novo Shell para interpretá-lo e as variáveis foram valoradas neste Shell filho. Não adiantaria sequer exportá-las, já que não podemos exportar variáveis para Shells pais. O que fazer então?

Felizmente, existe um comando do Shell chamado **.** (ponto, fala-se *dot*) que pode quebrar-nos este galho, cujo *formato geral é:*

```
. arquivo
```

[7] Existem formas melhores de fazer isto, como veremos mais a frente neste mesmo capítulo.

e cujo propósito é executar as instruções contidas em um arquivo no, Shell chamador, que neste caso será sempre o *Shell corrente,* como se elas tivessem sido digitadas neste ponto. Então, por não ser gerado um Subshell filho, tudo o que for executado pelo comando **.** (*dot*) tem efeito direto no ambiente da sessão na qual estamos conectados.

Por não estarmos executando um *script* em Shell, mas sim um comando, o **.** (*dot*), o `arquivo` a ser executado não necessita ter permissão de execução. Vamos então executar novamente o *direts* com o auxilio do **.** (*dot*):

```
$ . direts                          Comando . executando o script
$ cd $TRANSM
$ pwd                               Será que as variáveis permanecem valoradas?
/prd4/staout/durjcv01               Iuppiiii!!!
```

Conforme você viu, as variáveis continuam com os valores atribuídos pelo *script*, desta forma validando o uso do comando **.** (*dot*).

7.3 - Principais Variáveis do Sistema

Para que você possa ver as variáveis pré-definidas, que são associadas a cada sessão Shell que você abra, basta você fazer:

```
$ set | pg
```

Abaixo montaremos uma tabela mostrando as principais *variáveis especiais* juntamente com os seus conteúdos, para que possamos entender os seus usos dentro de *scripts*:

Variável	Conteúdo
HOME	Nome do diretório onde você é colocado no momento em que se conecta.
PATH	Caminhos que serão pesquisados para tentar localizar um programa especificado.
CDPATH	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 diretório com bastante níveis.
PWD	Diretório corrente. (Válido somente no Bourne Shell - sh)
LOGNAME	Login Name do usuário.
PS1	Caracteres que compõem o prompt primário (default=$ ou # para root).
PS2	Caracteres que compõem o prompt secundário (default = >).
IFS	Já foi visto antes. Contém o caracter que está servindo como separador default entre os campos.
TERM	Tipo de terminal que está sendo emulado.
EXINIT	Parâmetros de ambiente do vi (editor)

Exemplos:

Vou aproveitar o exemplo do $HOME e mostrar uma série de modos de mudar de diretório usando o comando *cd*:

```
$ pwd                                          Onde estou?
/prd1/usr/julio/curso
$ cd                                           Volto para o diretório home (home directory)
$ pwd
/prd1/usr/julio
$ cd -                                         Volta para o diretório anterior ao último cd*
/prd1/usr/julio/curso
$ cd $HOME                                     Outra forma de ir para o home directory
$ pwd
/prd1/usr/julio
$ cd -                                         Vide nota de rodapé*
/prd1/usr/julio/curso
$ cd ~julio                                    O til significa home. Assim vou p/ home do julio*
$ pwd
/prd1/usr/julio
$ cd ~jneves                                   Vou para o home do jneves*
$ pwd
/dsv/usr/jneves
$ bronze                                       Agora vamos executar o bronze. Lembra-se?
ksh: bronze:  not found                        Não achou porque prog. estava em ~julio/curso
$ echo $PATH                                   Onde procurei o bronze?
/usr/bin:/usr/local/bin:.                      Diretório que o bronze reside não foi procurado
$ PATH=$PATH:~julio/curso                      O separador entre os diretórios de $PATH é :*
$ echo $PATH
/usr/bin:/usr/local/bin:.:/prd1/usr/julio/curso ~julio/curso foi incluído na variável*
$ bronze                                       Agora sim a execução será bem sucedida
11
22
33
44
55
66
77
88
99
$ cd
$ d                                            Script que lista diretórios abaixo do especificado
/prd1/usr/julio:
  /prd1/usr/julio/arrec
  /prd1/usr/julio/bancos
  /prd1/usr/julio/benef
  /prd1/usr/julio/c
  /prd1/usr/julio/curso                        } Diretórios abaixo de /prd1/usr/julio
  /prd1/usr/julio/movpack
  /prd1/usr/julio/newtrftp
  /prd1/usr/julio/stados
  /prd1/usr/julio/trftp
$ cd curso
/prd1/usr/julio/curso
$ cd trftp                                     Se tivesse feito cd ../trftp teria funcionado...
ksh: trftp:  not found
$ cd bin                                       Isto não pode funcionar
/usr/local/bin                                 Ué!!! Funcionou...
$ echo $CDPATH
.:/usr/local                                   Só foi procurado o diretório corrente e /usr/local
```

---

[*] As construções *cd* - e o ~ *(til)* referindo-se ao *diretório home* só são válidas no *Korn Shell* (*ksh*)

```
$ CDPATH=.:..:/usr/local                Pai do diretório corrente adicionado à pesquisa
$ cd -                                  Vide nota de rodapé da página anterior*
$ pwd
/prd1/usr/julio/curso
$ cd trftp
/prd1/usr/julio/trftp                   Agora consegui fazer o cd direto
$ cd -                                  Vide nota de rodapéda página anterior*
```

O prompt secundário aparece quando damos um <ENTER> para finalizar uma linha sem encerrar o comando. Como exemplo didático, vamos alterar o conteúdo de PS2 para simularmos uma indentação:

```
$ echo $PS1
$
$ echo $PS2
>
$ PS2=">     "                          Atribui a PS2 um > seguido por 5 espaços
$ for i in $LOGNAME $TERM
>     do
>     echo $i
>     done
julio                                   Login Name do usuário conectado
vt220                                   Terminal que esta sessão esta emulando
```

Para finalizar é comum termos no *$HOME/.profile* as seguintes linhas de preparação de nossas variáveis de ambiente:

```
export PS1=`uname -n`:'$PWD>'
export EXINIT='set showmode number autoindent ignorecase'
PATH=$PATH:.
export CDPATH=.:~julio:..:/usr/local
```

Repare que nas linhas acima, o mesmo comando que atribui valor às variáveis também as exporta.

No PS1 do *$HOME/.profile* é comum fazermos esta construção, que indica o nome da máquina na qual se está conectado[8] (comando `uname -n`), seguido de dois-pontos (**:**), do caminho absoluto de onde se está posicionado[9] e do sinal maior-que (**>**).

A variável `$EXINIT` passa para o editor *vi* as seguintes definições de ambiente:

`showmode:` Desta forma, na última linha da tela, aparecerá o modo em que o `vi` está trabalhando (`insert, replace, change, append,....`);

`number:` Com esta opção ativada o `vi` colocará à esquerda, somente na tela sem interferência alguma no arquivo, o número seqüencial de cada linha;

`autoindent:` Colocamos `autoindent` em `$EXINIT`, para que o `vi` ajuste a margem esquerda de cada linha pela da linha anterior, indentando o texto que está sendo editado;

[8] Só se deve incluir o nome da máquina, no prompt de usuários que navegam por diversas máquinas UNIX ligadas em rede.

[9] O caminho absoluto do diretório onde nos encontramos, só é informado no Korn Shell. No Bourne Shell não existe a variável *PWD* e o *prompt* ficaria: **Nome_da_Máquina:$PWD>**

`ignorecase:` Deste modo, quando estivermos pesquisando uma *cadeia de caracteres* no texto que está sendo editado, o `vi` não fará distinção entre letras maiúsculas e minúsculas.

Além das variáveis descritas acima existem muitas outras, que creio não caberem no escopo deste trabalho já que, para efeito de programação em Shell, têm pouquíssima utilidade. Para termos uma idéia, sabendo-se que acabei de conectar-me e ainda não exportei nem sequer criei nenhuma variável, veja só as linhas a seguir:

```
$ set | wc -l
     35
```

Isto significa que foram criadas 35 variáveis de ambiente no instante em que me conectei .

Exercícios

1. Escreva um programa chamado `meurm` que recebe como argumento os nome dos arquivos a serem removidos. Se a variável global MAXFILES estiver valorada, então tome este valor como o número máximo de arquivos a remover sem perguntas. Se a variável não existir ou estiver com valor nulo, use 10 como o máximo. Se o número de arquivos a ser removido exceder a contagem, solicite ao operador a confirmação antes de removê-los.

A seguir o resultado esperado:

```
$ ls | wc -l
25
$ meurm *
Removo 25 Arquivos? (s/n) n
Os Arquivos nao foram removidos
$ MAXFILES=100
$ meurm *
$ ls
$
```

Capítulo 8

Saco de Gatos

Neste capítulo, veremos comandos que, por uma razão ou por outra, não se encaixam em nada que vimos até agora. Não existe também nada de particular na seqüência de apresentação dos tópicos. Ou seja: **É O FINAL, PESSOAL**. Acabando este, podemos partir para as louras geladas para comemorar o dispêndio de saco ao longo dos últimos 5 dias.

8.1 - A *1ª Faz Tchan, a 2ª Faz Tchun, e Tchan, Tchan,..., Tchan*

Esta seção descreverá um comando bastante diferente do que estamos acostumados em Shell: `eval`. Seu formato é o seguinte:

```
eval <linha-de-comando>
```

Onde `<linha-de-comando>` é uma linha de comando comum, que você poderia executar teclando direto no terminal. Quando você põe *eval* na sua frente, no entanto, o efeito resultante é que o Shell resolve a linha de comandos antes de executá-la[10]. Para um caso simples, realmente não tem efeito:

```
$ eval echo "Meu Login Name eh $LOGNAME"
Meu Login Name eh julio
```

Esta linha de comandos produziu o mesmo resultado que teria caso não tivesse sido usado o `eval`.

Exemplos:

Considere agora o exemplo seguinte sem o uso do `eval:`

```
$ paipi='|'
$ ls $paipi wc –l
|: No such file or directory
wc: No such file or directory
–l: No such file or directory
```

[10] O que acontece é que o *eval* simplesmente executa a linha de comandos passada para ele como argumento; então, o Shell processa esta linha de comandos enquanto passa os argumentos para o *eval*, e novamente a linha de comandos é executada, desta vez pelo *eval*. O efeito resultante é a dupla execução da linha de comando pelo Shell.

Os erros acima são oriundos do comando `ls`. O Shell tentará fazer os *pipelines* e os redirecionamentos de E/S (I/O) antes da substituição das variáveis, então ele nunca reconhecerá o símbolo de *pipe* (|) dentro da variável *paipe*. O resultado disto é que **|**, `wc`, `-l` são interpretados como parâmetros do comando `ls`.

Se pusermos o `eval` na frente da linha de comandos teremos o resultado desejado. Então, temos que fazer:

```
$ eval ls $paipi wc -l
     37
```

Vejamos outro:

```
$ cat medieval
#
#  medieval - Modulo-Exemplo Da Instrucao EVAL
#

echo Recebi $# parametros.
i=1
while [ "$i" -le $# ]
do
    echo "parametro $i = \c"
    echo $`echo $i`
    i=`expr $i + 1`
done

$ medieval Marcos Valdo da Costa Freitas
Recebi 5 parametros.
parametro 1 = $1
parametro 2 = $2
parametro 3 = $3
parametro 4 = $4
parametro 5 = $5
```

No exemplo acima, queríamos mostrar os parâmetros passados para o *script* `medieval`, numerando-os. Quase conseguimos, só faltou o Shell resolver os parâmetros posicionais, isto é, quando na execução do *script* o Shell chegou até aos nomes dos parâmetros `($1, $2,...$5)`, faltando então, mais uma passada do interpretador para resolvê-los. Para dar esta passada a mais, é que usaremos o comando `eval`. Vejamos então como ficará:

```
$ cat medieval
#
#  medieval - Modulo-Exemplo Da Instrucao EVAL
#

echo $# parametros.
i=1
while [ "$i" -le $# ]
do
    echo "parametro $i = \c"
    eval echo $`echo $i`                    A 1ª faz tchan, a 2ª faz tchun e tchan, tchan...
    i=`expr $i + 1`
done
$ medieval Marcos Valdo da Costa Freitas
Recebi 5 parametros.
parametro 1 = Marcos
parametro 2 = Valdo
parametro 3 = da
parametro 4 = Costa
parametro 5 = Freitas
```

Isto também poderia ter sido feito assim:

```
$ cat medieval
#
#  medieval - Modulo-Exemplo Da Instrucao EVAL
#

echo Recebi $# parametros.
i=1
while [ "$i" -le $# ]
do
    echo "parametro $i = \c"
    eval echo \$$i                          A | na 1ª passagem inibe a interpretação do $
    i=`expr $i + 1`
done
```

8.2 - *wait* a Minute Mr. Postman

Se você passar uma linha de comandos para execução em *background*, esta linha de comandos será executada em um Subshell que é independente do seu Shell corrente (diz-se que está em modo assíncrono). Algumas vezes é conveniente esperar que a execução deste processo em *background* termine antes de prosseguir.

Suponha que você tenha um grande arquivo que deva ser classificado, cujo `sort` foi mandado para *background*, e agora deseja esperar o fim deste `sort` já que você precisa usar estes dados classificados.

Para isto usamos o comando *wait* e seu formato geral é:

```
wait PID
```

Onde `PID` é o `Process-ID` que desejamos aguardar a conclusão. Se omitido, o Shell espera a finalização de todos os processos filhos. A execução do Shell corrente é suspensa até que o processo ou processos terminem a execução. Vamos ver um exemplo desenvolvido direto no terminal:

```
$ sort ArqGrande -o ArqGrande &       Manda para background
6924                                  O PID é mandado para a tela pelo Shell
$ ....                                execute outras tarefas independentes...
$ wait 6924                           Agora espera o final do sort para prosseguir
$                                     Quando o sort termina o prompt é devolvido
```

8.3 - Para Evitar Trapalhadas Use o *trap*

Quando você descontinua a execução de um programa, seja teclando `DELETE`, `BREAK` ou `<CTRL>+C` no seu terminal durante a execução de um *script*, normalmente este programa é imediatamente terminado e o *prompt* do terminal é retornado. Isto nem sempre é desejável. De repente você está largando alguma sujeira (Opa, você não, o *script*), como um arquivo temporário, que deveria ser limpa.

A finalização de um programa, entre outras coisas, manda o que conhecemos como *sinal* para o programa ainda em execução. O programa pode então especificar qual

atitude deve ser tomada quando receber um determinado *sinal*. Isto deve ser feito com o comando `trap`, cujo uso geral é:

```
trap "<comando1>; <comando2>; <...>; <comandon>" sinais
```

Onde `<comando1>; <comando2>; <...>; <comandon>` significa um ou mais comandos (separados por ponto-e-vírgula, claro) que serão executados caso um dos sinais especificados por `sinais` seja recebido.

Números são associados aos diferentes tipos de sinais, e os mais comumente usados em programação Shell por, normalmente significarem fim de execução de programa, estão listados na tabela abaixo:

Sinal	Gerado por
0	Saída normal do programa
1	Quando recebe um `kill -hup`
2	Interrupção pelo teclado (ex. `<ctrl>+c`)
15	Sinal de terminação do software (mandado pelo `kill` *default* ou `kill PID`)

Exemplos:

Vamos voltar ao exemplo dado lá no início do manual (página 4.1). Veja só:

```
$ ftp -ivn remocomp << FimFTP >> /tmp/$$ 2>> /tmp/$$
>    user fulano segredo
>    binary
>    get arqnada
>FimFTP
$
```

*Os sinais **>** são prompts secundários do UNIX. Enquanto não surgir o label FimFTP o **>** será o prompt (PS2), para indicar que o comando não terminou.*

Repare no trecho do `ftp` acima que, tanto as saídas do `ftp`, como os erros encontrados, estão sendo redirecionados para */tmp/$$*. Caso este `ftp` seja interrompido por um `kill` ou um *`<CTRL>+C`*, certamente deixará lixo no disco. É exatamente esta a forma que mais se usa o comando `trap`. Se o `ftp` acima fosse trecho de um *script*, deveríamos, logo no início deste *script*, como um de seus primeiros comandos, fazer:

```
trap "rm -f /tmp/$$ ; exit" 0 1 2 15
```

Desta forma, caso houvesse uma interrupção brusca (*sinais 1, 2* ou *15*) antes do programa encerrar (no `exit` dentro do comando `trap`), ou um fim normal (*sinal 0*), o arquivo */tmp/$$* seria removido.

Caso na linha de comandos do `trap` não houvesse o comando `exit`, seus comandos seriam executados e o fluxo do programa retornaria ao ponto em que estava quando recebeu o *sinal* para execução deste `trap`.

Caso a linha de comandos do `trap` possua mais do que um comando, eles deverão estar entre aspas (""). Note também que o Shell pesquisa a linha de comandos enquanto o `trap` é executado e novamente quando um dos *sinais* listados é recebido. Então, no último exemplo, o valor de `$$` será substituído no momento que o comando `trap` for executado. Se você desejasse que a substituição fosse realizada somente quando o *sinal* fosse recebido, o comando deveria ser colocado entre apóstrofos(''), ficando da seguinte maneira:

```
trap 'rm -f /tmp/$$ ; exit' 0 1 2 15
```

Outra grande aplicação do `trap` é tornar programas mais amigáveis. Caso você estivesse executando um programa do tipo menu (p.ex. *teles*) que tivesse chamado outro (p.ex. *pp*) e caso este último recebesse um *sinal* para ser descontinuado, por intermédio do comando `trap` poderíamos fazer com que o comando do programa fosse devolvido para o programa anterior, no caso o menu (*teles*).

O comando `trap`, quando executado sem argumentos, lista os *sinais* que estão sendo monitorados no ambiente, bem como a linha de comando que será executada quando tais *sinais* forem recebidos.

Se a linha de comandos do `trap` for nula (vazia), isto significa que os *sinais* especificados devem ser ignorados quando recebidos. Por exemplo, o comando:

```
trap "" 2
```

Especifica que o *sinal* de interrupção deve ser ignorado, porque você pode querer ignorar certos *sinais* quando executar alguma operação. No caso acima, quando não se deseja que sua execução seja interrompida.

Note que o primeiro argumento deve ser especificado para que o *sinal* seja ignorado, e não é equivalente a escrever o seguinte, que tem um sentido diferente por si próprio:

```
trap 2
```

Se você ignora um *sinal*, todos os Subshells irão ignorar este *sinal*. Portanto, se você especifica qual ação deve ser tomada quando receber um *sinal*, então todos os Subshells irão também tomar a ação quando receberem este *sinal*. Para o *sinal* que temos mostrado (sinal 2), isto significa que os Subshells serão encerrados.

Suponha que você execute o comando:

```
trap "" 2
```

e então execute um Subshell, que tornará a executar outro *script* como um Subshell. Se for gerado um *sinal* de interrupção, este não terá efeito nem sobre o Shell principal nem sobre os Subshell por ele chamados, já que todos eles ignorarão o *sinal*.

No entanto, se ao invés de executar o comando `trap` prévio, você fizesse:

```
trap : 2
```

e então executasse Subshells, quando recebesse o sinal de interrupção o Shell pai o ignoraria (ele executará o comando nulo), porém os subsequentes (filhos) que estivessem ativos seriam encerrados.

Se você mudou a ação *default* a ser tomada quando receber um *sinal*, você pode desfazer isto usando o comando `trap`, se você simplesmente omitir o primeiro argumento, então:

```
trap 1 2
```

retornará a ação a ser tomada no recebimento dos *sinais* 1 e 2 para o *default*.

8.4 - Parâmetros

Muita coisa já foi vista até aqui referente a parâmetros tais como parâmetros posicionais (`$1`, `$2`,...,`$9`), quantidade de parâmetros passados ou recebidos (`$#`), código de retorno (`$?`). Veremos agora outras formas de lidarmos com parâmetros.

Construções com parâmetros e variáveis

- **${parâmetro}**

Suponha que a variável (parâmetro) `Num` cada vez que passar em um ponto de um *script* seja incrementada e seja impresso um aviso do ordinal de passadas por aquele ponto. Abaixo este fragmento de programa:

```
...
Num=1
while [ "$Num" -le 3 ]
do
    echo ${Num}a. passada
    Num=`expr $Num + 1`
done
...
```

Se o executarmos teremos:

```
1a. passada
2a. passada
3a. passada
```

Se não colocássemos a variável `Num` entre chaves (`{}`) viria:

```
. passada
. passada
. passada
```

O resultado acima foi obtido quando o Shell interpretou a linha:

```
echo $Numa. passada
```

Ora, como não existe a variável `Numa`, foi gerado um nulo, seguido de **.** passada, o que é totalmente indesejável.

Concluímos então que as chaves (`{}`) devem ser usadas para evitar uma eventual interpretação erronea do nome de variáveis causada pelos caracteres que seguem o seu nome.

Exemplos:

```
$ mv $Arq ${Arq}1
```

- **${parâmetro:-valor}**

Neste tipo de construção caso a variável *parâmetro* seja nula, receberá *valor,* caso contrário permanecerá com o conteúdo anterior.

Exemplos:

No trecho de *script* abaixo, eu ofereço `$LOGNAME` como valor *default* para a variável `User`. Caso seja teclado simplesmente um `<ENTER>, desta forma` aceitando-se o valor oferecido, a variável `User` receberá o valor de `$LOGNAME`.

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

A forma convencional (e conservadora) de se escrever esta rotina, sem a substituição de parâmetros seria:

```
echo "Login Name em $Site ($LOGNAME):"
read User
if  [ -z "$User ]
then
    $User=$LOGNAME
fi
echo $User
```

- **${parâmetro:+valor}**

O oposto da anterior, isto é, se o *parâmetro* é não nulo, o Shell substitui seu valor; caso contrário, o Shell ignora este comando.

Exemplos:

```
$ Var=x
$ echo Variavel ficou com o valor: ${Var:+"Outro Valor"}
Variavel ficou com o valor: Outro Valor
$ Var=
$ echo Variavel ficou com o valor: ${Var:+"Outro Valor"}
Variavel ficou com o valor:
```

8.5 - Funções

A partir da *Release* 2 do *System* V do UNIX, a interpretação de funções foi agregado ao Shell, com o seguinte formato geral:

```
Funcao ()   { comando1; comando2; ...; comandon }
```

Ou então, para tornar mais legível:

```
funcao ()
{
    comando₁
    comando₂
    ...
    comandoₙ
}
```

Onde `funcao` é o nome da função, os parênteses dizem ao Shell que uma função está sendo definida, e os comandos envolvidos pelas chaves (`{}`) definem o corpo da função. Estes comandos serão executados sempre que a função `funcao` for chamada à execução. Note que pelo menos um espaço em branco deve ser colocado entre os comandos (primeiro e último) e as chaves de início e de fim (`{}`).

Os argumentos listados após a chamada da função na linha de comandos do programa, são tratados pela função como se fossem parâmetros posicionais `$1`, `$2`, ..., como qualquer outro comando.

Exemplos:

Suponhamos que, quando um campo que o programa solicitasse a entrada pelo teclado fosse informado com valor nulo, o programa interpretasse isto como uma possível desistência do operador. A rotina abaixo, deveria ser colocada após cada `read`:

```
while true                                  Loop de leitura
do
    read isto
    if  [ -z "$isto" ]                      Se o campo estiver vazio...
    then
        Pergunta "Deseja continuar" S N     Veja: 1º parm.=frase, 2º valor default, 3º o outro
        if  [ "$SN" = N ]                   Vamos ver se o operador continua ou desiste
        then
            exit
        fi
        continue
    fi
    ...
done
```

A minha função `Pergunta` recebe como primeiro parâmetro a frase que será exibida (que é a própria pergunta), o segundo parâmetro é a resposta considerada *default* (a que tem mais probabilidade de ocorrer) e o terceiro parâmetro é o outro valor possível.

Para desenvolver a função parti de alguns pressupostos:

- A função colocaria um ponto-de-interrogação no final da frase;
- Após o ponto-de-interrogação, seria oferecido o valor *default,* seguido de um par de barras (`//`) para convencionar que aquele valor é o *default;*
- Este conjunto acima seria exibido no centro da linha 21.

A função que foi escrita tem a seguinte cara:

```
Pergunta ()
{
    DefVal=`echo "$2" | tr "[a-z]" "[A-Z]"`         Converte o valor defaul par maiúscula
    OthVal=`echo "$3" | tr "[a-z]" "[A-Z]"`         Ídem com o outro valor
    Quest=`echo "${1}"\? "${DefVal}"'//'`           A variável Quest recebe o texto já montado
    Len=`expr length "$Quest"`                      }
    Col=`expr "(" 80 - "$Len" ")" / 2`              } Centralizo o texto na linha 21
    tput cup 21 $Col                                }
    echo "$Quest\07\c"                              O \07 é para soar o alarme
    read SN                                         Variável SN recebe a resposta do operador

    SN=${SN:-"$DefVal"}                             Se operador deu <ENTER>, SN recebe default

    SN=`echo $SN | tr "[a-z]" "[A-Z]"`              Converte resposta para maiúscula
    if  [ "$SN" != "$OthVal" ]                      Qualquer coisa que não seja o outro valor...
    then
        SN="$DefVal"                                Será tomada como o valor default
    fi

    tput cup 21 1                                   Limpando a linha da pergunta
    echo "
"
    return
}
```

Usei o exemplo desta forma para melhorar o seu detalhamento. Mas, como o Shell é um interpretador (acho que intérprete fica melhor, né?), ele lê o programa seqüencialmente, do início para o fim, e só entenderá uma chamada de função caso ele leia a sua declaração antes desta chamada. Com isto eu quero dizer que a boa estruturação de um programa deve ser feita conforme o gráfico abaixo:

No gráfico acima, o que é chamado de *declaração de variáveis globais* é a área onde deve-se especificar as variáveis que serão vistas pelo *corpo do programa*, e por alguma(s) função(ões) e/ou por algum(ns) Subshell(s).

Exercícios

1. Escreva uma função para colocar na tela uma mensagem aguardando que o operador tecle <ENTER> para poder continuar.

Esta rotina será chamada recebendo a mensagem, a linha e a coluna. Caso a coluna não seja passada, a mensagem deverá ser colocada no centro da linha especificada.

2. Fazer um programa (usando a instrução eval) que liste os parâmetros recebidos em ordem inversa.