04

Mais sobre globbing e quoting

Globbing e o asterisco

Já vimos diversos pontos importantes dessa seção, faltam o *globbing* e o *quoting*.

Vamos começar com o *globbing*.

Estamos em nosso terminal, qual o nome do diretório mesmo? Para saber isso digitamos `pwd` e ele nos contesta `/home/guilherme` .

```
> pwd
/home/guilherme
```

Vamos mudar de diretório, vamos para o diretório `Documents/` . Para isso, digitamos `cd Documents/` .

Agora, vamos criar alguns arquivos de texto. Vamos criar um primeiro arquivo de texto. Para tanto, digitamos `gedit` e o nome do arquivo que queremos editar, `texto1.txt` e irá abrir o editor. Nesse editor escreveremos `O conteudo do primeiro arquivo` e estaremos criando um arquivo de texto. Vamos salvar e fechar isso.

Vamos usar o `gedit` de novo, poderia ser outro editor. Escreveremos `gedit texto2.txt` . Vai abrir o editor e nele escreveremos `O conteudo do segundo arquivo` . Vamos salvar isso e fechar.

Vamos editar mais um arquivo, para isso digitaremos `gedit texto3.txt` e abrirá o editor onde escreveremos `Conteudo do terceiro arquivo` . Salvamos e fechamos isso.

Vamos editar, ainda, o conteúdo de um décimo arquivo, para isso, digitamos `gedit texto10.txt` e no editor que abrirá escreveremos `Conteudo do decimo arquivo` . Salvamos isso e fechamos.

Vamos observar nossa tela:

```
> pwd
/home/guilherme
> cd Documents
~/Documents$ gedit texto1.txt
~/Documents$ gedit texto2.txt
~/Documents$ gedit texto3.txt
~/Documents$ gedit texto10.txt
```

Observe que temos diversos arquivos de textos.

Agora, imagine que estamos em nossa linha de comando e gostaríamos de ler o conteúdo de um desses arquivos.

Como sabemos quais são os arquivos que estão em nosso diretório atual? Para isso utilizamos o `ls` e teremos o seguinte:

```
~/Documents$ ls
curriculum texto10.txt  texto1.txt  texto2.txt  texto3.txt
```

Como observamos o conteúdo de um arquivo? Utilizamos o `cat` e o nome do arquivo, por exemplo `cat texto1.txt`. Vamos observar o que acontece:

```
cat texto1.txt
O conteudo do primeiro arquivo
```

Ele nos mostra o conteúdo do arquivo, isto é, o que está escrito nesse arquivo. Podemos utilizar o cat para todos os arquivos que acabamos de criar, o `cat texto 2.txt`, o `cat texto3.txt` e `cat texto10.txt`. Vamos observar:

```
~/Documents$ cat texto2.txt
O conteudo do segundo arquivo
~/Documents$ cat texto3.txt
Conteudo do terceiro arquivo
~/Documents$ cat texto10.txt
Conteudo do decimo arquivo
```

Podemos observar o conteúdo de todos os arquivos e podemos dar `cat` um a um para verificar o que esses arquivos possuem de conteúdo.

E, poderíamos fazer isso de uma maneira diferente?

Sim, poderíamos digitar `cat texto1.txt texto2.txt texto3.txt texto10.txt`. Ao digitar isso teremos os conteúdos de todos os arquivos que digitamos os nomes. Teremos o seguinte:

```
~/Documents$ cat texto1.txt texto2.txt texto3.txt texto10.txt
O conteudo do primeiro arquivo
O conteudo do segundo arquivo
Conteudo do terceiro arquivo
```

Ele pegou o conteúdo de todos esses arquivos e mostrou para nós. O comando `cat` recebe o nome de vários arquivos. Agora, o que gostaríamos de fazer é executar o `cat` para os quatro arquivos que criamos, sem ter que digitar seus respectivos nomes, um a um. Vamos observar quais são os arquivos, para ver isso, usaremos o `ls`:

```
~/Documents$ ls
curriculum  texto10.txt     texto1.txt      texto2.txt      texto3.txt
```

Então, queríamos ler o nome de vários arquivos. Queríamos dar um `cat` em tudo, podemos digtar `cat* txt` e estaremos dizendo, tudo que termina em `txt`. Ele nos mostrará o seguinte:

```
~/Documents$ cat* .txt
Conteudo do decimo arquivo
O conteudo do primeiro arquivo
O conteudo do segundo arquivo
Conteudo do terceiro arquivo
```

O que aconteceu aqui? Já conversamos um pouco sobre casos similares. O *shell* antes de executar o comando *cat,* interpreta os detalhes do comando, da linha, por exemplo o `$` é trocado pelo valor de uma variável. O asterisco procura arquivos com o padrão "qualquer coisa" mais o `.txt`. Esse comando vai ser substituído por `cat textotexto10.txt texto1.txt texto2.txt`

`texto3.txt` e conforme essa ordem o asterisco está sendo expandido para todos esses comandos. Só depois o *shell* executa esse comando e traz o conteúdo. Lembre que o asterisco é interpretado e expandido pelo *shell* antes de executar o comando, e é assim também com as variáveis, o `$` que era primeiro interpretado e expandido para o valor das variáveis e depois era executado de verdade.

Da mesma maneira que pudemos utilizar o `*` junto com o `txt`, poderíamos fazer `cat texto*`. Com isso, estaríamos dizendo para pegar tudo o que começa com `texto` e mostrar o conteúdo disso. Teremos:

```
~/Documents$ cat texto*
Conteudo do decimo arquivo
O conteudo do primeiro arquivo
O conteudo do segundo arquivo
Conteudo do terceiro arquivo
```

Repare que esse uso, apesar de um pouco distinto do outro jeito, o `cat *.txt` expandi-se da mesma maneira, é extremamente comum esse tipo de expansão, onde o asterisco é utilizado como uma espécie de coringa para vários arquivos. Por exemplo, gostaríamos de compilar todos os arquivos que terminam com `.c`, isto é, os arquivos de programação c. Então, poderíamos digitar `gcc *.c`, com isso estamos dizendo, tudo o que termina com `.c`. Não iremos executar isso, apenas daremos um "Ctrl+C" para parar de executar esse comando.

```
~/Documents$  gcc *.c^C
```

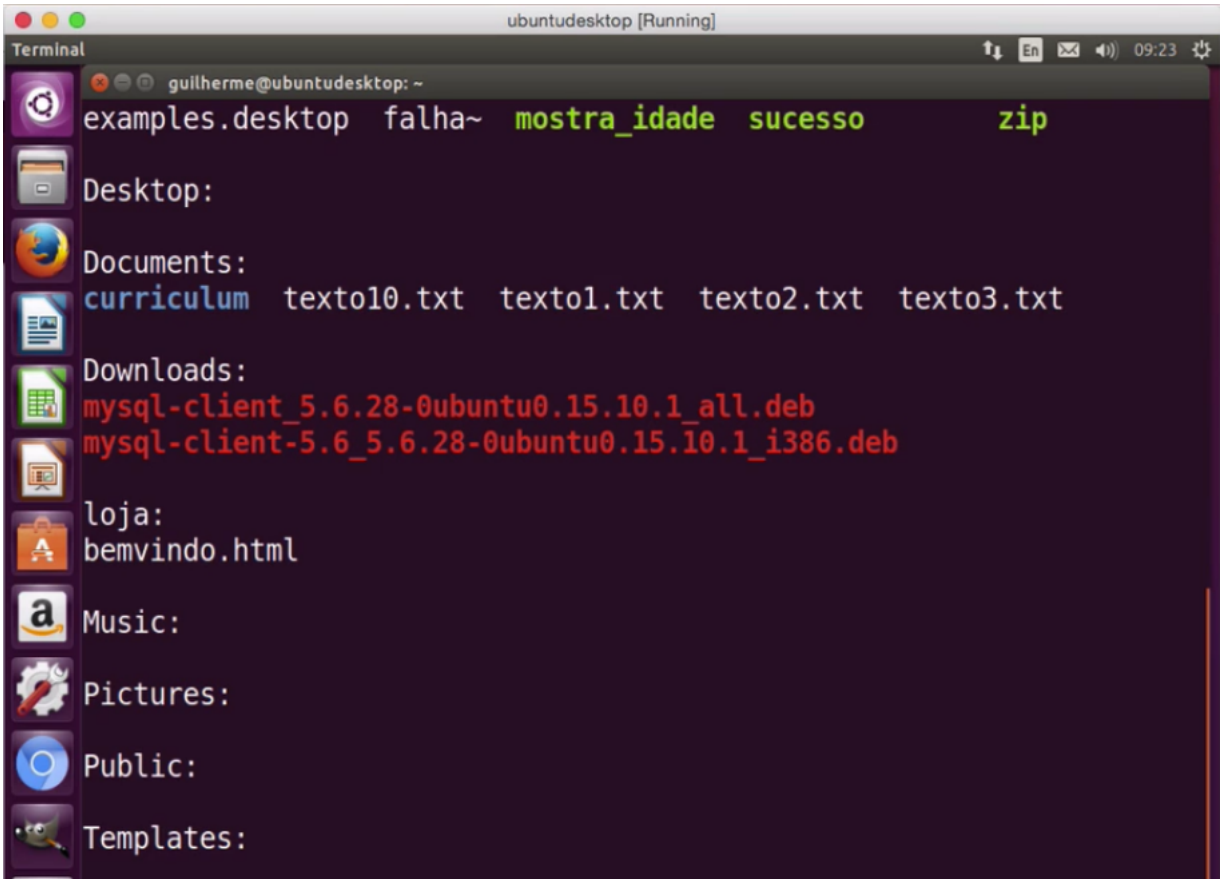
Gostaríamos de ver todos os arquivos que estão no diretório e que terminam em `txt`. Podemos digitar, `ls *.txt`. O que acontece aqui? Quando digitamos o `ls *.txt` o próprio *shell* substitui o `*.txt` pelo seguinte:

```
~/Documents$  ls*.txt
texto10.txt     texto1.txt      texto2.txt      texto3.txt
```

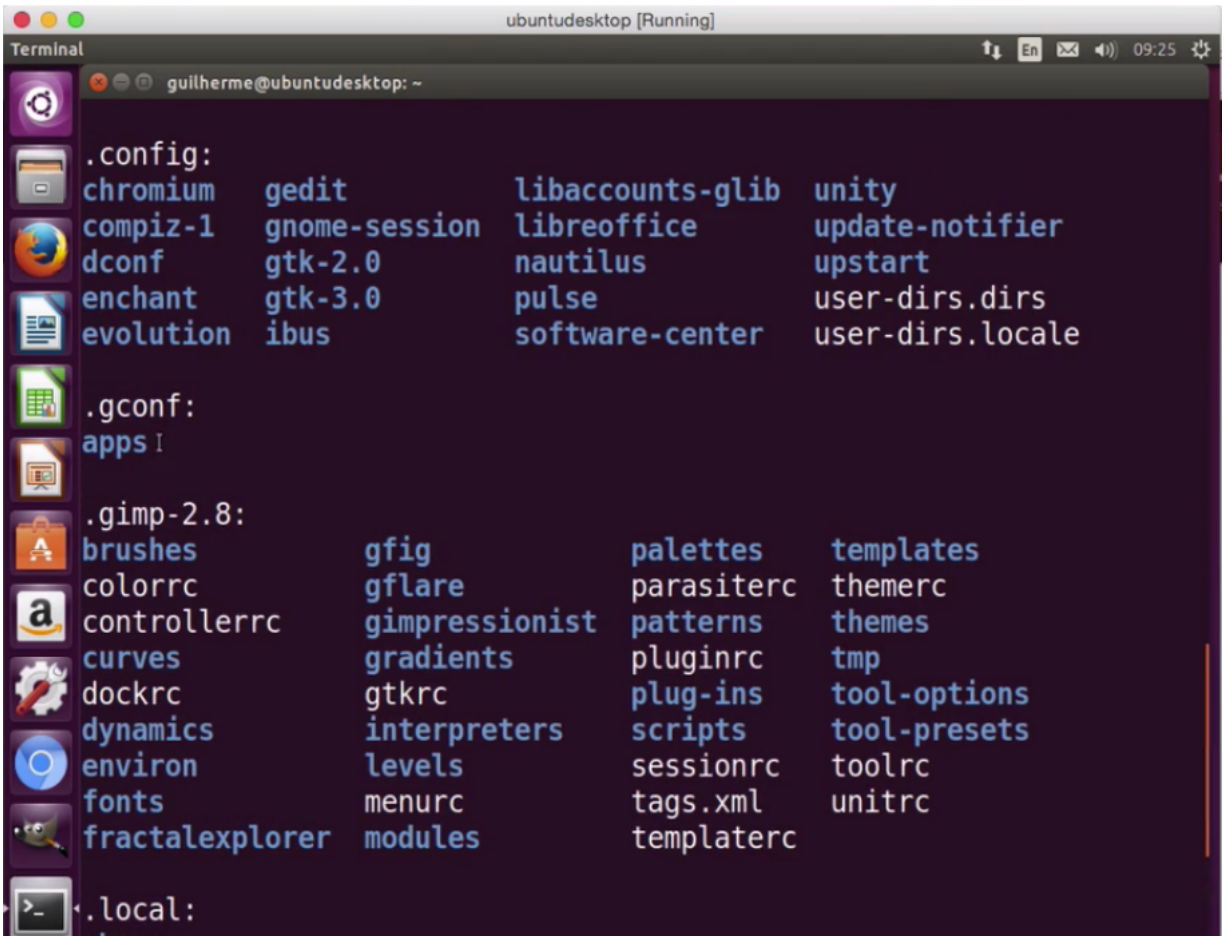
O `*.txt` é substituído por `texto10.txt texto1.txt texto2.txt texto3.txt`. No final estamos executando `texto10.txt texto1.txt texto2.txt texto3.txt`. Repare que não é o `ls` que interpreta o asterisco, é o próprio *shell* que interpreta o asterisco e executa isso. Vamos voltar ao diretório inicial e vamos dar um `ls`:

```
~/Documents$  ls
arquivos.zip        examples.desktop        loja        Pictures        Templates
Documents       falha       mostra_idade Public     Videos
Downloads       falha~      mostra_idade~       sucesso     zip
Downloads       help        Music       sucesso~
```

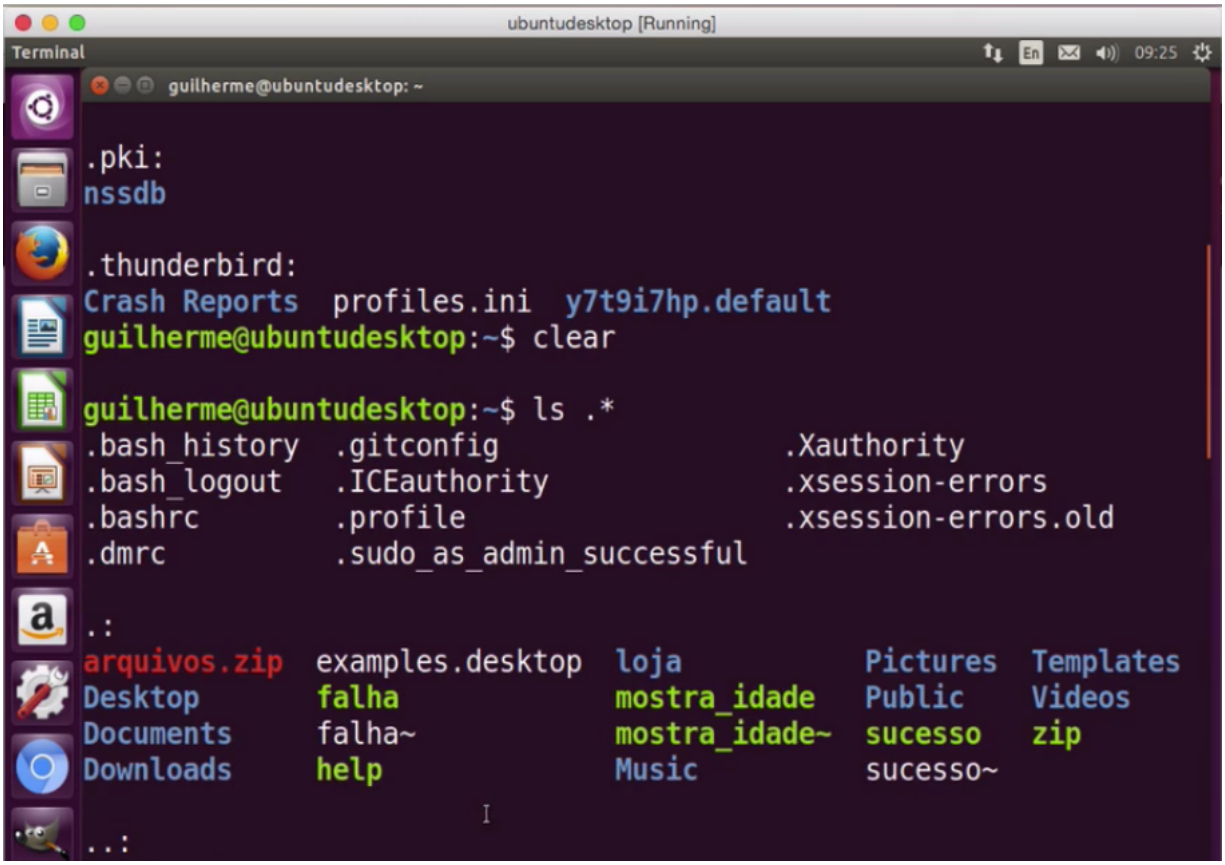
O `ls` traz tudo isso. Agora, o que acontece se digitarmos junto ao `ls` um asterisco? Por exemplo, `ls *`. O que será que ele faz?



Ele traz um monte de coisas. O `ls` expandiu o `*` para o `Desktop` , `arquivos.zip` , `Desktop` , `falha` , `falha~` ... Ele expandiu para todos os arquivos que estavam na lista. Ele chamou um comando gigante e o que ele resultou? Trouxe o nome dos arquivos, trouxe *Documents* e os arquivos que estão lá dentro, trouxe o `Desktop` e os arquivos que estão lá e etc. Mas ele não entrou nesses diretórios, entrou em apenas um nível entrou no `ls Music` , no `ls Pictures` , no `ls Templates` e etc...



Ele executou esse `ls` grandão e tudo de uma vez só. Temos que tomar cuidado, se estamos fazendo esse `ls*` parece que ele traz todos os arquivos, mas na verdade não traz . Alguns arquivos no *Linux* são chamados de invisíveis, são os arquivos que começam com um `.` . Para ver esses arquivos temos que colocar um `.` , isto é, deixar explícito o `.` . Temos que escrever, por exemplo, `ls.*` aí, teremos os arquivos invisíveis, tais como o `.geconf` , o `.gimp-2.8` e vários outros arquivos que começam com o `.` . Ele mostrará tanto os arquivos do diretório atual quanto dos diretórios filhos, um nível, pois ele vai fazer o `ls` desse diretório também. Só que, se quisermos observar os arquivos invisíveis do diretório atual, temos que escrever `ls.*` para pegar os arquivos invisíveis e realmente, poderemos visualizar que todos os arquivos que iniciam com `.` são mostrados, os que iniciam sem ponto, não são mostrados.



Mas repare que o `ls.*` é um pouco estranho pois ele entra nos diretórios. Mas, em geral, não queremos entrar no diretório, queremos listar os arquivos. O `ls -a` mostra todos os arquivos, inclusive os que começam com ponto. Não costumamos usar o `*` na hora de fazer o `ls` , e podemos ter uma pegadinha aqui, pois fazemos um `ls.*` ou um `ls*` e pode ser que tenhamos um resultado diferente do que o esperado.

A grande sacada é lembrar que o `*` é interpretado pelo *shell* e expandido pelos vários nomes que temos.

Primeiro, vamos entrar no diretório `Documents` através do `cd Documents` e vamos dar um `ls` . Teremos o seguinte:

```
~/Documents$ ls
curriculum texto10.txt      texto1.txt      texto2.txt      texto3.txt
```

E agora, se dermos um `echo *` o que acontece?

```
~/Documents$ echo *
texto10.txt      texto1.txt      texto2.txt      texto3.txt
```

Ele transforma o `echo *` na seguinte resposta: `texto10.txt texto1.txt texto2.txt texto3.txt` O asterisco é o básico do `globbing` , ele pega tudo que tem aquele padrão e expande no comando. Então, se executamos `echo Possuo os seguintes arquivos *` ele interpreta o `asterico` e joga tudo o que tínhamos achado anteriormente para depois da mensagem e mostra o seguinte:

```
~/Documents$ echo  echo Possuo os seguintes arquivos *
Possuo os seguintes arquivos curriculum texto10.txt      texto1.txt      texto2.txt      texto3.txt
```

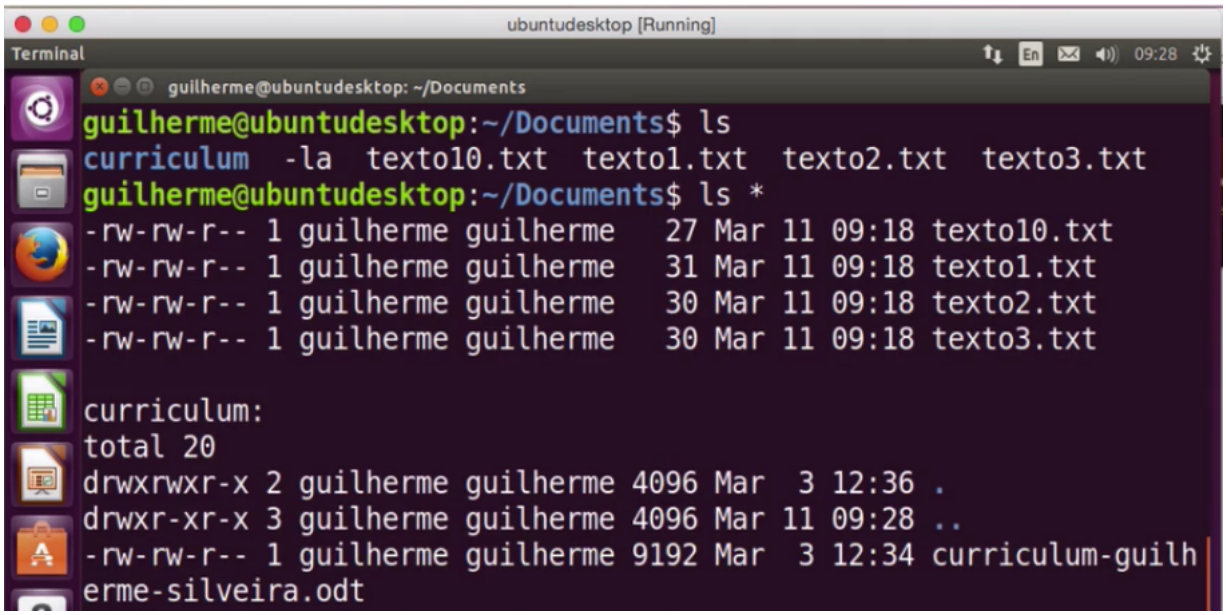
Ele troca o asterisco por todos os nomes. Lembra que o asterisco é interpretado antes de executarmos um comando? Por isso temos um caso muito estranho, por exemplo, se criarmos um arquivo chamado `gedit` , vai abrir o editor e nele escreveremos `Arquivo simples` , salvaremos isso no *diretorio documents* e chamaremos esse arquivo de `-la` .

Vamos limpar essa tela usando o `clear` .

Vamos relembrar o que o `ls` nos diz? Ele traz todos os arquivos para nós:

```
~/Documents$ ls
curriculum  -la     texto10.txt     texto1.txt      texto2.txt      texto3.txt
```

O que acontece se dermos um `ls *` ? Ele transforma tudo, ele substitui o asterisco por todos os arquivos:



O que ele executa então? Ele executa `ls curriculum -la texto10.txt texto1.txt texto2.txt texto3.txt` . Foi isso que ele executou, pois, primeiro ele interpretou o `*` e depois executou o comando, mas repare que o `ls` , na hora que ele interpretou o `*` , um dos nomes dos arquivos era `-la` . Ao em vez de listar o arquivo `-la` ele interpretou isso como se fosse uma opção do `ls` e aí, mostrou o `ls` no formato de lista e inclusive arquivos invisíveis, arquivos ou diretórios que começam com `.` .

O `ls *` realmente interpreta o `*` primeiro, inclusive se o `asterisco` for menos alguma coisa, o nosso comando `ls` vai pegar isso, por exemplo, `-la` e vai usar como opção. Lembra-se que no `ls` a opção não precisa vir em primeiro lugar? Em geral um comando é `nome do comando opções parâmetros` . Repare que no `ls` misturamos o nome, com o parâmetro currículo e opção -la e o `ls` aceita.

Cada comando vai ter o seu padrão, alguns vão aceitar isso e outros não. A tendência em geral é, entretanto, não aceitar, se existe um padrão mais fixo e se tem uma regra é ela que vai ser obedecida. Mas, nesse sentido o `ls` é bastante flexível. As opções serão aplicadas a todos aqueles que estão sendo passados como parâmetros.

Então, muito importante, apareceu um asterisco em uma prova ou, se você for utilizar em seu dia a dia, lembre-se ele será interpretado e substituído por todos os arquivos do diretório. E esse resultado vai ser utilizado no comando que será executado agora. Só depois disso é que teremos o comando sendo executado.

Vamos ver outras opções de `globbing` mais adiante.

Avançando no globbing e no quoting

Vimos uma opção de *globbing*, uma maneira de expandir e interpretar um asterisco em diversos valores, mas existem outros padrões interessantes. Por exemplo, vamos entrar no diretório `Documents`, usando o `cd Documents` e vamos usar um `ls`:

```
> cd Documents
~/Documents$ ls
curriculum     -la     texto10.txt     texto1.txt     texto2.txt     txto3.txt
```

E se quisermos listar todos os arquivos que se chamam "texto + alguma coisa + .txt"? Poderíamos utilizar um `cat texto*.txt` assim, queremos ver tudo que começa com texto, tem qualquer coisa no meio e, por fim, possui um `.txt`. Vamos executar isso e verificar o que acontece:

```
cat texto*.txt
Conteudo do decimo arquivo
O conteudo do primeiro arquivo
O conteudo do segundo arquivo
Conteudo do terceiro arquivo
```

Ele traz o conteúdo dos quatro arquivos.

O asterisco pega diversos caracteres, mas e se quiséssemos apenas um carácter? Para isso, utilizamos apenas um ponto de interrogação, `?`, depois da palavra `texto`. Ficaremos com o seguinte `cat texto?.txt`. Com isso, estamos dizendo que pode ser preenchido com uma coisa qualquer, um número, uma letra... Executando isso teremos o seguinte:

```
O conteudo do primeiro arquivo
O conteudo do segundo arquivo
Conteudo do terceiro arquivo
```

Ele devolve apenas três arquivos.

Cuidado!

O *globbing* do *bash*, do *shell* não é uma expressão regular, embora, ele se assemelhe de maneira remota a uma expressão regular. Assim, o asterisco, o carácter anterior, e o ponto de interrogação não significam uma expressão regular. Como existem alguns caracteres em comum, que são parecidos e similares, temos a sensação de que é uma expressão regular ou que funciona mais ou menos como uma expressão regular.

Mas, não tem relação com uma expressão regular. O asterisco, `*`, significa qualquer coisa quantas vezes forem. O ponto de interrogação, `?`, significa um elemento, qualquer que seja, mas uma única vez.

Essa abordagem é bem comum e bastante usada quando queremos ler vários arquivos de `log`, por exemplo, queremos ler vários arquivos de `log` que foram salvos do dia 1 ao dia 9. Vamos limpar a tela utilizando o `clear` e dar um exemplo, queremos ler todos os arquivos de `log` do dia 1 ao dia 9, para isso, digitaremos `log-2016-05-0?`. Ele pega todos os arquivos que possuem esse padrão e nos mostra:

```
~/Documents$ cat log-2016-05-0?.txt
cat:log-2016-05-0?.txt: No such file or directory
```

Vamos fazer esse exemplo? Vamos abrir o `gedit`:

Após abrir o `gedit`, escreveremos nele `log 1` e salvaremos com o nome "log-2016-05-01.txt" no `Diretorio documents`.

Então, se executarmos o `cat log-2016-05-0?.txt`, ele nos responde o seguinte:

```
~/Documents$ cat log-2016-05-0?.txt
log 1
```

Ele nos contesta com o conteúdo do arquivo, isto é, com `log 1`.

Vamos salvar diversos outros arquivos?

Abrindo o editor, mais uma vez, podemos escrever `log 2` e salvamos com o nome `log-2016-05-02.txt`. Podemos salvar também um arquivo com o escrito `log 3` e com o nome de `log-2016-05-03.txt` e salvaremos também o `log 10` com o nome `log-2016-05-10.txt`. E por fim, salvaremos um último arquivo que chamaremos "0t", seu nome completo será `log-2016-05-0t.txt` e escreveremos em seu arquivo `log t`.

Voltando para o nosso terminal, se digitarmos `cat log-2016-05-0?.txt` teremos o seguinte:

```
~/Documents$ cat log-2016-05-0?.txt
log 1
log 2
log 3
log t
```

Ele nos responde com o `log 1` , `log 2` , `log 3` e `log t` . Faz todo sentido termos o `log t` , pois ele é um carácter qualquer, seja um algarismo, seja qualquer carácter válido para nós. Isso demonstra que ele está funcionando, normalmente.

Mas e se quiséssemos que ele trouxesse somente os caracteres que fossem números. Para fazer isso poderíamos digitar entre colchetes os números 1, 2 e 3. Podemos digitar `log-2016-05[123].txt` e teremos o seguinte:

```
~/Documents$ log-2016-05[123].txt
log 1
log 2
log 3
```

Poderíamos digitar também `log-2016-05[12].txt` e teremos como resposta:

```
~/Documents$ log-2016-05[12].txt
log 1
log 2
```

E teremos apenas o log 1 e 2.

E se colocarmos o `log-2016-05[1].txt` ? Teremos o seguinte:

```
~/Documents$ log-2016-05[1].txt
log 1
```

Teremos apenas o `log 1` .

Se colocarmos `log-2016-05[123].txt` estaremos dizendo para trazer a classe de caracteres que temos aqui, o `1` ou o `2` ou o `3` .

E se escrevermos `log-2016-05[1-3].txt` ?

O que estamos dizendo é que buscamos os logs do 1 até o 3.

E o que estaríamos dizendo se escrevêssemos `log-2016-05[0-9].txt` ?

Estaríamos dizendo do `0` até o `9` , isto é, do `log-2016-0` até `log-2016-9` .

Analisamos o asterisco, o ponto de interrogação e os colchetes. O asterisco indica qualquer quantidade de caracteres, de elementos válidos, naquela posição. Temos o ponto de interrogação, o `?` , que é um único elemento naquela posição. E, por fim, temos os colchetes `[]` que indica qualquer um dos elemento que estão dentro dos colchetes, por exemplo, podemos dizer apenas "1 *e* 2" e usamos, para tanto, `[12]` . Podemos dizer do "1 *até* o 9" e para isso usamos o `[1-9]` .

Vamos dar um `clear` .

Um outro cuidado que temos que tomar é quando executamos um comando e digitamos `cat`, damos um espaço e digitamos o restante, `log-2016-05*`. Teremos o seguinte:

```
~/Documents$ cat log-2016-05*
log 1
log 2
log 2
log 3
log t
log 10
```

Repare que o espaço que existe entre `cat` e `log` não precisa, necessariamente, ser apenas um espaço, podem ser vários. Um espaço no *shell* pode ser um ou pode ser vários, isso é indiferente, no momento em que for interpretar o comando.

```
~/Documents$ cat       log-2016-05*
log 1
log 2
log 2
log 3
log t
log 10
```

Existem casos que podemos inserir, inclusive, uma quebra de linha. Aqui não, pois se colocarmos uma quebra de linha após o `cat` ele executa isso, por não estar esperando mais nada. Então, aqui, a quebra de linha não funciona!

Podemos usar a quebra de linha no *bash* de outra maneira, podemos digitar `cat`, dar um espaço e usar uma barra invertida `\` e podemos digitar outras coisas na linha seguinte. Lembra-se do `PS2`? Esse é o `PS2`. Podemos digitar outras coisas e então, executar. Teremos o seguinte:

```
~/Documents$ cat  \
> log-2016-05*
log 1
log 2
log 2
log 3
log t
log 10
```

Se usarmos o `history`, o para cima, o comando que ele fala que executou é o `log-2016-05*`. Só que digitamos ele em duas linhas, usando a barra invertida.

Então repare, da mesma maneira que temos o carácter que é o ponto de interrogação e que é expandido para qualquer carácter que está lá dentro, temos também o asterisco que é expandido para vários elementos que estão lá dentro. Falamos carácter, mas na verdade ele não significa carácter de "A até Z" ele quer dizer qualquer elemento.

Além do asterisco temos também o colchete e por vezes acabamos quebrando o comando usando a barra invertida, pois o espaço é indiferente para nós.

O que mais podemos fazer nesse caso? Podemos querer pegar todos os arquivos que tenham uma letra naquele lugar, para isso podemos escrever `cat log-2016-05-0[A-Z].txt`. Com isso estamos dizendo que queremos um carácter qualquer de A até Z. Vamos observar o que ele responde:

```
~/Documents$ cat log-2016-05-0[A-Z].txt
log t
```

Ele nos respondeu o `log t`, ele pegou o `t`. Mas, você pode estar se perguntando o que aconteceria se as letras estivessem em minúsculo, ao em vez de [A-Z] fosse [a-z]?

```
~/Documents$ cat log-2016-05-0[a-z].txt
log t
```

Ele também pegaria o `log t`.

Então, ele pega os arquivos que sequem esse padrão, isto é, a classe de caracteres de "a até z". Repare que independentemente de ser maiúsculo ou minúsculo ele pega o `log t`. É claro que quando falamos de numerais, essa questão do maiúsculo ou minusculo é indiferente.

Poderíamos também escrever em forma de negativa, por exemplo, gostaríamos de tudo aquilo que não é número. Escreveríamos `cat log-2016-05-0[!0-9]`. O ponto de interrogação serve para indicar a negativa, isto é, que queremos aquilo que não são números. Vamos observar o que acontece escrevendo isso?

```
cat log-2016-05-0[!0-9]
log t
```

Ele vai trazer apenas o `t`.

E agora, temos que tomar cuidado com outras questões também. E se tivermos o arquivo que é o do `log t` e decidirmos criar um outro arquivo que tem o mesmo nome de `log`, apenas com ou `ç` no meio, `log-2016-05-0ç.txt`. Vamos salvar isso e escrever `log cedilha ç`.

Vamos tentar executar o `cat` do que não é número:

```
~/Documents$  cat log-2016-05-0[!0-9].txt
log cedilha ç
log t
```

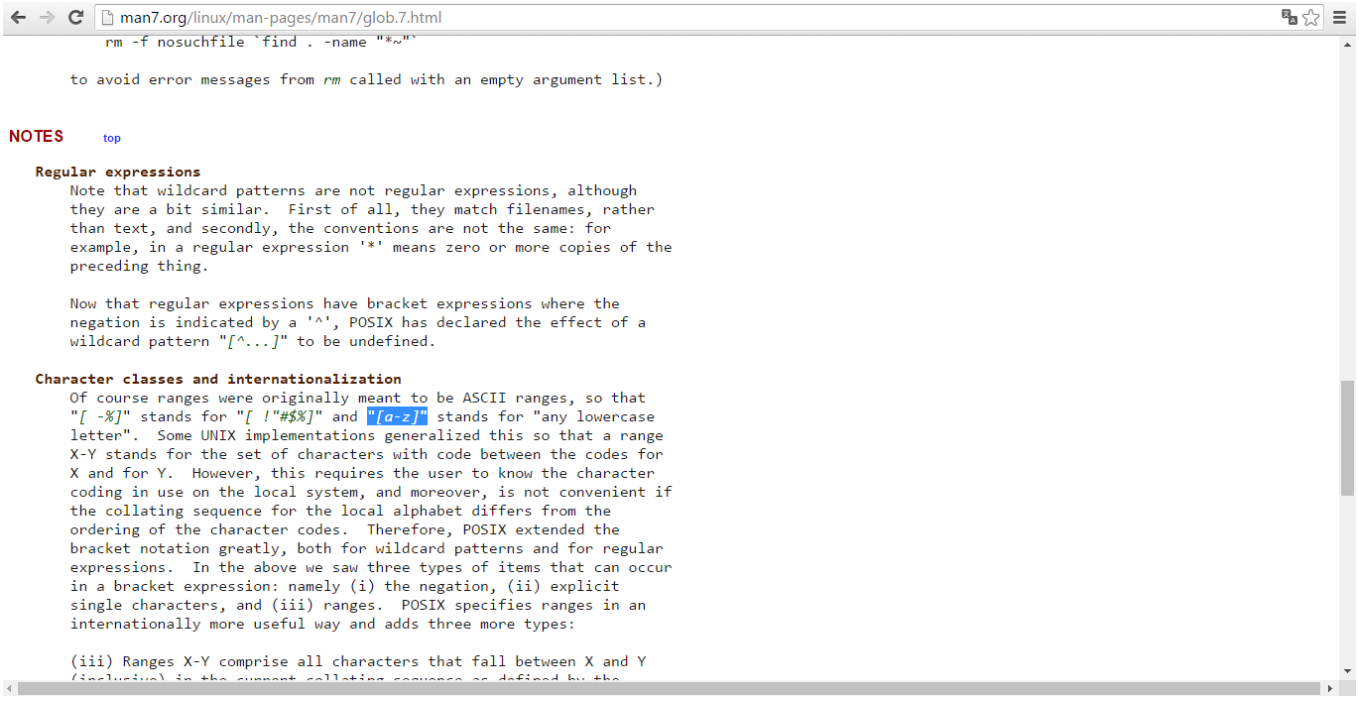
Ele traz para nós o `log cedilha ç`. E se fizermos o `cat` de `[A-Z]`?

```
~/Documents$ cat log-2016-05-0[A-Z].txt
log cedilha ç
log t
```

Ele traz novamente o `log t` e o `log cedilha ç`.

Temos que tomar cuidado quando colocamos a classe de `a-z`, seja em minúsculo ou maiúsculo. Temos que entender o que significa de "A até Z" no bash, no *shell*.

Podemos procurar *"shell globbing caracter class"*. Podemos acessar a seguinte página e olhar a documentação:



Na documentação podemos buscar informações encontramos que ao utilizarmos o `[a-z]` minúsculo ele nos fornecerá qualquer letra minúscula. Mas, tendo executado das duas maneiras, ambas não tem funcionado? E, inclusive, ao buscar de "a até z" ele nos trouxe o cedilha.

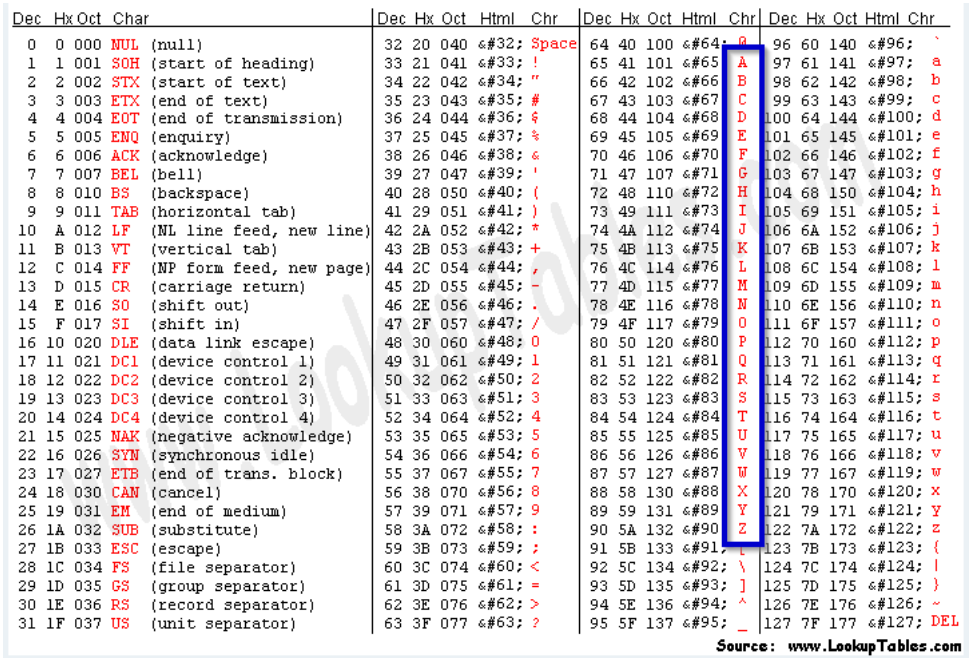
O que o *Ubuntu Desktop* está fazendo no nosso *shell*?

Repare que na documentação é mencionado que algumas implementações do *Linux* generalizaram essa definição de letras de "x até y" para trazer os caracteres com código entre "x e y". Mas isso requer que nós, quando estivermos executando um comando, conheçamos qual é o `in code` que está sendo utilizado nesse instante. Se ele estiver utilizando o padrão, saberemos que é da letra "a até z" de acordo com a tabela `ASCII`. Portanto, não será de acordo com a tabela de `in code` que nós estivermos utilizando. Então o problema está que cada *Linux* ou cada *shell* pode estar utilizando um `in code` distinto. Pode ser um `in code` brasileiro ou de qualquer outro lugar. No caso, se alguém usar o `[a-z]` podemos ter um resultado distinto do resultado de outra pessoa que usa o `[a-z]`.

Para a prova o importante é sabermos o padrão. Assim, o padrão diz que `[a-z]` em letras minúsculas traz letras minúsculas e `[A-Z]` escrito em letras maiúsculas traz letras maiúsculas. Assim, no caso do cedilha ou de acentos, isso não é trazido junto.

Então, por mais que no nosso *Linux*, rodando o *Ubuntu Desktop*, seja interpretado que de "a até z" maiúsculo e minúsculo pode trazer qualquer carácter válido da língua portuguesa temos que nos ater ao conteúdo da prova, portanto, é importante lembrar que na definição do *Linux* sobre o comando *globbing* que diz que de "a até z" vai apenas de "a até z" na tabela *ASCCI*.

Vamos visualizar a tabela *ASCII* encontrada no site www.asciitable.com:



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Vamos observa da letra "a" até a letra "z", é apenas isso que é contemplado e que conta.

Para a prova é importante sabermos disso, isto é, que dentro disso não entra acento e tampouco cedilha. Ainda que, na prática, aqui tenha pego.

O colchetes serve para essa classe de caracteres e números. No padrão os caracteres não pegam acentos ou cedilhas e maiúsculo e minúsculo é sensitivo, assim como no *Linux* em geral.

E se quiséssemos colocar dois padrões totalmente diferentes. Por exemplo, quando pedimos o `ls` temos dois padrões:

```
~/Documents$ l
curriculum              log-2016-05-03.txt~     log-2016-05-10.txt~
-la                     log-2016-05-0ç.txt      texto10.txt
log-2016-05-01.txt      log-2016-05-0ç.txt~     texto1.txt
log-2016-05-02.txt~     log-2016-05-0t.txt~     texto3.txt
log-2016-05-03.txt      log-2016-05-10.txt
```

Temos os arquivos que iniciam com `log` e os arquivos que iniciam com `texto`. Então, queremos pegar o conteúdo de todos os arquivos que terminam com `1.txt`. Escreveremos `cat *1.txt` e nos será respondido:

```
cat *1.txt
log 1
O conteudo do primeiro arquivo
```

E se quiséssemos pegar algum outro tipo de padrão? Por exemplo, dois padrões bem diferentes? Tudo o que começa com `log-2016-05-` e que termina com `1*` que significa, tudo o que termina com `1` e qualquer coisa e ainda, queremos pegar também, todos os arquivos que começam com `texto`, seguidos de qualquer coisa (`?`) e tendo no fim `.txt`. Para dizer que queremos ou o primeiro padrão ou o segundo, colocamos isso dentro de chaves e teremos o seguinte:

```
cat {log-2016-05-1*, texto?.txt}
log 10
O conteudo do primeiro arquivo
```

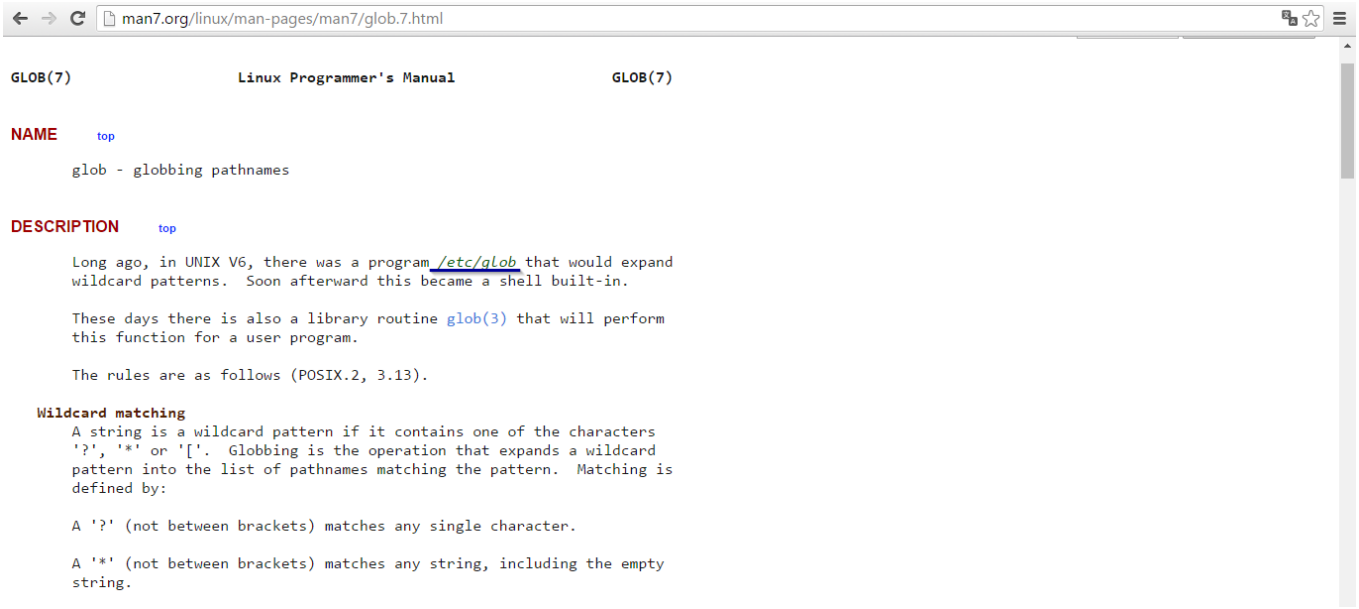
```
O conteudo do segundo arquivo
Conteudo do terceiro arquivo
```

Ele traz os dois padrões, os que começam com o `log` e o número `1` e o que começa com `texto` e um único número e `.txt`. As chaves são usadas para dizer "ou", isto é, elas dizem "ou" o primeiro padrão, "ou" o segundo padrão.

O que vimos do *globbing*?

- O asterisco é expandido para tudo que tenha arquivo com a limitação do `.*`;
- O ponto de interrogação serve para um único elemento, pode ser um carácter, um número ou qualquer outra coisa. Por força de costume acabamos falando um carácter, mas pode ser um dígito, um sinal ou qualquer outra coisa;
- O espaço na linha não faz diferença;
- O barra invertida é uma maneira de falar que o comando continua na próxima linha;
- O colchete quer dizer que é uma classe de carácteres, por mais que no *Ubuntu Desktop* ele pegue maiúsculo e minúsculo e até mesmo cedilha ou outros caracteres. ***No padrão do* Linux `[a-z]` *seriam letra minúsculas e* `[A-Z]` *são letras maiúsculas*** e ignoram-se os acentos, cedilhas e outras coisas, que não vão ser conectados com isso.
- Vimos que utilizando as chaves, `{}`, podemos escrever o padrão `{primeiro*, segundo*}` que significa, tudo o que começa com primeiro e tudo o que começa com segundo e pegando tudo isso podemos fazer alguma coisa com todos eles.

o *globbing* trabalha com isso e na documentação podemos perceber que ele trabalha com os `pathnames`, os caminhos no linux. Repare que isso era feito através de um programa o `/etc/glob`.



Atualmente, se buscarmos o programa `glob` não o encontramos. Para buscar isso digitamos `glob` e teremos:

```
~/Documents$ glob
No command 'glob' found, did you mean:
    Command 'glom' from package 'glom'  (universe)
    Command 'glob2' from package 'glob2' (universe)
    Command 'globs' from package 'globs' (universe)
    Command 'dglob' from package 'debian-goodies' (main)
glob: command not found
```

Se tentamos buscar `glob` através do `type glob` teremos a seguinte resposta:

```
~/ Documents$ type glob
bash: type: glob: not found
```

Também não o encontramos.

O `glob` faz parte do `Linux`, isto é, ele não é mais um programa separado, ele faz parte do *shell*. Ele interpreta aplicando o `glob` para depois executar um comando.

É importante sabermos que não é uma expressão regular, pode-se fazer vários testes, testar diversas variações e no dia a dia isso é extremamente útil para nós. Para não termos que digitar diversos nomes, podemos utilizar sempre padrões para trabalhar com os arquivos.

Ponto e vírgula

Vimos que o *globbing* serve para pegar um padrão e expandir esse padrão antes que nós executemos um comando. Assim, usando o *globbing*, temos o poder de, com poucas letras ou palavras, executar um comando para diversos `paths`.

Claro, existem momentos em que não vamos querer utilizar isso. Um exemplo bastante simples é quando queremos mostrar, apenas, uma mensagem de `bem-vindo`. Para isso, escrevemos `echo * Bem Vindo *` e colocamos um asterisco antes e depois da mensagem que queremos enviar. Ele vai nos mostrar o seguinte:

```
> echo * Bem Vindo *
arquivoz.zip Desktop Documents Downloads examples.desktop fal
ha falha~help loja mostra_idade mostra_idade~Muisc Pictures Public suce
sso sucesso~ Templates Videos zip Bem vindo arquivos.zip Desktop Do
cuments Dowloads examples. desktop falha falha~help loja mostra_iddade mostra_id
ade~Music Pictures Public sucesso sucesso~ Templates
Videos zip
```

E ao em vez de mostrar o texto "Bem vindo", ele mostra um monte de coisas e vários nomes de arquivos.

Tem vezes que não queremos que para um pedaço do comando, que se execute o *globbing*, isto é, que ele se expanda. Para não expandir esse carácter podemos dizer para ele escapar, isto é, para que se ignore o carácter asterisco, uma vez que ele não é um carácter especial. Para dizer isso utilizamos uma barra invertida, `\`. A barra invertida fala que o próximo carácter não deve ser interpretado como ele costuma ser interpretado no *bash*.

Teremos o seguinte:

```
> echo \* Bem vindo * \
* Bem Vindo *
```

Como podemos observar o comando estará funcionando normalmente. O barra invertida `\` afirma que o próximo elemento que está por vir, não deve ser interpretado como o *bash* costuma interpretá-lo.

Por isso, no caso do `\* Bem vindo * \`, o asterisco não é interpretado e, por isso, não é expandido. Como também, em outro caso já exposto aqui, o caso do `ls` e do da barra invertida, `\`, que é utilizado para indicar que continua na próxima linha. Isto é, o próximo elemento não deve ser interpretado como uma nova linha, ele deve ser visto como um nada, como um espaço em branco. Mais uma vez utilizamos a barra invertida, `\`, ela serve para dizer que o próximo elemento não deve ser interpretado pelo *bash* como ele costuma interpretá-lo. Ele deve ser interpretado como um elemento solto, simples, sem

segundas intenções. No momento em que digitamos o `ls` nossa intenção não era executar o comando, era apenas continuar na próxima linha:

```
> ls \
- la
```

O barra invertida é o que chamamos de um `scape caracter` . É um carácter que escapa, que foge do comportamento padrão. No asterisco, no ponto de interrogação, em tudo onde tivermos um carácter especial em nosso *shell*. Como por exemplo, a quebra de linha, a barra invertida e mais um "Enter", ou a barra e um asterisco e etc. Podemos escapar de todos esses casos utilizando a `\` .

Mas, existe, ainda, outra maneira de escapar!

Podemos dizer, ainda, `echo * Bem Vindo *` e podemos falar para o *bash* que `* Bem Vindo*` é um único parâmetro e que ele não deverá ser interpretado pelo *bash*. Para indicar isso, podemos colocar dentro de aspas simples o texto `*'Bem vindo' *` .Colocando isso entre aspas simples, estamos dizendo que isso não pode ser interpretado pelo *bash*, pelo *shell*. Ele não pode ser interpretado pois é um texto texto.

É por isso que quando digitamos `echo 'Bem vindo*,` e damos um "Enter", ele continua a digitar o texto na próxima linha:

```
> echo 'Bem vindo,
> Guilherme
```

Por que isso ocorre? Isso ocorre, pois, esse "Enter" é um "Enter" normal, ou seja, é uma quebra de linha normal, não é para executar o comando. Enquanto não fecharmos as aspas simples, não terminamos o nosso parâmetro e podemos dar vários "Enter" que continuaremos dentro desse parâmetro. Quando colocarmos a outra aspas simples que falta, aí sim, estaremos finalizando o parâmetro e quando dermos um "Enter" após a aspas teremos, `Bem vindo` , e na outra linha, `Guilherme` . Observe:

```
> echo 'Bem vindo,
> Guilherme
>
>
>
>
>
>
>
> '
Bem vindo,
Guilherme
```

Observe também que além de executar o texto, "Bem vindo, Guilherme" ele executa também os "Enter" que digitamos, pois, eles também fazem parte desse nosso parâmetro.

A aspas simples, assim como as aspas duplas, servem como uma maneira de *quoting*. Observe o uso das aspas duplas com a mesma função:

```
> echo "Bem vindo
>
>
>
>
> "
Bem vindo
```

O *quoting* serve para fugirmos do *globbing*. Além do *scape caracter* podemos utilizar também o *quoting*. Vimos que aspas simples e aspas duplas, possuem, em geral, nessas situações, o mesmo significado. E, serve como uma maneira de fugir do *globbing* e falar para o *bash*, por favor, não interprete isso de uma maneira a expandir isso. Não interprete esse "Enter" como você deveria "interpretar". Ele simplesmente é um "Enter" solto, é uma quebra de linha dentro do parâmetro, é um asterisco dentro do parâmetro. Ou seja, passe o asterisco para dentro do programa, não interprete e não expanda esse asterisco.

Tanto a barra invertida é um *scape caracter* quanto as aspas simples e duplas. A barra invertida é um *scape caracter* apenas do próximo carácter, as aspas duplas falam que das primeiras aspas duplas até as segundas aspas duplas, ou, que das primeiras aspas simples até as segundas aspas simples, isso tudo é um parâmetro e tem que ser interpretado de uma vez só e deve ser passado para o programa da maneira como ele estiver. E esse é o nosso *quoting*, assim, *scape caracter* e *quoting* são extremamente importantes para que não executemos diversos comandos que, afinal, não queríamos executar.

Vamos dar um exemplo?

Vamos abrir o editor e vamos criar um arquivo que se chamará `log completo.txt`. Salvamos isso e escrevemos `meu log completo`. Repare que o nome do arquivo é "log completo" e que existe um espaço entre "log" e "completo". Se tentarmos ler esse arquivo através do `cat`, o que será que vai acontecer?

```
> cat log completo.txt
cat: log: No such file or directory
cat: completo.txt: No such file or directory
```

Ele vai executar o comando `cat` para o argumento `log` e para o argumento `completo.txt` e concluirá que não é possível encontrar nenhum deles. Era para ter interpretado `log completo.txt` como um todo, como um único parâmetro. Era para o *bash* não ter interpretado o espaço como um espaço, era para ter interpretado `log completo.txt` como um único parâmetro e passado esse espaço para o comando.

Vamos colocar `log completo.txt` entre aspas duplas e simples e ver o que acontece:

```
cat 'log completo.txt'
meu log completo
cat "log completo.txt"
meu log completo
```

Agora sim, ele passa o espaço para o comando.

Podíamos também ter utilizado o carácter de *scape*, o `\` e um espaço. Digitando `cat log\ completo.txt`.

```
> cat log\ completo.txt
meu log completo
```

Inclusive, se dermos um `cat` e começarmos a escrever `log` e apertarmos um "Tab" ele usa o carácter de *scape*. Aliás, é bem comum que esse carácter apareça.

É claro, na prática evitamos arquivos que contenham espaço no nome e, também, buscamos não utilizar arquivos que tenham caracteres diferentes no nome. E evitamos isso por que? Porque queremos evitar que um *script* ou programa mal feito quebre porque decidimos colocar um espaço no nome do arquivo. Só por causa disso. Por mais que o sistema operacional permita nome de arquivos com espaço, nós acabamos evitando para, justamente, não termos que sofrer mais adiante. Então, de repente, podemos rodar um arquivo avançado e ele realiza diversas quebras e não conseguimos descobrir o por quê disso acontecer e até sabermos que ele não suporta arquivos com espaço no nome. É bem comum que evitemos arquivos com espaço no nome, asteriscos, pontos de interrogação ou caracteres em geral, pois estes, podem ser interpretado pelo *bash*, pelo *shell*, e buscamos, portanto, evitá-los. Mesmo que determinados caracteres sejam válidos como nomes de arquivos. Veremos esse tema mais adiante quando nos debruçarmos sobre o nome dos arquivos.

Repare, como fazemos para fugir do *globbing* ou *scape caracter* que é a barra invertida `\` ou colocamos as *double quotes*, `""` , que são as aspas duplas, ou as *simple quotes*, as aspas simples `''` . Então, as aspas servem para escaparmos do *globbing* e para, simplesmente, dizer: "passe esse conteúdo para o programa". Esse é o *quoting*.

Falta um último ponto que passaremos na execução de programas.

Um último exemplo é quando queremos executar dois programas.

Mas, já não aprendemos isso?

Quando usamos os dois "e" comerciais, `&&` que significam "e" ou as duas barras verticais `||` que significam `ou` . Não, queremos executar um programa e também queremos executar um segundo programa logo depois, independente de funcionar ou não. Como fazemos isso?

Usamos o ponto e vírgula, `;` . Então, podemos escrever, execute o `ls ; echo Listagem completa` . Quando escrevemos isso o que estamos dizendo é que queremos executar o `ls` e, logo em seguida, o `echo Listagem completa` . Teremos o seguinte:

```
> ls ; echo Listagem completa
arquivos.zip            falha                   loja                Public            zip
Desktop                falha                  mostra_idade        sucesso
Documents               help                    mostra_idade~        sucesso~
Dowloads                log completo.xt           Music                Templates
examples.desktop        log completo.txt~       Pictures            Videos
Listagem completa
```

O ponto e vírgula `;` serve para dizer que onde ele está é onde termina um comando que gostaríamos de executar e a partir dele começa o próximo comando que gostaríamos de executar. Inclusive, o espaço é opcional, o próprio ponto e vírgula define isso:

```
 ls ;echo Listagem completa
arquivos.zip            falha                   loja                Public            zip
Desktop                falha                  mostra_idade        sucesso
Documents               help                    mostra_idade~        sucesso~
Dowloads                log completo.xt           Music                Templates
examples.desktop        log completo.txt~       Pictures            Videos
Listagem completa
```

Isso quer dizer que se o ponto é vírgula é um carácter especial do *bash*, onde ele diz que aqui terminou uma linha, um comando a ser executado e a partir desse ponto temos um outro comando a ser executado. Isso quer dizer que tem situações onde não gostaríamos que o ponto e vírgula fosse interpretado pelo *bash*.

Por exemplo, podemos ter o seguinte:

```
> echo Minha mensagem; e minha outra mensagem
Minha mensagem
e: command not found
```

Com isso ele vai executar `echo Minha mensagem` , e vai tentar executar `e` , como se fosse um comando, mas ele não existe. O que deveríamos fazer nesse caso?

Poderíamos colocar um *scape caracter* e ele mostra a mensagem ou colocamos aspas simples e ele também mostrará tudo o que queremos:

```
> echo Minha mensagem\; e minha outra mensagem
Minha mensagem; e minha outra mensagem
> echo 'Minha mensagem; e minha outra mensagem'
Minha mensagem; e minha outra mensagem
```

Uma das duas maneiras vai fazer com que ele ignore o ponto e vírgula, `;` , como carácter especial do *bash* e mande isso para o programa do *echo* e o *echo* é que deverá lidar com isso. No caso, o `echo` só realiza a impressão. Então, utilizamos o ponto e vírgula, `;` para executar dois comandos em uma mesma linha.

É bastante feio realizar isso na prática. Tentamos evitar que isso aconteça em diversas situações, principalmente, no *bash* quando estivermos digitando os comandos. Mas, tem situações em que isso se encaixa e acaba aparecendo, principalmente em *scripts*.

As vezes quanto mais clareza, melhor, então, se puder evitar isso, melhor, se não der para evitar, não tem problema também.

Para nós é importante saber que o `;` permite que terminemos um comando e comecemos um novo logo depois, independente de ele ter funcionado ou não, ele simplesmente executa um ou o outro. Lembra-se do `zip` ?

Podemos digitar `zip` , um arquivo que não existe, um `;` e `echo Terminei` . Vamos executar isso:

```
> zip arquivos.zip  fehehehe ; echo Terminei
        zip warning:name not matched: fehehehe
zip error: Nothing to do! (arquivos.zip)
Terminei
```

Ele tenta executar e não funciona o `fehehehe` . Mas, ele imprimiu a mensagem `Terminei` . Se tivesse dado certo a primeira parte, se ele tivesse compactado um arquivo que existe, por exemplo, o arquivo chamado "Sucesso", também teríamos a mensagem impressa:

```
> zip arquivos.zip sucesso ; echo Terminei

updating: sucesso (stored 0%)
Terminei
```

Ele também mostra a mensagem `Terminei`. Então, independente do sucesso ou da falha do primeiro, do *exit status* do primeiro, ele executa o segundo, isto é, executa o que tiver do `;` para frente. Terminou o primeiro, ele executa o segundo.

Tem uma situação muito engraçada, onde ele não executa o segundo. Essa situação é a seguinte:

```
exit; echo Tudo bem
```

Se dermos um "Enter" nisso ele na verdade sai do terminal, então, acaba por aqui, nós fechamos o *shell*.

Revisando:

O ponto e vírgula (;) serve para dizer que vamos executar dois comandos, um antes do `;` e um depois do `;`. Poderíamos ter vários `;` separando outros comandos. Entramos, novamente, na questão do *scape*, que pode ser ou com aspas duplas, ou aspas simples, ou barra invertida.

Conclusão

Com isso, vimos toda a a parte básica do uso do *shell*. Vimos a sintaxe da linha de comando, a sintaxe tradicional até mesmo a definição de variáveis que são extensões do *bash*. Vimos variáveis, variáveis de *shell* e de ambiente, vimos o *globbing* e também o *quoting*. Falamos bastante sobre o *bash*, sobre diversos comandos que são *builtin* do *bash* e do *shell* e vimos comandos que não são, ,mas que são programas ou arquivos de *script*. Vimos o *echo*, o *history*, a variável de ambiente `PATH` e outras variáveis de ambiente importantes. Vimos *export* para exportar uma variável de *shell* como uma variável de ambiente. Vimos o *type* para saber se aquilo que estamos chamando é um *builtin* ou não é um *builtin*. Se está cacheado no *hash* ou se não está.