
DOMINANDO O
SED
um livro inacabado de
Aurélio Marinho Jargas

Escrevi este livro em 2002, entre os meses de Março e Setembro. É uma obra inacabada, que contém cerca de 60% do conteúdo planejado. Após este período a escrita parou e eu não tive mais vontade de continuar. A idéia era fazer um guia completo sobre o sed, uma bíblia que abrangesse todos os tópicos, dando dicas avançadas.

Hoje (Novembro de 2005) estou publicando o livro na Internet, com o texto intocado, como parou em 2002. Considero este um trabalho que não deve ficar restrito ao meu computador pessoal, pois mesmo incompleto pode ajudar outras pessoas a dominarem o assunto.

Se você é novato em sed, leia o sed-HOWTO (http://aurelio.net/sed/sed-HOWTO) primeiro. Terminada a leitura deste livro, domine também as Expressões Regulares (http://aurelio.net/er) para obter o máximo do sed.

Boa leitura!

índice

Instalação

Antes de mais nada, o site http://sed.sf.net é a fonte de informação mais atualizada sobre onde encontrar o sed para as várias plataformas suportadas. Consulte-o!

UNIX/Linux/Mac OS X

Espera aí, você vai me dizer que tem um UNIX/Linux e não tem sed instalado nele? Conta outra! &:D

Para atualizar uma versão existente, o procedimento é o mesmo do de outros programas, e isso pode requerer a instalação de um RPM, um pacote .deb ou baixar um .tar.gz e compilá-lo.

Windows/DOS

Usuários de Windows têm duas opções para usar o sed:

- SED.EXE no DOS
- sed no Cygwin

O sed nasceu no UNIX, mas várias versões do programa foram feitas para MS-DOS devido à falta de uma ferramenta semelhante nesse sistema. O nome genérico é `SED.EXE`, mas também pode ser encontrado como `SED.ZIP`, caso acompanhe documentação.

A instalação não tem segredo. Basta copiar o arquivo `SED.EXE` para um diretório que esteja no PATH do seu MS-DOS. Lugares comuns são `C:\Windows`, `C:\WIN` ou `C:\WINNT`.

Para quem gosta de UNIX/Linux, é aconselhável instalar o Cygwin (http://aurelio.net/cygwin), uma solução completa com ferramentas UNIX que roda no Windows. Com um clique, você tem acesso a uma janelinha preta com sed, bash, cat, tr, grep, find, vi e dezenas de outros programas. Vale conferir!

Endereço

Prepare-se. Respire fundo. Abra sua mente para receber conceitos novos, empolgantes e poderosos!

Para começar, deixemos claro que endereço não é um comando, mas sim o seu contexto. É o elemento que diz exatamente em quais linhas um comando, ou bloco deles, deverá ser aplicado.

O QUE FAZER ---> comando

ONDE FAZER ---> endereço

O endereço é algo simples se seus problemas forem simples. Mas quando se quer resolver algo realmente cabeludo com sed, o endereço também pode se tornar um monstro, daqueles que, de tão feios, quem olha pensa: "Tomara que eu nunca precise fazer manutenção nisso".

Para facilitar, podemos fazer uma analogia com os endereços de nossas casas, compostos por: nome da rua, número, cidade e CEP. O comando sed seria o carteiro que, baseado no endereço escrito na carta, deve encontrar o local correto.

Sabemos que quanto mais detalharmos o endereço, colocando todos os dados corretamente, mais fácil é para o carteiro encontrar o destino de nossa carta. Porém, na falta do CEP ou do número da casa, a carta não chegará ao seu destino por causa do endereço incorreto.

No sed é assim que acontece. Um endereço correto é indispensável para que o comando seja aplicado no lugar certo. Assim, torna-se essencial para o programador, ao escrever um comando sed, especificar o endereço com exatidão e sem ambigüidades.

> O domínio do endereço diferencia os gurus dos gafanhotos

Façamos um mergulho profundo no assunto para dissecar todas as formas de utilização e conhecer as soluções para os problemas mais rotineiros.

Há três tipos de endereços:

1. Endereço pelo número da linha
2. Endereço pelo conteúdo da linha
3. Endereço múltiplo, que abrange mais de uma linha

Um endereço sempre aponta para uma linha inteira

A primeira coisa que deve ser assimilada sobre o conceito de endereço é que ele sempre referencia uma linha inteira, não somente parte dela. Afinal, sendo o sed um editor orientado à linha, o endereço não poderia ser diferente.

Então, sempre que pensar em endereços, pense em linhas. Linhas inteiras.

Pode-se redefinir os três tipos de endereço e ilustrar como cada um deles é interpretado pelo sed:

1. "Quero a linha número N"
2. "Quero a linha que contém a palavra ABCDE"
3. "Quero todas as linhas que estejam entre essas duas linhas"

Como endereçar pelo número da linha

A maneira mais simples de se determinar um endereço é indicar diretamente sua posição no arquivo, informando ao sed o número da linha à qual se quer aplicar o comando. Por exemplo, para apagar a 5ª linha de um arquivo:

```
prompt$ sed '5 d' arquivo
```

Fácil, não? Observe que o espaço em branco entre o endereço e o comando é opcional, então também poderia ser `5d`. Mas para facilitar a visualização de quem é quem, vamos separá-los sempre.

E assim se endereça, numericamente, qualquer comando à linha desejada, desde que se saiba qual é sua posição no arquivo.

Como endereçar a primeira e/ou a última linha

Para endereçar a primeira linha, é barbada: 1. Nada mais. Vamos apagá-la?

```
prompt$ sed '1 d' arquivo
```

Moleza. Mas e a última? Nem sempre sabemos exatamente quantas linhas tem o arquivo, e é incômodo ter que usar outro programa para fazer isso antes de passarmos esse número ao sed.

Para resolver esse problema, temos um caractere especial de endereço: o cifrão `$`, que representa a posição da última linha do arquivo. Então é fácil apagar a última linha:

```
prompt$ sed '$ d' arquivo
```

Note que em sistemas UNIX o uso das aspas simples é obrigatório. Sem elas o shell tentaria expandir a variável `$d` e o sed receberia um comando vazio.

Como endereçar a linha que contém determinada palavra

Embora seja prático e fácil endereçar diretamente pelo número da linha, os problemas da vida real nos mostram que nem sempre temos o privilégio de saber exatamente em qual linha estão os dados procurados.

E quanto mais se trabalha com dados, mais se aprende que não é bom confiar em posições fixas porque elas raramente são fixas de verdade. Elas se mudam sem deixar telefone, email... &:)

Tendo em vista estas necessidades, também podemos definir endereços que sejam palavras ou trechos de uma linha.

Por exemplo, hoje no lanche comeremos frutas que... PÁRA! Não é odioso ter em livros técnicos esses exemplos "didáticos" envolvendo bananas, maçãs e abacaxis? Não é repugnante o autor tratar os leitores como crianças de primário? Seus problemas acabaram! Nada de exemplos com frutas por aqui.

Hoje, no lanche, comeremos **verduras** :) e como somos modernos e informatizados (nerds), colocamos a lista de compras num arquivo.

```
prompt$ cat verduras.txt
- alface
- cenoura
- couve
- nabo
- vagem
```

Mas, como hoje não é um dia bom para comer couve, vamos apagá-la da lista:

```
prompt$ sed '/couve/ d' verduras.txt
- alface
- cenoura
- nabo
- vagem
```

Note que, para especificar uma palavra como endereço, devemos colocá-la entre `/barras/`. Dentro dessas barras, pode-se colocar qualquer padrão que se queira casar para se encontrar uma linha. Pode ser uma ou mais palavras, símbolos ou expressões regulares.

A melhor maneira de se ler um endereço composto por um padrão é: "Nas linhas que contenham a palavra XXXX, aplique o comando YYYY".

> Note bem: "na**S** linha**S**".

Um endereço pode servir para mais de uma linha, então o comando será aplicado em todas as linhas que forem encontradas. Dessa forma, para apagar da nossa lista de verduras todas as linhas que tenham a letra "o", fazemos:

```
prompt$ sed '/o/ d' verduras.txt
- alface
- vagem
```

Apenas como curiosidade, este exemplo gera um resultado idêntico ao do comando:

```
prompt$ grep -v o verduras.txt
```

Como aplicar vários comandos em um mesmo endereço

Agora que já sabemos quais são os dois tipos básicos de endereço e como defini-los, vamos começar a complicar um pouco o assunto.

Digamos que você goste muito de couve, mas muito mesmo, e queira que ela apareça mais três vezes na lista de verduras. Teremos então três comandos a serem aplicados na mesma linha:

```
prompt$ sed '/couve/ p ; /couve/ p ; /couve/ p' verduras.txt
- alface
- cenoura
- couve
- couve
- couve
- couve
- nabo
- vagem
```

Ficar repetindo o endereço para cada comando é muito inconveniente. Utilizamos então as chaves `{}`, que nos permitem agrupar vários comandos em um mesmo bloco. Este bloco recebe um endereço, que será usado por todos os comandos contidos nele.

Nosso exemplo anterior fica assim:

```
prompt$ sed '/couve/{ p; p; p; }' verduras.txt
```

Preste atenção no seguinte detalhe: um bloco está **sempre** associado a um endereço, e a chave de fechamento "}" é como se fosse um comando do sed, devendo portanto ser separada de outros comandos com ponto-e-vírgula ";".

> Ao fechar o bloco, coloque sempre um **;** antes da **}**

Sei que você leu e entendeu, mas tenha absoluta certeza de que você vai se esquecer desse último ";" várias, diversas, milhares de vezes. &:)

Como endereçar um trecho entre duas linhas, incluindo-as

Tudo ia bem na nossa vida de endereçadores, até o dia em que precisamos endereçar mais de uma linha. Ou melhor, um trecho de texto entre duas linhas.

Por exemplo: como apagar da linha 5 à linha 10 de um arquivo? Numa primeira tentativa, afobada, a carreirinha

```
prompt$ sed '5d ; 6d ; 7d ; 8d ; 9d ; 10d' arquivo
```

... funciona. Mas e se quiséssemos apagar 20 ou 30 linhas? Esse método não é prático.

No sed, podemos especificar dois endereços, um de início e outro de fim, para representar os limites de um trecho entre duas linhas. Uma vírgula separa os dois endereços. Então, a tarefa anterior de apagar as linhas fica assim:

```
prompt$ sed '5,10 d' arquivo
```

Ou seja, da linha 5 até a linha 10, apague tudo. Lembre-se porém, que um endereço também pode ser o cifrão `$` ou um padrão. Então agora podemos misturar todos os tipos de endereço. Veja os exemplos:

```
1,/couve/        da primeira linha até a linha que contém 'couve'
10,$             da linha 10 até o fim do arquivo
/couve/,$        da linha que contém 'couve' até o fim do arquivo
/couve/,/vagem/  da linha que contém 'couve' até a linha que contém 'vagem'
1,$              da primeira linha até a última (ou seja, todas)
```

Como pudemos notar, o trecho casado **inclui** as linhas de início e fim do endereço. Se não era este seu objetivo, veja o tópico seguinte.

Como endereçar um trecho entre duas linhas, excluindo-as

No tópico anterior, vimos como endereçar um trecho contendo várias linhas e aprendemos que limitam o trecho também são incluídas no endereço.

Mas, às vezes, queremos endereçar apenas as linhas que estão dentro de um trecho, excluindo as linhas delimitadoras de início e fim. Nesse caso, precisamos de algo mais elaborado.

Vamos usar um outro arquivo de exemplo, com os números de um a seis por extenso:

```
prompt$ cat numeros.txt
um
dois
três
quatro
cinco
seis
```

Supondo que se queira destacar tudo entre as linhas que contiverem `um` e `cinco`, incluindo as próprias linhas delimitadoras, temos:

```
prompt$ sed '/um/,/cinco/ s/^/---/' numeros.txt
---um
---dois
---três
---quatro
---cinco
seis
```

Mas, se quisermos excluir as linhas delimitadoras, devemos tratá-las individualmente, dentro de um bloco:

```
prompt$ sed '/um/,/cinco/ { /um/ b ; /cinco/ b ; s/^/---/ ; }' numeros.txt
um
---dois
---três
---quatro
cinco
seis
```

Mais detalhadamente:

```
/um/,/cinco/ {  ;# entre a linha 'um' e a linha 'cinco' ...
  /um/ b        ;# se for a linha 'um', salte para o fim do script e
                ;# processe a próxima linha (ou seja, não faça nada nesta)
  /cinco/ b     ;# se for a linha 'cinco', salte para o fim do script e
                ;# processe a próxima linha (ou seja, não faça nada nesta)
  s/^/---/      ;# nos outros casos, coloque '---' no começo da linha
}
```

Então, o comando `b` sozinho funciona como um pulo, uma exceção que não deixa o processamento chegar até o `s/^/---/` nas linhas especificadas.

Como negar um endereço, não aplicando comandos nele

Assim como podemos definir um endereço para aplicação de um comando, também podemos determinar que um comando **não** seja aplicado nesse endereço, ou seja, que o comando se aplica a todas as linhas menos aquela. Veja como apagar todas as linhas do arquivo **menos** a linha 5:

```
prompt$ sed '5! d' arquivo
```

O modificador de endereço `!` é o responsável pela inversão, e essa lógica às avessas pode confundir. Uma leitura desse comando poderia ser: "Na linha 5, não a apague".

Para facilitar o entendimento correto de sua função, leia o comando de maneira inversa, usando a palavra "exceto" ou "menos". Assim: "Apague TODAS as linhas EXCETO a linha 5".

O mesmo vale para um bloco de comandos e para um endereço duplo:

```
/padrão/! { p; d; }

/padrão1/,/padrão2/! { p; d; }
```

Com blocos, a leitura fica: "Aplique os comandos do bloco em TODAS as linhas EXCETO as do endereço".

Assim sendo, fica fácil mostrar apenas um trecho de texto e apagar todo o resto:

```
prompt$ sed '/dois/,/cinco/! d' numeros.txt
dois
três
quatro
cinco
```

Em outro exemplo, vamos emular o comando `head` do UNIX, que imprime as 10 primeiras linhas de um arquivo. Uma das possibilidades seria esta:

```
sed '11,$ d'       apague da linha 11 até o fim do arquivo
```

Ou então, usando o modificador `!`, podemos inverter a lógica e fazer assim:

```
sed '1,10! d'      apague todas as linhas EXCETO da linha 1 até a 10
```

Pode parecer estranho (e é!) pensar invertido, mas você se acostuma &:)

Como endereçar a primeira ocorrência de um padrão

Como já vimos, o endereço pode servir para mais de uma linha. Após encontrar a primeira ocorrência do padrão e executar o comando, o sed continua lendo o arquivo. O comando então será novamente aplicado em todas as outras linhas em que o padrão for encontrado.

Mas, e para endereçar apenas a **primeira** ocorrência do padrão e não as restantes?

Aqui, precisamos de um pouco de criatividade. A idéia é especificar uma "área de atuação" para o comando, para que seu endereço somente seja válido nesta área.

Para nosso objetivo, a área deve começar no início do arquivo e ir até a primeira linha onde o padrão for encontrado. Ei, já sabemos como fazer isso, basta endereçar!

```
1, /padrão/ { comandos ; }
```

Usando a linha 1 como primeiro endereço, conseguimos "ancorar" o bloco de comandos no início do arquivo. Dessa maneira garantimos que apenas a primeira ocorrência do padrão está contida neste endereço.

Definida nossa área de atuação, agora ficou fácil! Por exemplo, para modificar somente a primeira linha que contenha a letra "o" no nosso arquivo de números:

```
prompt$ sed '1,/o/{ /o/ s/^/---/ ; }' numeros.txt
um
---dois
três
quatro
cinco
seis
```

Vamos ler este comando: "Dentro da área que vai da primeira linha até a linha que conter a letra 'o', faça: se for uma linha com 'o', adicione '---' no começo".

Há também uma tática mais simples, que funciona em casos específicos. Para tarefas como extração de dados, onde se busca apenas a primeira ocorrência e nada mais, o comando de interrupção de processamento pode ser utilizado.

A idéia é simples: colocar num bloco o comando desejado e a interrupção. Como a interrupção só pode ser executada uma vez, está feito nosso "condicional" para obter apenas o primeiro.

Exemplo: extrair do arquivo apenas a primeira linha com a letra "o":

```
prompt$ sed -n '/o/{ p; q; }' numeros.txt
dois
```

Com a opção `-n`, silenciamos a saída do sed. Na linha com "o", é executado um Print para mostrá-la e um Quit para sair.

A mesma tática é utilizada para extrair os cabeçalhos de um e-mail, que ficam sempre no início, separados do corpo da mensagem por uma linha em branco:

```
sed '/^$/q' email.txt
```

Traduzindo: "Ao encontrar a primeira linha em branco do arquivo, saia".

Como endereçar a primeira ocorrência de um trecho

Um endereço duplo também está sujeito às mesmas regras de um endereço normal, então um trecho de texto pode se repetir várias vezes e o seu comando (ou bloco) será executado tantas vezes quantas necessárias.

Assim sendo, o comando `/(/,/)/ d` apagará TODOS os trechos do texto que estiverem delimitados por parênteses, podendo o parêntese que fecha estar numa linha diferente do que abre.

Para limitar esse endereço ao primeiro trecho encontrado, usamos a mesma tática de ancoragem com a primeira linha usada no tópico anterior:

```
1,/)/ {
  /(/,/)/ d
}
```

Traduzindo: "Do começo do texto até a primeira linha que contiver ')', apague tudo que estiver entre os parênteses, inclusive eles próprios".

Como endereçar a última ocorrência de um padrão

Entramos agora num tópico complexo. Endereçar a primeira ocorrência de um padrão ou trecho é algo relativamente fácil. Mas a última não é tão simples:

```
prompt$ sed '/o/,${ /o/ d ; }' numeros.txt
um
três
seis
```

A tentativa óbvia falha porque o sed lê as linhas de cima para baixo, então o endereço especificado acaba significando: "Procure da primeira linha que tiver a letra 'o' até a última linha do texto".

Para fazermos um `s/primeira/última/` com este significado, temos algumas alternativas. A mais fácil, porém exclusiva de quem possui o comando `tac`, que imprime um arquivo da última para a primeira linha, pode ser:

```
prompt$ tac numeros.txt
seis
cinco
quatro
três
dois
um
```

```
prompt$ tac numeros.txt | sed '1,/o/{ /o/ d ; }' | tac
um
dois
três
quatro
seis
```

Então, invertendo a ordem das linhas, aplicamos o mesmo conceito de ancorar pela primeira linha do arquivo (que na verdade é a última), e então basta desinverter o arquivo aplicando `tac` novamente.

Quem não possui o `tac`, pode emulá-lo com o sed:

```
prompt$ sed '1! G ; h ; $! d' numeros.txt
seis
cinco
quatro
três
dois
um
```

E trocando o `tac` do comando anterior por este em sed, ficamos com:

```
sed '1!G;h;$!d' numeros.txt | sed '1,/o/{ /o/ d ; }' | sed '1!G;h;$!d'
```

Feio, muito feio. E esse método de inversão complica tudo, pois o script a ser aplicado ao trecho, caso seja necessário mexer com mais de uma linha, também precisa tratá-las de maneira invertida. Em algumas situações onde se tem uma seqüência certa de linhas, inverter a lógica pode ser impraticável.

Mas, com exceção dessa tática, não há muitas alternativas. Dependendo do que precisamos fazer com o padrão, podemos usar outras abordagens.

Por exemplo, se precisarmos simplesmente imprimir na tela a linha da última ocorrência, podemos usar o sed duas vezes:

```
prompt$ sed '/o/! d' numeros.txt | sed '$! d'
cinco
```

O primeiro obtém todas as linhas que contêm o padrão, e o segundo mostra apenas a última linha, apagando todas as outras. Um comando UNIX similar seria:

```
prompt$ grep 'o' numeros.txt | tail -1
```

Para fazer isso usando apenas um sed, podemos utilizar o RESERVA para ir armazenando (sobrescrevendo) todas as linhas que contém o padrão, e quando chegarmos na última linha do arquivo, mostramos o conteúdo do RESERVA:

```
prompt$ sed '/o/ h ; $! d ; $ g' numeros.txt
cinco
```

Para entender melhor:

```
/o/ h    ;# guarde (sobrescrevendo) no RESERVA as linhas que contêm 'o'
$! d     ;# apague todas as linhas, menos a última
$ g      ;# se for a última linha, pegue o conteúdo do RESERVA
```

Infelizmente, não há uma "receita de bolo" para endereçar de maneira genérica a última ocorrência. Cada caso é um caso, e merece uma solução personalizada.

Como endereçar a N-ésima ocorrência de um padrão

Fazendo magia negra. :)

Já vimos como é complicado endereçar a última ocorrência do padrão, e não há como fazer isso de maneira genérica. Agora imagine especificar uma posição arbitrária de repetição de um padrão.

Caso seja algo simples, como só mostrar na tela a terceira linha que contenha a letra "o", por exemplo, podemos usar a tática anterior de usar o sed duas vezes:

```
prompt$ sed '/o/! d' numeros.txt | sed '3! d'
cinco
```

Sim, é possível, dependendo do caso. Mas contando que o grau de dificuldade é enorme para se fazer isso num problema mais elaborado, além do código resultante ficar grande e difícil de se manter, o aconselhável é usar outra linguagem, como `awk`, que tem o conceito de linhas e registros, incluindo contadores, ou usar o sed em conjunto com um script shell.

Como nem um (shell) nem outro (awk) faz parte do nosso tópico, fim de papo.

Tem uma barra / no padrão que procuro, e agora?

Ao procurar linhas com datas, nomes de diretórios, ou outros padrões que possuem o caractere "/" podemos nos surpreender com as mais diversas mensagens de erro:

```
prompt$ sed '//tmp/lixo.bmp/ d' arquivo
sed: -e expressão #1, caractere 2: não há uma expressão regular anterior

prompt$ sed '/tmp/lixo.bmp/ d' arquivo
sed: -e expressão #1, caractere 7: há caracteres sobrando após o comando

prompt$ sed '/31/12/2000/ d' arquivo
sed: -e expressão #1, caractere 5: comando desconhecido: `1'
```

As mensagens de erro do sed são em geral curtas e esclarecedoras, mas no caso de problema com o delimitador, elas ficam especialmente obscuras. Anote em algum canto de seu cérebro:

> Se a mensagem de erro do sed é alienígena, confira os delimitadores.

O que aconteceu em todos os exemplos é que o sed confundiu a barra normal dos padrões com a barra delimitadora do endereço. O que temos que fazer aqui para que essas barras normais não sejam interpretadas como delimitadoras, é escapá-las.

Segundo o Aurélio (o outro), um dos significados de escapar é "passar despercebido". E é exatamente isso o que precisamos: fazer com que a barra não seja considerada especial pelo sed. Quem faz esta mágica é a barra invertida "\", que colocada antes de um caractere normal, o escapa, impedindo que o sed o considere um delimitador de endereço.

Em nosso caso, a barra "/" escapada fica: `\/`. Como dica geral, **sempre** escape as barras normais do padrão, evitando dores de cabeça.

Mas vamos voltar aos exemplos anteriores, agora com as barras devidamente escapadas:

```
prompt$ sed '/\/tmp\/lixo.bmp/ d' arquivo
prompt$ sed '/tmp\/lixo.bmp/ d' arquivo
prompt$ sed '/31\/12\/2000/ d' arquivo
```

Note que são escapadas apenas as barras internas, as barras delimitadoras do endereço continuam as mesmas.

Como usar outro delimitador fora a barra /

Agora cá entre nós, esse negócio de ficar escapando barras não é nem um pouco prático. Pior ainda se nosso padrão estiver dentro de uma variável, e não for tão visível o conflito de barras, como em `sed "/$PWD/ d"`, onde `$PWD` contém o diretório atual de trabalho.

Para esses e outros possíveis problemas, o sed nos dá a liberdade de escolher **qualquer** caractere da tabela ASCII como delimitador de endereço.

Então como regra geral para escolher qual símbolo usar, utiliza-se um delimitador que se tem **certeza** que não vai ser confundido com nenhum caractere do padrão a ser procurado.

O único porém para usar algo diferente da barra /, é que precisamos dizer ao sed: "Ei, o próximo caractere aqui vai ser o delimitador". Para isso, basta escapar o primeiro (somente o primeiro!) delimitador, assim:

```
sed '\,31/12/2000, d' arquivo
```

Então utilizamos a vírgula como delimitador de endereço, escapando a primeira para que o sed a veja como especial.

Delimitador: escolha **qualquer** ASCII, escape o primeiro

Fora a vírgula, outros delimitadores reserva que são clássicos de usar é a barra vertical `|`, o arroba `@` e a exclamação `!`. Agora, para evitar dor de cabeça e usar um delimitador que dificilmente coincidirá com qualquer padrão, use caracteres esquisitos como: `§`, `£` e `¢`.

Ou ainda, se você quiser fazer algo realmente esquisito, use caracteres brancos como o espaço ou o TAB, ou letras normais do alfabeto, números (cuidado!)

```
sed '\ 31/12/2000 d' arquivo

sed '\i31/12/2000i d' arquivo

sed '\831/12/20008 d' arquivo
```

Ou pior ainda, até a quebra de linha pode ser usada!

```
prompt$ sed '\
> 31/12/2000
> d' arquivo
```

Quem disse que o sed não é divertido? &:)

Uma regra boa para a escolha do delimitador, além dele não coincidir com algum caractere do padrão, é que ele seja **visualmente** oposto ao padrão. Isso quer dizer que se o seu padrão tiver uma predominância de caracteres altos, como uma palavra em MAIÚSCULAS, é aconselhável usar caracteres baixos como delimitadores, como a vírgula, dois pontos e o sublinhado. E se o padrão for predominantemente baixo, caracteres como / ! | % e @ são mais indicados. Veja:

visual	**confuso**
`s_$PWD_/TMP_`	`s!$PWD!/TMP!`
`s|...=|=|`	`s:...=:=:`

O que acontece com endereços inválidos

Um endereço inválido pode ser uma palavra que não é encontrada em nenhum lugar no texto, ou uma linha que não existe, por exemplo a linha 20 num arquivo que só possui 15 linhas.

E agora a dúvida que assola a humanidade: "E se eu colocar um endereço inválido?" a resposta é: depende.

Se for um endereço simples de uma linha, o comando simplesmente não vai ser executado, pois nenhuma linha satisfez o padrão.

Se for um endereço de um trecho entre duas linhas, o resultado é diferente caso o endereço inválido seja o primeiro ou o segundo. Vamos acompanhar um exemplo utilizando a palavra `prego` como padrão no nosso arquivo de verduras:

```
prompt$ sed '/prego/,$ d' verduras.txt
- alface
- cenoura
- couve
- nabo
- vagem
prompt$ sed '1,/prego/ d' verduras.txt
prompt$
```

No primeiro exemplo, era para apagar da linha que tivesse `prego` até o final. Como nenhuma linha tinha `prego`, o primeiro endereço falhou e nada foi apagado.

Já no segundo exemplo era para apagar da primeira linha até a linha que tivesse `prego`. A primeira linha foi encontrada, então o comando `d` vai ser aplicado até achar o segundo endereço. Mas como não tem `prego` no arquivo, o comando vai sendo aplicado até bater na última linha. É como dizer: "Ninguém me mandou parar, então continuei".

```
1º endereço inválido: endereço inválido, nada acontece
2º endereço inválido: endereço incompleto, vai até o final
```

E se eu colocar um endereço vazio como //?

Um endereço vazio referencia o último endereço pesquisado.

Essa é uma funcionalidade **muito** útil e que pode poupar muita redundância no endereço.

Lembra do exemplo de imprimir mais três vezes a linha da couve?

```
prompt$ sed '/couve/ p ; /couve/ p ; /couve/ p' verduras.txt
```

Esse comando poderia ser reescrito assim:

```
prompt$ sed '/couve/ p ; // p ; // p ' verduras.txt
```

E aquele outro de ancoragem para imprimir apenas a primeira linha que contivesse a letra o?

```
prompt$ sed '1,/o/{ /o/ d ; }' numeros.txt
```

Este pode ficar assim:

```
prompt$ sed '1,/o/{ // d ; }' numeros.txt
```

A diferença parece pequena porque os exemplos usam endereços bem simples, mas quando eles são monstros gigantes que cospem expressões regulares, o endereço vazio ajuda muito.

E tem ainda o clássico `//s///` que é um comando válido e bem utilizado. Você se arrisca a adivinhar o que ele faz?

E se eu não colocar nenhum endereço?

O comando serve para **todas** as linhas. Ponto.

```
prompt$ sed 'd' verduras.txt
prompt$
```

Detalhes sórdidos sobre endereços

Apesar de os tópicos anteriores esmiuçarem vários aspectos do endereço, ainda restam alguns detalhes que devem ser assimilados.

- Os únicos comandos que **não** recebem endereço são o `:` e o `}`, ambos por serem comando relativos a posições do próprio script e não do texto que está sendo processado.

- Os comandos que **não** recebem **dois** endereços são `=`, `a`, `i`, `q` e `r`, pois só podem ser aplicados a apenas uma linha por vez. Mas se diretamente não conseguimos endereçá-los a um trecho como em `1,5=`, usando blocos não há problema: `1,5{ =; }`

- Os comandos que **devem** receber endereço são `!` e `{`. O `!` por ser um inversor de endereços, aplicando o comando **exceto** no endereço indicado e o `{` que define um bloco de comandos a ser aplicado no endereço especificado. Se não tiver endereço, o bloco será aplicado para todas as linhas tornando desnecessário o agrupamento.

- As linhas de início e fim de um trecho não podem ser a mesma linha, então mesmo que seu endereço tenha padrões idênticos como `/cebola/, /cebola/`, serão necessárias duas linhas diferentes que contenham a palavra `cebola` para que este endereço seja encontrado. Então um trecho sempre tem no mínimo, duas linhas.

- O `\n` como identificador de quebra de linha no endereço só é válido após a aplicação do comando `N`, que gruda linhas no PADRÃO separando-as pelo `\n`. Do contrário, não haverá mais de uma linha no PADRÃO para se processar.

Arquivo

Já vimos que o sed não é um editor de arquivos, mas de fluxos de texto. Como tal, a sua função é aplicar os comandos de edição no texto e mostrar o resultado na tela.

Mas mesmo tendo seu funcionamento independente do conceito de arquivos, o sed precisa lidar com eles, pois nós humanos gostamos de arquivos!

Temos nossos dados guardados em arquivos, então o sed precisa ter a capacidade de ler e gravar linhas em arquivos. Podemos ainda querer guardar todo o resultado da edição feita pelo sed em um arquivo. Ou ainda, podemos querer colocar nossos comandos sed num arquivo, pois eles estão ficando muito complicados. Feito isso, podemos até torná-lo um arquivo executável, que chama o sed automaticamente para interpretá-lo!

Ufa! Para um programa que edita somente fluxos de texto, até que temos vários detalhes para ver no assunto "Arquivo", não? &:)

Como gravar o resultado num arquivo

Apesar do sed ter sido concebido como um filtro, que repassa o fluxo de texto para a tela do computador, nós usuários também temos necessidades de armazenamento desse fluxo.

Seja para consulta posterior, para edição de arquivos (e não fluxos) ou para registros e extração de dados, é importante guardamos o texto processado pelo sed em um arquivo.

Mas sendo o sed um filtro de fluxos, o que ele entende de arquivos? Quase nada. Por isso a solução desse problema está um pouco mais embaixo, no **shell**, que é o ambiente onde o sed é executado.

A tela é chamada de "saída padrão" do sed, o caminho natural que o fluxo segue após ser processado. Mas como essa saída passa também pelo shell antes de ir para a tela, ele tem meios de "desviá-la" e mandá-la para um arquivo. Esse desvio é chamado de "redirecionamento", e o caractere utilizado para representá-lo é o maior-que ">".

> Informatiquês: Precisamos redirecionar a saída padrão!

Então aplicamos os comandos sed desejados no texto e redirecionamos o resultado dessa edição para um arquivo:

```
prompt$ sed 'comandos' texto.txt > texto-alterado.txt
```

Este redirecionamento é chamado **destrutivo**, pois caso já exista o arquivo texto-alterado.txt, ele será "truncado". Truncar em informatiquês significa cortar, apagar, excluir, remover, limpar, zerar, ou seja, o conteúdo anterior do arquivo será perdido. Caso o arquivo ainda não exista, o shell o criará.

A outra opção é usar o redirecionamento **incremental**, representado por dois sinais de maior-que ">>". Ao contrário do destrutivo, se o arquivo já existir, seu conteúdo original será preservado e o texto novo será anexado após a última linha. Caso o arquivo ainda não exista, o shell o criará.

```
prompt$ sed 'comandos' texto.txt >> texto-alterado.txt
prompt$ sed 'outros-comandos' texto.txt >> texto-alterado.txt
```

Memorizar é fácil!

Usando...	...acontece
>	o arquivo é sempre zerado
>>	o arquivo vai crescendo, crescendo...

A maioria dos ambientes shell conhecidos utiliza a notação do > e >> para redirecionamentos, aqui vão alguns deles:

- no UNIX/Linux: sh, ash, bash, ksh
- no Macintosh: MPW Shell
- no Windows: MS-DOS

Como gravar o resultado no próprio arquivo original

Olhe ali para aquela câmera, você acaba de cair na pegadinha de gravar no mesmo arquivo!

- Quem nunca apagou um arquivo importante para descobrir que o UNIX/Linux não tem undelete nem "Lixeira"?
- Quem nunca perdeu um HD inteiro para descobrir a importância do becape?
- Quem nunca apertou o botão "Ok" na mensagem "Deseja sair sem salvar as alterações?" e teve que redigitar o documento?
- Quem nunca deu um `rm -rf / tmp/lixo` ou um `rm -rf * .txt` ou um `rm -rf .*` ?

Pois é amigo, brincadeiras à parte, fazer cacas irreversíveis em informática é tão fácil que uma simples apertada na tecla Enter na hora errada pode destruir um dia de trabalho (ou vários...).

Essa introdução descontraída serve para aliviar o peso da culpa daqueles que já descobriram pelo jeito difícil que gravar o resultado no mesmo arquivo pode ser cruel &:)

Problema inicial

Como já visto, utiliza-se o redirecionamento do shell para gravar o resultado do sed num outro arquivo. Mas é muito comum ao se editar um determinado arquivo, querer gravar estas alterações no próprio arquivo original. Faz sentido não faz? Então a tentativa óbvia e intuitiva é o clássico tiro no pé:

```
prompt$ sed 'comandos' texto.txt > texto.txt
```

Além de não dar certo, todo o conteúdo original do arquivo **será perdido**. Que dureza hein? Mas se é errando que se aprende, caso você nunca tenha feito essa caquinha (ainda), agora é sua chance de aprender sem precisar sofrer.

O que ocorre é que ao fazer o redirecionamento destrutivo >, a primeira coisa que o shell faz é truncar o arquivo referenciado, antes mesmo de começar a executar o sed. Então o conteúdo do arquivo será apagado **antes** do sed poder lê-lo.

Quando finalmente o sed for chamado pelo shell, receberá um arquivo vazio, e aí não dá para fazer mágica: aplicados quaisquer comandos num arquivo vazio, o resultado será sempre um arquivo vazio. E fim de papo.

Solução genérica

Para solucionar este dilema, voltamos a tática já conhecida de redirecionar o resultado para um outro arquivo, e depois **mover** o arquivo novo sobre o original:

```
prompt$ sed 'comandos' texto.txt > texto-alterado.txt
prompt$ mv texto-alterado.txt texto.txt
```

Para a grande maioria dos casos, isso é suficiente. Porém caso as características do arquivo original devam ser mantidas, temos um problema.

Como acabamos de criar um arquivo novo para guardar o texto alterado, ele terá os atributos padrão do sistema. Ao movê-lo sobre o arquivo original, estes atributos padrão serão herdados.

Atributos especiais que o arquivo original pudesse ter como: grupo diferente do padrão do usuário, permissões específicas (rwx) ou referências para outros arquivos (links, simbólicos ou não) **serão perdidos**.

Solução segura

Para uma solução à prova de falhas, usa-se uma abordagem mais conservadora e segura: copiar o arquivo e redirecionar a saída para o original.

A primeira ação a se fazer é **copiar** o arquivo original para um arquivo temporário qualquer. Feito isso, então se aplica os comandos sed neste temporário. Agora vem o pulo do gato: redireciona-se a saída do sed para o arquivo original:

```
prompt$ cp texto.txt texto-tmp.txt
prompt$ sed 'comandos' texto-tmp.txt > texto.txt
```

Dessa maneira não estamos criando um arquivo texto.txt novo, mas apenas trocando o seu conteúdo, permanecendo inalteradas as suas características no sistema. Para finalizar a operação, basta apagar o arquivo temporário.

```
prompt$ rm texto-tmp.txt
```

É importante ter em mente a importância desta abordagem segura e o porquê de utilizá-la. Não apenas para o sed, mas também para outros comandos do sistema que atuam como filtros, mandando o resultado para a tela.

Solução moderna

Para evitar todas essas preocupações e não ter que criar um arquivo temporário, algumas versões do sed (FreeBSD, ssed, gsed-4) possuem a opção "**-i**", que aplica os comandos "in-place", ou seja, no mesmo arquivo.

Dessa forma, não se depende mais do shell para contornar a situação:

```
prompt$ sed -i 'comandos' texto.txt
```

Como guardar os comandos sed num arquivo (script)

À medida que vamos evoluindo no aprendizado, os comandos sed vão ficando cada vez mais extensos e complicados. Ficar sempre redigitando tudo na linha de comando torna-se um incômodo. Seria interessante poder guardar os comandos num arquivo!

Nesse momento em que sentimos a necessidade de algo mais prático, presenciamos nossa própria evolução, a passagem da infância para a adolescência no aprendizado do sed. Ao invés de fazer um "comando sed", agora vamos fazer um "script sed", com bastante comandos e estruturado. Na fase adulta de aprendizado, podemos ainda evoluir para um "programa sed", que ao invés de editar textos, os domina.

O simples fato de colocar os comandos sed num arquivo ao invés de digitá-los na linha de comando, muda alguns aspectos da brincadeira divertida de manipular texto. Vamos aos detalhes sórdidos!

A primeira vantagem é que não precisamos mais nos preocupar com o shell, tendo que proteger os comandos entre aspas para não serem confundidos e interpretados. Dentro do arquivo podemos colocar quaisquer caracteres, despreocupadamente.

Mas é no mesmo ponto que temos a primeira desvantagem: perdemos o contato com o shell, nosso aliado poderoso. Com suas variáveis, operações e comandos, o shell complementa onde o sed é limitado. Quando colocamos os comandos sed num arquivo, eles tornam-se estáticos, e a interatividade direta com o shell é perdida.

Uma outra vantagem da utilização do arquivo, é que podemos separar melhor os comandos, colocando um em cada linha. Podemos fazer inclusive o alinhamento estruturado ("indentation") para que os blocos de comandos fiquem mais visuais.

E mais, agora podemos colocar também comentários! Desprezado por muitos, eles são uma das partes mais importantes de um programa. Só quem já precisou dar manutenção num código complexo e sem comentários (ou mal comentado) sabe a falta que eles fazem...

E por favor, pelo seu próprio bem e dos que um dia precisarem dar manutenção num script sed de sua autoria, encha-o de comentários! O sed é especialmente abstrato e desafiador, com seus comandos de apenas uma letra, sem variáveis e muitas expressões regulares. A falta de comentários pode deixar um script inutilizável se ninguém souber como atualizá-lo. O caractere que precede os comentários num script sed é aquele que cada um chama de um jeito: #. Gradinha, cerquilha, sustenido, jogo da velha, hash, ...

Script sed: shell--, alinhamento++, comentários++, legibilidade++

Vamos relembrar o exemplo de endereçar entre duas linhas, excluindo-as:

```
prompt$ sed '/um/,/cinco/ { /um/ b ; /cinco/ b ; s/^/--- / ; }' numeros.txt
```

É um endereço com um bloco de comandos dentro das chaves {}. No bloco estão três comandos, separados por ponto-e-vírgula. Há uma certa dificuldade em identificar e entender as partes desse comando, por estar tudo misturado numa única linha. Vamos colocá-lo num arquivo e ver se melhora?

```
### meu primeiro script sed
# ei, isso é um comentário!

# vamos delimitar um bloco de linhas
# endereço: entre a linha que contém 'um' e a linha que contém 'cinco'
/um/ , /cinco/ {

    ### nesse ponto do script, só chegarão as linhas que estiverem entre
    ### 'um' e 'cinco', incluindo elas próprias.

    # se for a linha que contém 'um', vá até o final do script
    /um/ b

    # se for a linha que contém 'cinco', vá até o final do script
    /cinco/ b

    ### nesse ponto do script, as linhas 'um' e 'cinco' já não chegam mais.
    ### então somente para as outras será aplicado o próximo comando.

    # coloque um '---' no começo da linha
```

```
    s/^/---/

# fim do bloco
}

### fim do script
```

Tudo bem, os comentários estão exageradamente explicativos, mas a idéia é mostrar o quão didático pode ficar seu script sed. Com certeza ficou 78% mais fácil de ler! (Não são irritantes essas estatísticas subjetivas e absurdas que ouvimos diariamente?)

É notável a diferença do entendimento entre este script e a linha de comando anterior onde os comandos estavam todos grudados. Moral da história: fica a critério do programador fazer um código pequeno e indecifrável, ou extenso e legível. Ambas as formas têm suas audiências.

Como os espaços em branco no começo de cada linha são irrelevantes para o sed, temos a liberdade de estruturar as linhas como preferirmos, podendo utilizar espaços e TABs. Note também que como separamos os comandos um por linha, não precisamos mais colocar o ponto-e-vírgula entre eles.

Cada um tem a liberdade de nomear os scripts sed como bem entender, não há um padrão ou norma. Mas para facilitar a identificação de seu conteúdo, é aconselhável utilizar a extensão .sed. Este script de exemplo poderia se chamar seu-madruga-no-brasil.avi, mas bloco.sed é muito mais descritivo e apropriado, não concorda? &:)

Como utilizar um script sed

Agora que já sabemos como colocar os comandos sed num arquivo, tornando-os parte de um script sed, resta saber como fazer o sed utilizar este script.

Basta usar a opção **-f** ou **--file**, seguida do nome do script, que serve para para dizer ao sed: "Use os comandos que estão dentro deste arquivo". Veja:

```
prompt$ sed -f arquivo.sed texto.txt
```

Essa opção pode ser usada mais de uma vez na linha de comando, então vários scripts podem ser usados no mesmo comando:

```
prompt$ sed -f arq1.sed -f arq2.sed -f arq3.sed texto.txt
```

O sed os interpreta seqüencialmente, ou seja, a ordem de colocação dos vários "-f" tem importância. Se o exemplo anterior fosse trocado para:

```
prompt$ sed -f arq3.sed -f arq2.sed -f arq1.sed texto.txt
```

o resultado não será o mesmo.

Lembrando da escolinha: A ordem dos fatores ALTERA o produto

Não acredita? Tudo bem, vamos à demonstração então:

```
prompt$ cat abc.sed
s/abc/XXX/
prompt$ cat xxx.sed
s/XXX/xyz/
prompt$ echo abc | sed -f abc.sed
XXX
prompt$ echo abc | sed -f abc.sed -f xxx.sed
xyz
prompt$ echo abc | sed -f xxx.sed -f abc.sed
XXX
```

O script abc.sed troca "abc" por "XXX" e o xxx.sed pega esse "XXX" e troca por "xyz". Como o xxx.sed depende da alteração feita pelo abc.sed, ele precisa ser executado depois, senão não haverá "XXX" para ele trocar.

Além de vários "-f", também é possível colocar comandos normais junto na mesma linha de comando utilizando a opção **-e**, e fazer uma salada de frutas. As regras continuam as mesmas: a "ordem de chegada" é importante. Acompanhe:

```
prompt$ cat a.sed
s/a/X/g
```

```
prompt$ echo abc | sed -f a.sed
Xbc
prompt$ echo abc | sed -e 's/b/a/' -f a.sed
XXc
prompt$ echo abc | sed -e 's/b/a/' -f a.sed -e 's/c/X/'
XXX
prompt$ cat x.sed
s/X/./g
prompt$ echo abc | sed -e 's/c/a/' -f a.sed -e 's/b/X/' -f x.sed
...
```

Para não se perder na ordem, basta lembrar que o sed está recebendo vários pedacinhos de comandos, mas hora de executar, é uma coisa só. É como se ele fosse recebendo vários bilhetinhos de papel com tarefas à fazer. Cada bilhete novo recebido é colado com durex no final do anterior, até que todos tenham sido entregues. Ao começar a execução dos comandos, o sed terá uma grande folha remendada e disforme de tarefas a fazer.

Como tornar executável um script sed

No UNIX/Linux, scripts sed podem se tornar arquivos executáveis, bastando digitar seu nome para ser interpretado, sem precisar chamar com o `sed -f`. Essa mágica é feita em dois passos:

1. Informar ao sistema que o arquivo é executável
2. Informar ao sistema como executá-lo

O primeiro passo é um procedimento comum, aplicado a todos os arquivos do sistema que devam ser executáveis. É necessário alterar as características do arquivo, incluindo nele as permissões de execução. O comando chmod se encarrega disso e a opção a ser utilizada é a **+x**:

```
prompt$ chmod +x meu-script.sed
```

Quem utiliza o comando ls com cores, vai notar que script agora ficou verdinho. É sinal que ele já é um arquivo executável. Para garantir, ou no caso de um ls sem cores, use o comando `ls -l` e veja se as permissões do arquivo ganharam alguns "x", mudando de "`-rw-rw-r--`" para "`-rwxrwxr-x`". Veja:

```
prompt$ ls -l meu-script.sed
-rw-rw-r--    1 mussum   aa     284 Feb  2 17:10 meu-script.sed
prompt$ chmod +x meu-script.sed
```

```
prompt$ ls -l meu-script.sed
-rwxrwxr-x    1 mussum   aa     284 Feb  2 17:10 meu-script.sed
```

Ok, temos um arquivo executável. Mas ainda não informamos ao sistema que é o sed quem deve interpretar nosso script. Como fazer isso?

O shell, ao executar um arquivo, dá uma olhadinha na primeira linha dele e vê se os dois primeiros caracteres são gradinha-exclamação **#!**. Se for, o restante dessa linha é considerado um comando de chamada, que diz ao shell como executar o arquivo. Funciona como um manual de instruções para o shell ler &:)

Quem já tem alguma familiaridade com o shell e já fez ou viu algum script shell, certamente conhece a primeira linha mágica que os torna especiais:

```
#!/bin/sh
```

Assim o shell sabe que precisa chamar o programa `/bin/sh` (que é o próprio shell!) para executar este arquivo. E isso funciona com qualquer outro comando do sistema. Então para um script sed, basta colocar como primeira linha o comando que normalmente usaríamos para interpretá-lo:

```
#!/bin/sed -f
```

Note que é preciso especificar o caminho completo do executável do sed, pois o PATH não é levado em conta nesse momento. Pronto! O script agora é um arquivo executável legítimo. Vamos testá-lo?

```
prompt$ meu-script.sed texto.txt
bash: meu-script.sed: command not found
```

Opa! Como assim "comando não encontrado"? Ahhhh, é que o diretório corrente não está incluído no PATH do sistema, então é preciso colocar um ./ na frente do nome do script para o shell encontrá-lo:

```
prompt$ ./meu-script.sed texto.txt
```

Caso seu script fique muito bom (vai ficar, claro!) e você o utilize com bastante freqüência, copie-o para o diretório dos programas de verdade para não precisar mais do ./ para chamá-lo:

```
prompt$ cp meu-script.sed /usr/bin
```

Como monitorar um arquivo com a ajuda do sed

No sed da GNU, a partir da versão **3.02.80**, foi adicionada a opção **-u** (de "unbuffered") que instrui o sed a fazer um uso minimalista dos registradores ("buffers"), tentando mostrar a saída o mais rápido possível.

No comportamento normal, o sed fica esperando até acabar todo o fluxo de texto (EOF), para só então começar a mandar para a saída o resultado do processamento. Com esta opção nova, este tempo de espera não existe, e assim que ele acaba de processar uma linha, já a manda para a saída instantaneamente.

A grande vantagem desse comportamento, não é a velocidade de resposta, mas sim o fato de não esperar o final do texto. Isso torna possível editar um fluxo de texto interminável. Peraí, como assim interminável?

Sabe aqueles arquivos de registros ("log files"), que vão crescendo, crescendo indefinidamente, com mensagens de erro, data, informações e avisos? Alguns exemplos no Linux são `/var/log/messages` e o `/var/log/maillog`. Enquanto o serviço que gera estas mensagens estiver ativo, ele continua escrevendo mais e mais texto em seu arquivo de log.

É muito comum se precisar acompanhar em tempo real a aparição das mensagens de log, para diagnosticar problemas ou acompanhar eventos. Um utilitário UNIX que faz isso é o tail, quando usado em conjunto com a opção **-f**. Ele monitora um arquivo, mostrando na tela cada mensagem nova que aparece no fim do arquivo de registros.

Como geralmente um arquivo de registros mostra mensagens variadas, às vezes até com excesso de informações, torna-se necessário o filtrarmos um pouco, e obter apenas as mensagens relevantes à nossa necessidade no momento. Opa, falou em filtro? Sed nele!

Um exemplo "da vida real" seria mostrar apenas as mensagens do sistema relativas às conexões ssh:

```
prompt$ tail -f /var/log/messages | sed -nu '/sshd/p'
```

E claro, ao invés de simplesmente mostrar a linha, o sed também já podia editá-la, tirar partes que não interessam, colocar códigos de cores para destacar informações, etcetcetc. Mas não vou tirar do leitor o prazer de fazer sozinho essas fresc^W melhorias.

Ah, e cuidado com esse -nu perto de crianças! &:)

Como ler o conteúdo de um arquivo

Já sentiu a necessidade de no meio de um script sed, ler o conteúdo de um arquivo qualquer e mandá-lo para a saída? É exatamente esta a função do comando "**r**" (de "Read"). Mas atenção! Como o sed gosta de fluxos de texto e não de arquivos, esse comando é um pouco, como dizer... equivocado.

A impressão que se tem é que o sed estava pronto para sair do forno e de última hora chegou o "Sr. Gerente Foobar" e mandou o programador colocar suporte a leitura de arquivos, **em 5 minutos**. O resultado é que um comando que deveria ser simples (ler arquivos), é cheio de pegadinhas e falhas conceituais.

Então quando usar este comando, tenha em mente que o conteúdo do arquivo é enviado à saída...

- ... diretamente, **não podendo ser editado**
- ... somente **no final** do script, mesmo havendo comandos após
- ... somente **após** processada e impressa a linha atual do buffer
- ... **sempre**, ignorando a opção -n (ou o `#n` na primeira linha)

Por esse resumo já deu para perceber os furos de implementação não é? Mas antes de ver os detalhes, vamos aprender a usá-lo, começando com um exemplo bem comum: incluir cabeçalho e rodapé num arquivo qualquer.

Estamos fazendo algumas páginas de internet, e percebemos que os textos do início (título, menu de navegação) e o do final (contato) são sempre os mesmos para todas as páginas. Como somos espertos, guardamos estes textos padrão em arquivos separados (cabecalho.html e rodape.html), e usamos o sed para inclui-los automaticamente nas nossas páginas HTML! Um exemplo genérico seria:

```
prompt$ sed -e '1 r cabecalho.html' -e '$ r rodape.html' meio.html > pagina.html
```

Incluindo o cabeçalho no início (após a linha 1) e o rodapé no final (após a última linha). O comando UNIX similar seria:

```
prompt$ cat cabecalho.html meio.html rodape.html > pagina.html
```

O primeiro detalhe que percebemos no exemplo é que precisamos separar os comandos usando a opção **-e**. Mas por que isso? Bem, lembra do detalhe dos "5 minutos"? Pois é. Na definição de quais caracteres seriam válidos para um nome de arquivo, o **ponto-e-vírgula** foi incluído. Pense um pouco sobre o efeito colateral disso...

Tudo bem, o ponto-e-vírgula é um caractere que pode aparecer no nome de um arquivo. Mas ele também é o caractere que delimita onde termina um comando sed e começa outro! E você acha que há uma maneira especial de colocar o ponto-e-vírgula no nome do arquivo para que o sed não confunda com o separador de comandos? Ha-ha. Não.

Com essa implementação descuidada, o resultado é no mínimo incômodo. Nos quatro exemplos seguintes, o trecho em destaque é o "nome do arquivo" que o comando "r" de cada um tentará ler:

sed '1 r **cabecalho.html ; $ r rodape.html**' meio.html > pagina.html

sed 'r **/tmp/arquivo.txt ; s/abc/XXX/ ; 5d**' texto.txt

sed 'r **c:\temp\arquivo.txt ; 10q**' texto.txt

sed 'r **arquivo.txt ; 1,/End:/!d**' texto.txt

Entendeu a dificuldade? O ponto-e-vírgula que deveria separar os comandos, é entendido como parte do nome do arquivo, então não há caractere que sirva para terminar esse nome. Com isso, o sed considera o "nome do arquivo" tudo o que vier depois do "r", até o fim da linha.

Para contornar essa dificuldade, o comando "r" (assim como o seu irmão, o comando "w") deve estar isolado, não havendo nada após ele senão o nome do arquivo desejado. Num script sed, basta quebrar a linha após o nome do arquivo. Na linha de comando deve-se usar o **-e** para separar as partes do comando sed.

Entendido isso, vamos ao próximo "detalhe": o conteúdo do arquivo é mandado **diretamente** para a tela. O ideal seria o sed ler o arquivo e colocar o seu conteúdo no espaço padrão, para podermos manipulá-lo. Mas ao invés disso o conteúdo é simplesmente "jogado" para a saída, sem controle, sem edição.

> Obs.: O ssed e o gsed-4 implementaram essa funcionalidade de ler o arquivo para o espaço padrão com o comando "**R**" (maiúsculo)

Mas não acaba aí. Além de jogar o conteúdo na saída sem controle, o sed faz isso de maneira não seqüencial também! Olha o absurdo:

```
prompt$ cat abc.txt
  #### Eu sou o conteúdo do arquivo
  #### abc.txt, de duas linhas.

prompt$ echo | sed -e 's/.*/ANTES/p ; r abc.txt' -e 's/.*/DEPOIS/'
ANTES
DEPOIS
  #### Eu sou o conteúdo do arquivo
```

```
#### abc.txt, de duas linhas.
```

O comando que foi usado possui três partes:

1. Mostra a palavra "ANTES" na tela
2. Mostra o conteúdo do arquivo abc.txt
3. Mostra a palavra "DEPOIS" na tela

Esta **deveria** ser a ordem de execução dos comandos, mas como observado, o sed só mostrou o conteúdo do arquivo depois de completar o terceiro e último comando! Mais uma pegadinha do "efeito 5 minutos"...

Tem mais, tem mais! Caso tenha um comando "**N**" no script, que forçaria a leitura da próxima linha, o conteúdo do arquivo é colocado no lugar certo:

```
prompt$ echo | sed -e 's/.*/ANTES/p ; r abc.txt' -e 's/.*/DEPOIS/ ; N'
ANTES
  #### Eu sou o conteúdo do arquivo
  #### abc.txt, de duas linhas.
DEPOIS
```

Não tente entender, bug é bug. E como esse é um bug que vem desde os tempos mais remotos do sed, dificilmente será corrigido pois muitos scripts foram feitos baseados nessas regras, e mudá-las significa quebrar esses scripts.

Para finalizar a sessão de detalhes do comando "r", ele passa por cima da opção **-n**. Essa opção deixa a saída do sed silenciosa, apenas mostrando texto quando especificado pelos comandos "**P p l**". Na prática vemos que o comando "r" é imune a essa opção e mostra o conteúdo do arquivo sempre.

```
prompt$ echo | sed -n 'r abc.txt'
  #### Eu sou o conteúdo do arquivo
  #### abc.txt, de duas linhas.
```

Como gravar um trecho de texto num arquivo

Assim como temos o comando "r" para a leitura de arquivos, também temos o comando complementar "**w**", que grava num arquivo o conteúdo atual do espaço padrão.

Lembra do "Sr. Gerente Foobar" do tópico anterior? Pois é, além do comando de leitura, ele também mandou o pobre programador fazer um comando de gravação de texto em arquivos. Mas como dessa vez ele foi mais generoso e deu o prazo de **10 minutos**, este comando ficou um pouco melhor que o anterior. Detalhes:

- A linha é gravada **imediatamente**, não espera o fim do script como o "r"
- O arquivo é **sempre truncado**, e seu conteúdo anterior perdido
- O arquivo é **sempre truncado** mesmo que o comando "w" não seja executado
- O nome do arquivo aceita ponto-e-vírgula, então o comando deve estar **separado** dos demais (-e)

Mas antes dos detalhes, vamos ao uso. Por exemplo, se você quer gravar num arquivo o resultado de uma busca por linhas que contêm a palavra "estorvo". O comando UNIX seria:

```
prompt$ grep 'estorvo' texto.txt > estorvos.txt
```

E o similar em sed:

```
prompt$ sed -n '/estorvo/w estorvos.txt' texto.txt
```

Traduzindo: Para cada linha do arquivo texto.txt que encontrar a palavra estorvo, grave-a no arquivo estorvos.txt.

Como irmão do comando "r", o "w" sofre do mesmo problema da especificação do nome do arquivo e o ponto-e-vírgula. Isso quer dizer que se deve usar a opção **-e** para separar os comandos, ou quebrar a linha quando utilizá-lo. Como no exemplo anterior o "w" era o único comando, a separação não foi necessária.

Uma outra utilidade de se gravar dados num arquivo é a de servir como um **espaço auxiliar** caso você já esteja usando o RESERVA e ainda precisa de outro cantinho para guardar dados durante a execução do script. Um exemplo clássico, é se processar um documento estruturado, com títulos e seções, e no final imprimir uma lista de tópicos (TOC), apenas com os títulos:

```
prompt$ sed -e '/<H1>/ w /tmp/toc' -e '$ r /tmp/toc' documento.html
```

Neste exemplo simples, gravamos num arquivo temporário todos os títulos de um documento HTML, que são identificados pela marca "**<H1>**". Somente quando estivermos na última linha do arquivo ($), lemos o conteúdo do arquivo temporário, que contém todos os títulos agrupados.

Apenas como curiosidade, o comando similar que utiliza o RESERVA e a técnica já demonstrada de "como fazer um laço utilizando o reserva":

```
prompt$ sed '/<H1>/ H ; $! d ; g' documento.html
```

Vejamos agora uma característica, ou limitação dependendo do caso. Se o arquivo a ser gravado não existir, o sed o criará. Porém caso ele já exista, o sed primeiro o **truncará** para depois utilizá-lo.

Já vimos que truncar significa perder todo o conteúdo anterior. Então este arquivo não pode ser usado como arquivo de registros ("logfile") ou como algum tipo de histórico, pois ele é "varrido" a cada execução do script.

E uma característica da característica, é que além de truncar o arquivo, não é que o danado faz isso **imediatamente**, assim que é chamado?! E ele faz isso mesmo que o comando "w" nem venha a ser executado. Ou seja, se tiver algum "w" no script, primeiro o sed vai lá e trunca o arquivo, só depois o script é executado. Aí já não importa mais se o trecho do comando "w" vai ser necessário ou não, pois o conteúdo do arquivo já foi apagado mesmo...

Usando o "w", o arquivo é SEMPRE truncado

Esse comportamento é facilmente verificável com este comando:

```
prompt$ sed '1 q ; 2 w /tmp/arquivo' arquivo.txt
```

Como o primeiro comando diz que na linha 1 o script deve ser abortado (comando "q"), o resto dele (comando "w") nunca chegará a ser executado. Mesmo assim, o arquivo /tmp/arquivo será zerado.

Esse comportamento impede que se utilize o mesmo arquivo para um comando `r` seguido de um comando `w`, ou seja, mostrar seu conteúdo atual antes de gravá-lo. Muito cuidado!

Quebra de linha

Lidar com quebras de linhas é com certeza umas das tarefas mais chatas de se fazer com o sed, chatas no sentido de se tomar cuidado e saber de algumas "pegadinhas".

Mmmmmm, será que eu devo contar os segredos e desmistificar todo o assunto para você? ... Claro!

Para começar, a quebra de linha não pode ser casada, aliás, pode. Ou melhor: depende.

Então primeiro vamos entender o que exatamente é uma quebra de linha. Temos o cifrão `$` que indica o **fim da linha**, que é o que vem antes da **quebra** da linha. Lembre-se da máquina de escrever, ao chegar ao final da linha você puxa o bracinho e o carretel gira, esta seria a quebra de linha, que no Linux já vem de brinde com o retorno do carretel. Trrrrrrr tshhhhhh plin!

Então se você tem um arquivo como este:

```
- uma serra pequena
- um alicate de bico com cabo
de borracha
- uma chave de fenda
```

E quer juntar a linha do alicate que se quebrou, o que fazer? A tentativa óbvia seria `/alicate/ s/\n//` para apagar o `\n`, mas isso não funciona. E lembrando que a primeira parte do comando `s` nada mais é do que uma busca, então similarmente o endereço `/cabo\nde borracha/` não vai funcionar como esperado.

Como apagar uma quebra de linha, juntando duas linhas

Temos um problema: não podemos casar a quebra de linha, então como apagá-la?

Precisaremos usar de engenhosidade aqui. Lembra do comando `N`? Ele gruda a **próxima** linha no padrão, separando-a da linha atual por um `\n`. Então após aplicado o comando `N`, temos duas linhas juntas numa string só, e surpresa: esse `\n` pode ser usado na busca!

Então para nosso exemplo acima, quando estivermos na linha do alicate, grudaremos a próxima linha, apagaremos o `\n` e pronto! A quebra de linha foi embora. Acompanhe:

```
prompt$ sed '/alicate/ N ; s/\n//' arquivo
- uma serra pequena
- um alicate de bico com cabode borracha
- uma chave de fenda
```

Ôpa! Quase isso. Simplesmente apagar o `\n` vai grudar as duas palavras, então o trocaremos por um espaço ao invés disso.

```
prompt$ sed '/alicate/ N ; s/\n/ /' arquivo
- uma serra pequena
- um alicate de bico com cabo de borracha
```

```
- uma chave de fenda
```

Aaaaaaaaah, agora sim. Vamos ver o debug para visualizar esse `\n`?

```
10  PATT:- um alicate de bico com cabo$
11  HOLD:$
12  COMM:/alicate/ N
13  PATT:- um alicate de bico com cabo\nde borracha$
14  HOLD:$
15  COMM:s/\n/ /
16  PATT:- um alicate de bico com cabo de borracha$
17  HOLD:$
```

Como podemos ver lá está a linha do alicate no padrão (L10). Depois de aplicado o `N` (L12), lá veio a próxima linha, e ficou o `\n` enfiado no meio das duas (L13). Daí ficou fácil, chegou o `s` (L15) e trocou o separador pelo espaço em branco (L16).

Como apagar todas as quebras de linha, deixando tudo numa só linha

Mas o do alicate foi para apenas uma linha. E se precisássemos fazer isso no arquivo todo, juntando todas as linhas numa só? Vamos usar agora o arquivo de números para ver melhor.

Para facilitar o trabalho dos descrentes, o jeito não-sed seria com o `tr`:

```
tr -d '\012' < numeros.txt
```

Pensando logicamente, se antes aplicamos o `N` endereçado à linha do alicate, agora basta não colocar o endereço e ele será aplicado para todas as linhas!

```
prompt$ sed 'N ; s/\n/ /' numeros.txt
um dois
três quatro
cinco seis
```

Ei, o sed está maluco, ele pegou de dois em dois! Vamos ver o que aconteceu:

```
1  PATT:um$
2  HOLD:$
3  COMM:N
4  PATT:um\ndois$
```

```
 5  HOLD:$
 6  COMM:s/\n/ /
 7  PATT:um dois$
 8  HOLD:$
 9  um dois
10  PATT:três$
11  HOLD:$
12  COMM:N
13  PATT:três\nquatro$
```

Ele grudou a segunda linha (L4), apagou a quebra (L6) mas depois começou de novo na terceira linha (L10). É claro! Sabemos que ao chegar ao final do script, o sed lê a próxima linha e volta para o começo do script.

Como fazer um laço

O que precisamos aqui é fazer um laço (loop), impedir que o sed chegue ao final do script, e ao invés disso, voltar ao começo e sempre reexecutar o `N`.

Para isso, fazemos uma marca no começo do script, chamaremos de "inicio", e sempre voltaremos para ela em todas as linhas. Traduzindo isso para sed, fica:

```
:inicio ; N ; s/\n/ / ; b inicio
```

No sed da GNU esse script dará certo, mas em alguns seds ele sairá com erro pois o `N` foi aplicado na última linha (ver capítulo do `N`). Então colocaremos um condicional aqui, para não aplicar o `N` na última linha: `$!N`, ou falando a língua do sed: aplique o N para todas as linhas, exceto a última.

```
:inicio ; $! N ; s/\n/ / ; b inicio
```

Mmmmmmm, para os já iniciados em programação, notaram que o que acabamos de fazer foi um laço infinito? Se dissemos ao sed para sempre voltar ao começo do script, ele fará isso infinitamente, consumindo toda a CPU de seu computador. Legal né? &:)

Como fazer um laço com condicional de saída

O que precisamos aqui é colocar um condicional que diga: se acontecer tal coisa, saia do laço. O que seria essa "tal coisa"? Como queremos grudar o arquivo todo numa linha só, devemos aplicar o N até a última linha. Ôpa! Achamos nosso condicional: se estiver na última linha, saia do laço.

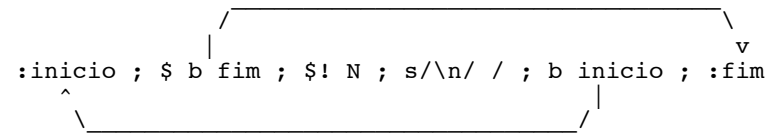
Vamos ver como fazer esse condicional. Sabemos como representar a última linha ($), mas como sair do laço? Vamos analisar nosso script:

```
:inicio ; $! N ; s/\n/ / ; b inicio
   ^                          |
    \__________________________/
```

Esse é o laço, precisamos de uma rota alternativa para o fim do script:

```
                  /_____________________________________________\
                 |                                               v
:inicio ; $<alguma-coisa> ; $! N ; s/\n/ / ; b inicio   <destino>
   ^                                                |
    \_____________________________________________/
```

Ei, se formos ver direto, essa saída do laço é a mesma técnica que foi usada no próprio laço! Basta fazermos uma marca no fim e pularmos até ela:



Então estamos no laço, sempre voltando ao começo, mas quando chegarmos na última linha, pulamos para o fim. Ei ei ei! Notaram que na última linha pularemos direto para o fim e o N nem será executado? Sendo assim, ele não precisa mais daquele condicional que tínhamos colocado antes:

```
prompt$ sed ':inicio ; $ b fim ; N ; s/\n/ / ; b inicio ; :fim' numeros.txt
um dois três quatro cinco seis
```

YOU WIN. PERFECT!

Como fazer um laço com condicional de entrada

Um outro tipo de condicional que podemos fazer é o contrário, colocando um condicional que faça o script voltar ao começo: se tal coisa, volte. Assim o laço fica condicionado, e só ocorre se a condição for satisfeita.

Além do comando `b`, também podemos dar pulos com o comando `t`, que só executa o pulo **se** o último comando `s` fez uma substituição. Ôpa! Aqui está a palavra mágica "se". Então o próprio `t` já é um comando com condicional.

Então basicamente usamos o mesmo exemplo anterior com a marca `:inicio` e trocamos o `b` pelo `t`:

```
:inicio ; N ; s/\n/ / ; t inicio
```

Mas lembrando do problema do `N` na última linha, precisamos colocar de volta aquele outro condicional esperto:

```
prompt$ sed ':inicio ; $! N ; s/\n/ / ; t inicio' numeros.txt
um dois três quatro cinco seis
```

Então em todas as linhas o `N` grudará a próxima, com o `\n` de separador e o `s` o trocará por um espaço. Como a substituição foi feita, o `t` pulará de volta ao início e o ciclo se fecha até a última linha.

Na última linha, o condicional `$!N` impede que o `N` seja executado, logo não haverá `\n`, então o `s` não vai ter o que substituir e assim o `t` não pulará de volta ao começo, finalizando o script.

FINISH HIM! FATALITY!

Como fazer um laço utilizando o reserva

Uma outra maneira de fazer um laço, é utilizar o espaço reserva e ir "estocando" lá as linhas que se deseja, e ao encontrar o ponto de parada, carregar o estoque para o padrão e trabalhá-lo.

Ainda na nossa árdua tarefa de juntar todas as linhas numa só, vamos ver como fica nosso script utilizando esta técnica:

```
prompt$ sed 'H ; $! d ; g ; s/\n/ /g' numeros.txt
 um dois três quatro cinco seis
```

Ganhamos um espaço em branco no começo da linha que pode ser facilmente apagado com um `s/.//` no final do script. Mas fora isso, como funciona o laço?

```
 1  PATT:um$
 2  HOLD:$
 3  COMM:H
 4  PATT:um$
 5  HOLD:\num$
 6  COMM:$ !d
 7  PATT:dois$
 8  HOLD:\num$
 9  COMM:H
10  PATT:dois$
11  HOLD:\num\ndois$
```

O `H` (L3) juntou o conteúdo do padrão ("um") no reserva, separando-o do conteúdo já existente por um `\n` (L5). Como é a primeira linha e o reserva estava vazio, não tem nada antes do `\n`. O próximo comando `d` está condicionado a **não** ser executado na última linha, ou seja, ele será aplicado para todas as outras. Como ainda estamos na primeira linha, ela foi apagada, e como o `d` após apagar já carrega a próxima linha no padrão e faz o script sed voltar ao começo, novamente o `H` vai ser executado (L9), fechando o ciclo. E assim o padrão vai sendo juntado no reserva (L11), linha após linha até a última, só executando os primeiros comandos `H; $!d`:

```
31  PATT:seis$
32  HOLD:\num\ndois\ntrês\nquatro\ncinco$
33  COMM:H
34  PATT:seis$
35  HOLD:\num\ndois\ntrês\nquatro\ncinco\nseis$
36  COMM:$ !d
37  PATT:seis$
38  HOLD:\num\ndois\ntrês\nquatro\ncinco\nseis$
39  COMM:g
40  PATT:\num\ndois\ntrês\nquatro\ncinco\nseis$
41  HOLD:\num\ndois\ntrês\nquatro\ncinco\nseis$
42  COMM:s/\n/ /g
43  PATT: um dois três quatro cinco seis$
44  HOLD:\num\ndois\ntrês\nquatro\ncinco\nseis$
45   um dois três quatro cinco seis
```

Ao chegar na última linha, ela também será juntada no reserva, mas agora o condicional `$!d` faz com que o `d` não seja aplicado, então o script pode continuar finalmente para o próximo comando `g` (L39). Com ele, recuperamos todo o conteúdo do nosso estoque de linhas, e o próximo comando `s` se encarrega de trocar **todos** (note o modificador `s///g`) os `\n` do padrão por espaços de uma só vez (L43).

Resumindo: estocamos as linhas no reserva, pegamos de volta e formatamos.

Como substituir alguma coisa por uma quebra de linha

Uma tarefa comum é você ter um arquivo com vários dados separados por algum delimitador qualquer e querer que cada um dos dados fique em uma linha, ou seja, trocar o delimitador por uma quebra de linha.

Mas com outras versões do sed que não entendem este escape, a quebra de linha deve ser inserida **literalmente** e deve ser escapada:

```
prompt$ sed 's/ /\

prompt$ /g' texto.txt
```

No sed da GNU, a partir da versão **3.02.80**(*), foi adicionado o `\n` como escape válido dos dois lados do comando `s///`. Com isso a tarefa de colocar cada palavra numa linha isolada, ou seja, trocar espaços em branco por quebras de linha, fica trivial:

```
prompt$ sed 's/ /\n/g' texto.txt
```

MAIÚSCULAS/minúsculas

A princípio parece ser um assunto tão bobinho, que não tem muito o que falar, mas a experiência mostra que o assunto maiúsculas/minúsculas sempre vai render problemas e soluções diversos. Vamos fazer um apanhado de todos os tópicos que envolvem o assunto, e não deixar nem uma sombra de dúvida no caminho.

Vamos começar com a notação, que pode ser novidade para muitos. Como vamos falar de tipos de caracteres e conversões entre eles, a linguagem tipográfica serve para definirmos os termos que identificam as ações e elementos do maravilhoso mundo das letras. Então vamos lá: letras em maiúsculas são letras com **caixa alta**, e em minúsculas, **caixa baixa**. Então **inverter a caixa** é o nome que se dá à troca de maiúsculas por minúsculas e vice-versa. Destes termos também podemos derivar: **levantar a caixa**, que seria a "maiusculização" (toupper) e **abaixar a caixa** seria a "minusculização" (tolower).

Como deixar todo o texto em caixa alta (ou baixa)

Uma tarefa das mais básicas e corriqueiras quando se manipula textos é precisar padronizar a caixa das letras, convertendo todas as palavras para caixa alta, ou baixa.

A ferramenta UNIX que faz isso é o `tr`, que possui o clássico `tr [A-Z] [a-z]` para abaixar a caixa de um texto. Ah! Em versões mais recentes os colchetes não são mais necessários então `tr A-Z a-z` basta. O sed possui também um comando de tradução de caracteres, o `y`. Essa tradução é feita caractere a caractere, por exemplo:

```
sed 'y/áéó/aeo/' arquivo.txt
```

Este comando trocará as letras com acento agudo pela própria letra sem acento. Diferente do comando `s///`, que é "troque isso por aquilo", o `y///` é "para cada letra daqui, troque-a pela de lá". Então no comando anterior, estamos dizendo ao sed: "troque á por a, troque é por e, troque ó por o".

Outra diferença do `s`, em que se tem controle sobre o número de substituições que serão feitas, com o `y` é tudo ou nada, ele converte **todas** as letras da linha que estiverem no comando. Veja:

```
prompt$ echo AAAABBBAAAA | sed 'y/AB/ab/'
aaaabbbaaaa
```

E o `y` é sempre assim, em pares. Por isso **sempre** deve haver o mesmo número de letras na primeira e na segunda parte do comando, senão o sed acusará erro.

Outro detalhe é que o `y` por causa dessa paridade, só aceita caracteres literais. Nada de expressões regulares ou qualquer outro caractere especial. Então o comando similar ao `tr` é inválido:

```
sed 'y/A-Z/a-z/'                # ERRADO!
```

Este comando diz: "troque A por a, troque - por -, troque Z por z", e só. Para conseguirmos converter de maiúsculas para minúsculas, temos que colocar **todas** as letras, uma por uma. Sim, isso é muito chato.

```
sed 'y/ABCDEFGHIJKLMNOPQRSTUVWXYZ/abcdefghijklmnopqrstuvwxyz/'
```

Essa minhocona fará o serviço de deixar todo o texto em caixa baixa. Será que você está bem atento? Apesar de já estar um comando gigantesco, não está faltando nada aí não? É claro! Estão faltando nossas queridas letras acentuadas. Se não as colocarmos, continuarão em caixa alta mesmo após a conversão. Respire fundo, vamos lá:

```
sed 'y/ABCDEFGHIJKLMNOPQRSTUVWXYZÁÀÃÂÉÊÍÓÕÚÜÇ/abcdefghijklmnopqrstuvwxyzáàãâéêíóõúüç/'
```

Ufa! Quanta coisa para resolver algo aparentemente tão simples. Caso tenhamos nos perdido no meio do caminho, e olha que isso é muito fácil de acontecer, o sed avisará:

```
sed 'y/FGHI/fg/'
sed: -e expression #1, char 10: strings for y command are different lengths
```

Para fazer o caminho inverso, deixando tudo em maiúsculas, basta inverter as partes do comando:

```
sed 'y/abcdefghijklmnopqrstuvwxyzáàãâéêíóõúüç/ABCDEFGHIJKLMNOPQRSTUVWXYZÁÀÃÂÉÊÍÓÕÚÜÇ/'
```

Uma dica muito, mas **muito** boa para a utilização do `y`, é usar outro delimitador. Assim como para endereços e para o comando `s`, podemos usar qualquer caractere ASCII como delimitador do comando. Então ao invés da barra /, use a **quebra de linha** como delimitador:

```
sed 'y
abcdefghijklmnopqrstuvwxyzáàãâéêíóõúüç
ABCDEFGHIJKLMNOPQRSTUVWXYZÁÀÃÂÉÊÍÓÕÚÜÇ
'
```

Assim facilita muito, já ficando visualmente notável caso as duas partes do comando tenham tamanhos diferentes, e também visualizamos na hora quais são os pares de caracteres, pois estão um embaixo do outro.

Claro que para digitar na linha de comando, ficar quebrando linhas assim é incômodo, mas dentro de um arquivo, é a melhor opção para não confundir.

Como inverter a caixa de todas as letras

Agora que já sabemos como usar o comando y para aumentar ou abaixar a caixa das letras, e sabendo que ele funciona em pares de letras, fica fácil inverter a caixa de todas as letras da linha.

Primeiro façamos um teste rápido para ver se ele não converte duas vezes a mesma letra:

```
prompt$ echo aaaAAA | sed 'y/aA/Aa/'
AAAaaa
```

Beleza! Então para fazer o superconversor basta pacientemente compor a superminhoca num superscript!

```
# inverte_caixa.sed   (dica: usa quebra de linha como separador)
y
abcdefghijklmnopqrstuvwxyzáàãâéêíóõúüçABCDEFGHIJKLMNOPQRSTUVWXYZÁÀÃÂÉÊÍÓÕÚÜÇ
ABCDEFGHIJKLMNOPQRSTUVWXYZÁÀÃÂÉÊÍÓÕÚÜÇabcdefghijklmnopqrstuvwxyzáàãâéêíóõúüç
```

Admita: o sed é lindo não é? Temos um script de apenas 3 linhas, sem nenhum símbolo, apenas letras. E isso inverte a caixa de todas as letras. Vamos testar?

```
prompt$ echo "SED significa Stream EDitor" | sed -f inverte_caixa.sed
sed SIGNIFICA sTREAM edITOR
```

Como levantar a caixa da primeira letra de cada linha

Antes de mais nada, não pense que vai ser fácil. Dê uma looonga respirada, vista o capacete anti-maionese, e vamos viajar!

Lembra que o comando `y` troca sempre **todas** as letras da linha? Então como fazer para que ele troque apenas a primeira letra e mantenha as outras intactas? Mmmmmmmmm, precisamos de um pouco de mágica aqui.

Temos dois componentes: a linha e o comando. Como não tem jeito de mudar o `y`, que sempre pega a linha toda, só nos resta então mudar a própria linha. A solução é isolar apenas a primeria letra no PADRÃO, deixando-a **sozinha** para que o `y` faça o seu serviço, e depois rejuntar o resto da linha novamente. É, é muito feio, mas é o que dá pra fazer.

Primeiro problema: isolar a primeira letra. Antes de mais nada, guardamos uma cópia da linha original no RESERVA (comando h). Em seguida, apagamos todo o resto da linha, deixando apenas a primeira letra. Geralmente a primeira letra da frase já está bem no começo da linha, então a expressão regular `^[a-z]` serve para representá-la. Mais adiante vamos refinar esta expressão, mas por enquanto deixemos assim. Então se já sabemos como identificar a primeira letra, apagamos tudo menos ela:

```
h ; s/^\([a-z]\).*/\1/
```

Supondo a frase "unidos venceremos!", após aplicado o comando acima teremos:

```
PATT: u$
HOLD: unidos venceremos!$
```

Com a primeira letra isolada, agora podemos levantar sua caixa tranquilamente com a famosa minhocona `y/abcde.../ABCDE.../`.

Beleza, então agora temos que voltar o resto da linha que está no RESERVA, mas antes temos que também **retirar** a primeira letra dela, pois ela é uma cópia da linha original lembra? Devemos trocar os registradores de lugar e apagar a letra:

```
x ; s/^[a-z]//
```

Agora temos em mãos:

```
PATT: nidos venceremos!$
HOLD: U$
```

Certo, agora basta grudar o PADRÃO no RESERVA (comando H), e apagar o `\n` que vai ficar entre eles:

```
H ; g ; s/\n//
```

Compondo o script todo fica:

```
h ; s/^\([a-z]\).*/\1/           ;# guarda cópia, deixa só a primeira letra
y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/   ;# levanta a caixa
x ; s/^[a-z]//                   ;# arranca a primeira letra da cópia
H ; g ; s/\n//                   ;# remonta a linha
```

Sem querer ser chato mas sendo, esse script ainda não é 100% funcional. Até aqui assumimos que geralmente a primeira letra da frase já está bem no começo da linha. Mas como isso nem sempre é verdade, temos que melhorar nossa expressão regular `^[a-z]`.

Antes da primeira letra, podemos ter espaços em branco, então é prudente também colocar esta condição na expressão: `^ *[a-z]`. É, mas além de espaços, também podem ter TABs, então é melhor garantir e fazer `^[[:blank:]]*[a-z]`. E o monstro vai crescendo...

Sendo mais chato ainda: e os acentos? Temos palavras no português que começam com acentos? Deixa ver: água, época, ódio, ócio, órbita... Ó céus, também precisamos incluir os acentos para que nosso script não falhe. Vamos lá: `^[[:blank:]]*[[:lower:]]`. Agora chega! Esse monstrinho representa a primeira letra de uma linha.

Vamos atualizar nosso script com essa informação nova. Ah! Lembre que o `y///` agora também precisará conter os caracteres acentuados!

```
# _L_inha.sed
```

```
h ; s/^\([[:blank:]]*[[:lower:]]\).*/\1/     ;# guarda cópia, deixa só a primeira letra
y/abcdefghijklmnopqrstuvwxyzáàãâéêíóõúüç/ABCDEFGHIJKLMNOPQRSTUVWXYZÁÀÃÂÉÊÍÓÕÚÜÇ/
x ; s/^[[:blank:]]*[[:lower:]]//             ;# arranca a primeira letra da cópia
H ; g ; s/\n//                               ;# remonta a linha
```

Que trabalheira para fazer uma coisa tão simples hein? Pois é, essa é a eterna diversão de se usar o sed &:D Apenas como comparativo, para fazer a mesma coisa no vim basta um `s/[[:lower:]]/\u&/`.

Como levantar a caixa da primeira letra de cada palavra

Se o tópico anterior de levantar a caixa só da primeira letra da linha já era complicado, imagina este: cabeludo. Tão cabeludo que precisaremos de três seds.

- O primeiro sed quebra a linha em **cada** palavra, deixando uma por linha
- O segundo sed é o _L_inha.sed do tópico anterior, já que temos uma palavra por linha
- O terceiro sed reajuntará as palavras numa linha.

O grande conselho é: use outro programa. Mas se o sed é a única opção, ou se S&M significa algo para você, vamos lá!

Para quebrar a linha em cada palavra, temos um carinha muito útil que é o `\<`, que representa um "início de palavra", então basicamente trocamos um "início de palavra" por uma "quebra de linha". Na língua do sed, isso é:

```
# uma-palavra-por-linha.sed

s/\</\
/g
```

Como o segundo sed já temos, já podemos testar os dois primeiros passos!

```
prompt$ echo "eu nunca farei isso." | sed -f uma-palavra-por-linha.sed | sed -f _L_inha.sed
Eu
Nunca
Farei
Isso.
```

E é claro, para juntar todas as linhas, há várias opções descritas no tópico "Como apagar todas as quebras de linha, deixando tudo numa só linha". Colocando uma versão compacta do laço com condicional de entrada, temos:

```
# junta-linhas.sed

:i ; $!N ; s/\n/ / ; ti
```

O comando completo fica algo deselegante, mas funciona:

```
prompt$ echo "eu nunca farei isso." | sed -f uma-palavra-por-linha.sed | sed -f _L_inha.sed
| sed -f junta-linhas.sed
Eu Nunca Farei Isso.
```

Se possível, é bom colocar todos estes comandos dentro de um script shell, e então você terá quatro arquivos para fazer uma tarefa simples!

```
#!/bin/sh
```

```
# _P_alavra.sh - levanta a caixa da primeira letra de cada palavra

sed -f uma-palavra-por-linha.sed |
sed -f _L_inha.sed |
sed -f junta-linhas.sed
```

Mas pelo menos a linha de comando fica menor:

```
prompt$ echo "eu nunca farei isso." | ./_P_alavra.sh
```

E já que chegamos até aqui, podemos ter apenas um arquivo, colocando o conteúdo dos três seds direto dentro do script shell:

```
#!/bin/sh
# _P_alavra.sh - levanta a caixa da primeira letra de cada palavra

sed 's/\</\
/g' |

sed '
h ; s/^\([[:blank:]]*[[:lower:]]\).*/\1/
y/abcdefghijklmnopqrstuvwxyzáàãâéêíóõúüç/ABCDEFGHIJKLMNOPQRSTUVWXYZÁÀÃÂÉÊÍÓÕÚÜÇ/
x ; s/^[[:blank:]]*[[:lower:]]//
H ; g ; s/\n//' |

sed ':i ; $!N ; s/\n/ / ; ti'
```

Novamente, o comando equivalente no vim seria mais simples: `s/\<./\u&/g`

Como ignorar a diferença de maiúsculas e minúsculas

É normal ao se pesquisar dados, querer ignorar a caixa do texto procurado, ou seja, tanto faz se o texto estiver em maiúsculas ou minúsculas, ambos são válidos.

A grande maioria dos editores de texto, leitores de email, navegadores e outros programas que manipulam texto têm a opção de ignorar a caixa quando pesquisando por um texto. No vim é `:set ic`, no grep é através da opção `-i` e no sed: depende.

Mais uma vez entramos num tópico onde a versão do sed influi muito nas características e poderes disponíveis, então se prepare para a chuva de possibilidades para algo aparentemente tão inocente.

Suponhamos a tarefa de apagar o telefone do amigo Pedro, que es...

A maneira mais tradicional, conservadora e garantida, que com certeza funcionará em todas as versões de sed existentes, é especificar uma a uma as possibilidades de cada letra do padrão, utilizando expressões regulares. Veja:

```
prompt$ sed '/[..][..][..]/d' bla.txt
```

Usando os colchetes, explicitamos as alternativas possíveis para cada posição de caractere do padrão.

Esta é claro, não é uma solução prática, pois sempre precisamos saber de antemão qual é o padrão a ser procurado para colocar as alternativas. Mas se este padrão é mutável, ou pode ser definido pelo usuário, é preciso pré-processá-lo, colocando automaticamente os colchetes e todas as alternativas, antes de passá-lo ao sed. Algo como:

```
- coloque o padrão todo em minúsculas
- para cada letra do padrão, componha:
    [
    + a própria letra
    + a letra em caixa alta
    + ]
```

Já imaginou todo esse trabalho só para pesquisar um texto? Infelizmente para uma grande parte das versões de sed existentes, é isso ou...

Se a saída puder ser toda em minúsculas ou maiúsculas, outra tática é antes de mais nada usar o comando y para converter a linha toda do arquivo de texto todo para maiúsculas, e usar o padrão também em maiúsculas. Essa tática é muito usada em programação, para comparar conteúdo de variáveis, por exemplo:

```
if ( toupper($comando) == 'SELECT' ) ; then ...
```

Em sed, seria:

```
# após ler a linha do arquivo texto, levante sua caixa
y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/

# procure o padrão e execute os comandos
/MEU-PADRAO/ {
  ...
}
```

Mas para alívio dos amantes do sed, os programadores da GNU incluíram um modificador especial de endereço `I` (de Ignore-case) a partir da versão **3.01-beta1**. A sintaxe pode parecer estranha, mas acostuma. No endereço, basta colar o modificador `I` logo após fechada a segunda barra delimitadora:

```
prompt$ sed '/root/I d' texto.txt
```

E também no comando `s///`, temos o mesmo modificador `I`, que pode ser incluído no final, junto com os outros modificadores já existentes `g`, `p`, `w`, ou números. Dessa maneira a primeira parte do comando, que nada mais é senão um endereço, é pesquisada ignorando a caixa das letras. Exemplo:

```
prompt$ sed 's/root/administrador/Ig' texto.txt
```

Cumbuca

Aqui é aquela seção que você vai ler agora apressadamente, e depois quando precisar, vai voltar para procurar um exemplo.

Exemplos Úteis

"Olha, não é que eu seja preguiçoso, eu já aprendi bastante coisa de sed, já me viro sozinho, manjo de fazer endereços, de fazer scripts sed e tudo mais. Mas será que não dava pra colocar num cantinho aí uns exemplinhos mastigados daquelas tarefas bem rotineiras e chatas?"

Apagar todas as tags HTML	`s/<[^>]*>//g`
Obter o número da linha onde está a palava VERDE	`/VERDE/= ; d`
Remover todas as linhas em branco	`/^ *$/d`
Remover todos os comentários	`/^ *#/d`
Comentar a linha que contém a palavra VERDE	`/VERDE/s/^/#/`
Desacentuar um arquivo	`y/ÁÀÃÂÉÊÍÓÕÚÜÇáàãâéêíóõúüç/AAAAEEIOOUUCaaaaeeiooouuc/`
Juntar todas as linhas numa só	`H ; $!d ; g ; s/\n/ /g`
Remover todos os colchetes	`s/[][]//`
Remover todos os colchetes e seu conteúdo	`s/\[[^]]*]//`
Tirar caracteres de controle como ^[33;43m (^[é Ctrl+v,ESC)	`s/^[\[[0-9;]*m//`

Emulando comandos UNIX/Linux

A utilidade do sed às vezes supera os limites do palpável. Se imagine numa máquina que só tenha a instalação padrão do Windows. É um golpe duro, um pesadelo, ser privado de todas as suas ferramentas UNIX/Linux que você está acostumado a utilizar no dia-a-dia, é como trabalhar com os braços amarrados.

Mas espere, nem tudo está perdido! Se você conseguir colocar nessa máquina pelo menos o sed, poderá usá-lo para emular alguns dos seus comandos queridos. Emular significa imitar, simular, ter comportamento idêntico.

Então instale o SED.EXE (DOS) e corra para o abraço!

```
comando      |  emulação
-------------+-----------------------------------------
cat          |  sed :
tac          |  sed 1!G;h;$!d
grep         |  sed /padrão/!d
grep -v      |  sed /padrão/d
head         |  sed 10q
head -1      |  sed q
tail         |  sed -e :a -e $q;N;11,$D;ba
tail -1      |  sed $!d
cut -c 10    |  sed s/\(.\)\{10\}.*/\1/
cut -d: -f4  |  sed s/\(\([^:]*\):\)\{4\}.*/\2/
tr A-Z a-z   |  sed y/ABCDEFGHIJKLMNOPQRSTUVWXYZ/abcdefghijklmnopqrstuvwxyz/
```

```
tr a-z A-Z    |  sed y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
tr -s ' '     |  sed s/ \+/ /g
tr -d '\012'  |  sed H;$!d;g;s/\n//g
wc -l         |  sed -n $=
uniq          |  sed x;G;/^\(.*\)\n\1$/d;g
rev           |  sed /\n/!G;s/\(.\)\(.*\n\)/&\2\1/;//D;s/.//
basename      |  sed s,.*/,,
dirname       |  sed s,[^/]*$,,
cp            |  sed w arquivo-novo
```

Agradecimentos

Em Outubro de 2002, alguns amigos toparam ajudar com o livro. Eu lhes mostrei o texto e eles sugeriram melhorias, arrumaram erros ortográficos e fizemos uma grande discussão que resultou no aumento de qualidade do conteúdo. A esse grande time meu MUITO OBRIGADO:

- Eliphas Levy Theodoro
- Érico "DyNaMiTe"
- Fernando Braga
- Gentil de Bortoli Júnior
- Julio Cezar Neves
- Leslie Harlley Watter
- Luciano Espírito Santo
- Marcelo Pereira
- Rodrigo Stulzer
- Rubens Queiroz de Almeida
- Thobias Salazar Trevisan