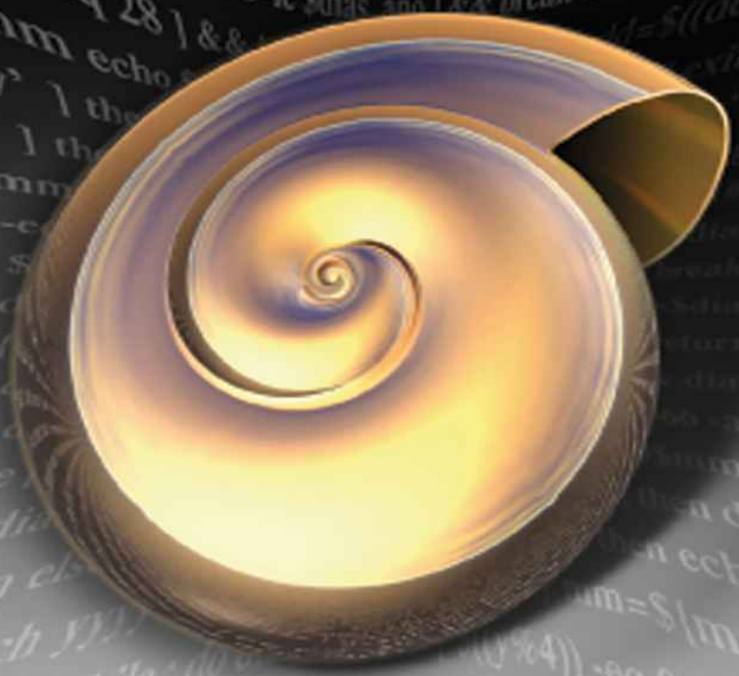
SHELL SCRIPT

PROFISSIONAL

Aurélio Marinho Jargas



novatec

SHELL SCRIPT
PROFISSIONAL

Aurelio Marinho Jargas

Novatec

Editor: Rubens Prates
Revisão gramatical: Maria Rita Quintella
Capa: Alex Lutkus

ISBN: 978-85-7522-576-9

Histórico de edições impressas:

Janeiro/2016 Oitava reimpressão
Novembro/2014 Sétima reimpressão
Agosto/2013 Sexta reimpressão
Setembro/2012 Quinta reimpressão
Outubro/2011 Quarta reimpressão
Janeiro/2011 Terceira reimpressão
Março/2010 Segunda reimpressão
Outubro/2008 Primeira reimpressão
Abril/2008 Primeira edição

Novatec Editora Ltda.
Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP – Brasil
Tel.: +55 11 2959-6529
E-mail: *novatec@novatec.com.br*
Site: *www.novatec.com.br*
Twitter: *twitter.com/novateceditora*
Facebook: *facebook.com/novatec*
LinkedIn: *linkedin.com/in/novatec*

Ao Rubens Prates que, durante quatro longos invernos, esperou.

Sumário

Mensagem final

Agradecimentos

À minha mãe, Geny, que me apoia em tudo, incondicionalmente. Sempre preocupada com meu bem-estar, seu carinho e cuidados são essenciais. É meu exemplo de vida, mostrando que a dedicação ao trabalho traz os resultados almejados. Te amo!

À minha noiva, Mog, que me inspira. Seu carinho, compreensão e incentivo foram fundamentais para que eu retomasse a redação do livro. Seu interesse constante e sua cobrança amigável mantiveram-me motivado até a finalização da obra. Sem o seu apoio, eu não teria conseguido. É tu.

À minha família que me incentiva a continuar escrevendo, em especial: Karla, Gabriel, Felipe, Larissa, Marilei, Edenilson, Gerson e Altemir. Valeu!

Ao Rubens Prates, editor da Novatec, que com vários e-mails e telefonemas ao longo de quatro anos não deixou apagar a chama, sempre me incentivando a escrever o já quase mítico livro de shell. Sua paciência é infinita e seu entusiasmo inabalável. Setembro de 2003... Rubens, este livro não existiria sem a sua persistência.

Ao Alex Lutkus, que criou a capa dos sonhos para este livro. Dono de um senso de humor invejável, o processo de criação foi uma divertida troca de e-mails com ideias, histórias, conceitos, simbolismos, devaneios e malacologia. Deveria ser Alex Lúdicus!

Ao Thobias Salazar Trevisan, meu amigo e parceiro de Funções ZZ, que contribuiu com muitas ideias para o livro de shell que íamos escrever a quatro mãos. É uma pena que seu tempo livre ficou escasso, pois teria sido uma ótima experiência. Mas tem legado teu aqui, guri.

Aos amigos Rodrigo Stulzer Lopes, João Cláudio Mussi de Albuquerque e Carla Faria, pelo apoio constante e pelas ideias para o título e a capa do livro. Vocês podiam abrir uma empresa de consultoria!

Ao Rubens Queiroz de Almeida, que há tantos anos divulga meus

projetos em sua incansável Dicas-L (`www.dicas-l.com.br`). Seu apoio é sempre tão espontâneo e entusiasmado que me incentiva em qualquer trabalho que eu faça. Amigo, minha dívida contigo é eterna.

Ao Franklin Carvalho, que me telefona várias vezes ao ano, todos os anos, para me instigar. Sempre com ideias novas, palavras de incentivo e muito otimismo, consegue arrancar um sorriso até mesmo do mais desanimado dos nerds. Seu apoio foi muito importante para vários projetos, e com este livro não foi diferente. Frank, você é o cara.

Ao Julio Cezar Neves que foi quem começou essa história toda em junho de 2003, quando me convenceu a ministrar cursos de shell. Então, escrevi uma apostila, que anos depois serviria de pontapé inicial para este livro. O Julio tem seu próprio livro de shell, Programação Shell Linux, que ensina os conceitos básicos de maneira bem-divertida. Ó o jabá: recomendo!

Ao Arnaldo Carvalho de Melo, meu guru, que me ensinou o bê-á-bá do shell, em 1997, quando eu sabia apenas usar o MS-DOS. Ele também me mostrou conceitos avançados que aguçaram minha curiosidade e, desde então, nunca mais parei de brincar com shell. Acme, seu investimento no aprendiz aqui deu certo. Pelo menos na parte de shell, já que no Quake sempre fui ruim de mira :)

Aos amigos de Matinhos, de Curitiba e da Internet, por sempre perguntarem "E o livro?". Vocês não fazem ideia de como isso foi importante para eu não desistir.

Aos amigos que contribuíram para que este livro se tornasse uma realidade: Andreyev Dias de Melo, Augusto Campos, Cícero Moraes, Felipe Kellermann, Frederico Argolo, Leandro Costa, Luciano Silveira do Espírito Santo, Paulo Ricardo Paz Vital, Ricardo Reiche, Rodrigo Baroni, Rodrigo Telles, Samuel Rettore, Sulamita Garcia.

Aos leitores atentos que sugeriram melhorias ao texto do livro: Cristiano Amaral, Douglas Soares de Andrade, Eliphas Levy Theodoro, Francisco José de Marco Lopes dos Santos, Franco Keneti Doi, Leonardo Saraiva, Luiz Alberto Horst, Marcelo Charan, Renato de Matos, Thiago Coutinho, Tiago Oliveira de Jesus e Vlademir M. de Moraes.

Aos mais de 400 compradores de minha Apostila de Shell que acreditaram na negociação informal homem a homem e não colocaram o conteúdo na Internet, provando que a confiança e o respeito ainda prevalecem. Agradecimento especial aos leitores que responderam à pesquisa de opinião. Com seu retorno positivo, fui convencido de que poderia ir além e fazer um livro sobre shell.

Obrigado a você pelo interesse e pela compra deste livro. É um incentivo que me dá forças para continuar escrevendo. Espero que fique satisfeito(a) com a aquisição.

Por último, porém não menos importante, agradeço a Deus por ter colocado todas estas pessoas especiais em minha vida e tornar tudo isto possível. Obrigado por todas as realizações, obrigado por sempre estar por perto.

Sobre o autor

Aurelio Marinho Jargas é pesquisador, programador e escritor. Mora na praia, na pacata Matinhos-PR. Gosta de surfar, andar descalço na pedra e subir morros. Trabalha com shell script há mais de 10 anos, já escreveu vários artigos técnicos e ministrou cursos corporativos de shell avançado, tornando-se uma referência nacional no assunto. Criou as Funções ZZ, software famoso por seus mais de 70 miniaplicativos para a linha de comando. É autor do livro *Expressões Regulares – Uma abordagem divertida*. Em seu site `www.aurelio.net` há diversos textos didáticos e bem-humorados, sendo parada obrigatória para interessados em shell, sed, expressões regulares e software livre.

Prefácio

Em 1997, fiz meu primeiro script em shell.

Eu tinha 20 anos, era estagiário na Conectiva (hoje Mandriva) e ainda estava aprendendo o que era aquele tal de Linux. Hoje é tudo colorido e bonitinho, mas há uma década usar o Linux significava passar a maior parte do tempo na interface de texto, a tela preta, digitando comandos no terminal. Raramente, quando era preciso rodar algum aplicativo especial, digitava-se `startx` para iniciar o modo gráfico.

Meu trabalho era dar manutenção em servidores Linux, então o domínio da linha de comando era primordial. O Arnaldo Carvalho de Melo (acme) era meu tutor e ensinou o básico sobre o uso do terminal, comandos, opções, redirecionamento e filtragem (pipes). Mas seu ensinamento mais valioso, que até hoje reflete em minha rotina de trabalho, veio de maneira indireta: ele me forçava a ler a documentação disponível no sistema.

A rotina era sempre a mesma: o Arnaldo sentava-se e demonstrava como fazer determinada tarefa, digitando os comandos, para que eu visse e aprendesse. Porém, era como acompanhar uma corrida de Fórmula-1 no meio da reta de chegada. Ele digitava rápido, falava rápido e, com seu vasto conhecimento, transmitia muito mais informações do que minha mente leiga era capaz de absorver naqueles poucos minutos.

Quando ele voltava para sua mesa, então começava meu lento processo de assimilação. Primeiro, eu respirava fundo, sabendo que as próximas horas iriam fatigar os neurônios. Com aquele semblante murcho de quem não sabe nem por onde começar, eu dava um `history` para ver quais eram todos aqueles comandos estranhos que ele havia digitado. E eram muitos.

Para tentar entender cada comando, primeiro eu lia a sua mensagem de ajuda (`--help`) para ter uma ideia geral de sua utilidade. Depois, lia a sua man page, da primeira à última linha, para tentar aprender o que aquele comando fazia. Só então arriscava fazer alguns testes na linha de comando

para ver o funcionamento. Experimentava algumas opções, porém com cautela, pensando bem antes de apertar o Enter.

Ler uma man page era uma experiência frustrante. Tudo em inglês, não tinha exemplos e o texto parecia ter sido sadicamente escrito da maneira mais enigmática possível, confundindo em vez de ajudar. A pior de todas, com larga vantagem no nível de frustração, era a temida `man bash`, com suas intermináveis páginas que eu lia, relia, trelia e não entendia nada. Ah, meu inglês era de iniciante, o que tornava o processo ainda mais cruel.

A Internet não ajudava em quase nada, pois as fontes de informação eram escassas. Só havia algumas poucas páginas em inglês, com raros exemplos. Fóruns? Blogs? Lista de discussão? Passo-a-passo-receita-de-bolo? Esqueça. Era man page e ponto.

Mas com o tempo fui me acostumando com aquela linguagem seca e técnica das man pages e aos poucos os comandos começaram a me obedecer. Os odiosos *command not found* e *syntax error near unexpected token* tornaram-se menos frequentes. Deixei de perder arquivos importantes por ter digitado um comando errado, ou por ter usado > ao invés de >>.

Dica: Leia man pages. É chato, cansativo e confuso, mas compensa. Afinal, quem aprende a dirigir em um jipe velho, dirige qualquer carro.

Quanto mais aprendia sobre os comandos e suas opções, mais conseguia automatizar tarefas rotineiras do servidor, fazendo pequenos scripts.

Códigos simples, com poucos comandos, mas que poupavam tempo e garantiam a padronização na execução. Ainda tenho alguns deles aqui no meu `$HOME`, vejamos: mudar o endereço IP da máquina, remover usuário, instalação do MRTG, gerar arquivo de configuração do DHCPD.

Em alguns meses já eram dezenas de scripts para as mais diversas tarefas. Alguns mais importantes que outros, alguns mais complexos: becape, gravação de CDs, geração de páginas HTML, configuração de

serviços, conversores, wrappers e diversas pequenas ferramentas para o controle de qualidade do Conectiva Linux. Que versátil é o shell!

Mas nem tudo era flores. Alguns scripts cresceram além de seu objetivo inicial, ficando maiores e mais complicados. Cada vez era mais difícil encontrar o lugar certo para fazer as alterações, tomando o cuidado de não estragar seu funcionamento. Outros técnicos também participavam do processo de manutenção, adicionando funcionalidades e corrigindo bugs, então era comum olhar um trecho novo do código que demorava um bom tempo até entender o que ele fazia e por que estava ali.

Era preciso amadurecer. Mais do que isso, era preciso fazer um trabalho mais profissional.

Os scripts codificados com pressa e sem muito cuidado com o alinhamento e a escolha de nomes de variáveis estavam se tornando um pesadelo de manutenção. Muito tempo era perdido em análise, até realmente saber onde e o que alterar. Os scripts de "cinco minutinhos" precisavam evoluir para programas de verdade.

Cabeçalhos informativos, código comentado, alinhamento de blocos, espaçamento, nomes descritivos para variáveis e funções, registro de mudanças, versionamento. Estes eram alguns dos componentes necessários para que os scripts confusos fossem transformados em programas de manutenção facilitada, poupando nosso tempo e, consequentemente, dinheiro.

A mudança não foi traumática, e já nos primeiros programas provou-se extremamente bem-sucedida. Era muito mais fácil trabalhar em códigos limpos e bem-comentados. Mesmo programas complexos não intimidavam tanto, pois cada bloco lógico estava visualmente separado e documentado. Bastava ler os comentários para saber em qual trecho do código era necessária a alteração. Quanta diferença!

Dica: Escreva códigos legíveis. Cada minuto adicional investido em limpeza de código e documentação compensa. Não basta saber fazer bons algoritmos, é preciso torná-los acessíveis.

Todo este processo fez-me evoluir como profissional e como programador. Primeiro, aprendi a procurar por conhecimento em vez de códigos prontos para copiar e colar. Depois, aprendi o valor imenso de um código limpo e informativo.

Então, nasceu este livro de shell.

Ao olhar para o Sumário, você verá uma lista de tecnologias e técnicas para tornar seus scripts mais poderosos. Mas não quero que você simplesmente aprenda a fazer um CGI ou consiga entender expressões regulares. Quero que você também evolua. Quero que você faça programas de verdade em vez de meros scripts toscos.

Prepare-se para uma imersão em shell. Ao ler cada capítulo, você fará um mergulho profundo no assunto, aprendendo bem os fundamentos para depois poder aplicá-los com a segurança de quem sabe o que está fazendo. Os códigos não são entregues prontos em uma bandeja. Passo a passo vamos juntos construindo os pedaços dos programas, analisando seu funcionamento, detectando fraquezas e melhorando até chegar a uma versão estável. Você não ficará perdido, pois avançaremos conforme os conceitos são aprendidos.

Imagine-se lendo o manual de seu primeiro carro zero quilômetro, sentado confortavelmente no banco do motorista, sentindo aquele cheirinho bom de carro novo. Você não tem pressa, pode ficar a noite toda saboreando aquelas páginas, testando todos os botões daquele painel reluzente. Se o manual diz que a luz acenderá ao apertar o botão vermelho, o que você faz imediatamente? Aperta o botão e sorri quando a luz acende.

De maneira similar, estude este livro com muita calma e atenção aos detalhes. Não salte parágrafos, não leia quando estiver com pressa.

Reserve um tempo de qualidade para seus estudos. Com um computador ao alcance das mãos, teste os conceitos durante a leitura. Digite os comandos, faça variações, experimente! Brinque na linha de comando, aprendendo de maneira prazerosa.

Vamos?

	Visite o site do livro (`www.shellscript.com.br`) para baixar os códigos-fonte de todos

os programas que estudaremos a seguir.

Capítulo 1

Programas sim, scripts não

Este livro ensina a fazer programas e não scripts. Seu objetivo é transformar "scripteiros" em programadores, dando o embasamento necessário e ensinando as boas práticas da programação. Isso melhorará a qualidade do código, facilitando muito o trabalho de manutenção futura. Chega de reescrever scripts porque o código original estava incompreensível, chega de dores de cabeça pela falta de limpeza e organização. Aprenda a fazer programas em shell, do jeito certo.

Em geral, qualquer código feito em shell é logo chamado de script, não importando seu tamanho, sua qualidade ou sua função. Mas há uma grande diferença entre um script e um programa.

O que é um script?

A palavra script também é usada em português para referenciar o roteiro de uma peça de teatro ou produção televisiva. Esse script é o caminho que os atores devem seguir, a descrição de todo o show do início até o fim.

Em programação shell, um script não é muito diferente disso. É uma lista de comandos para serem executados em sequência (um comando após o outro). Um roteiro predefinido de comandos e parâmetros. Geralmente o que está no script é aquilo que o usuário digitaria na linha de comando. Realmente, apenas colocando-se todo o histórico da linha de comando em um arquivo, tem-se um script!

Uma curiosidade do script é que as pessoas parecem não levá-lo muito a sério. Fazem códigos feios, sem comentários, sem alinhamento, sem muita clareza. Parece que scripts são sempre feitos "nas coxas" e por quem não têm muita noção de programação. No fim, este acaba sendo um grande trunfo da programação em shell: qualquer um pode fazer scripts! Basta saber usar a linha de comando e pronto, nasceu mais um scripteiro. Por isso, o shell é tão difundido e popular para automatização.

Como o shell é poderoso e gostoso de usar, com o tempo mais e mais tarefas começam a ser desempenhadas por scripts e estes começam a crescer. Logo começam a ficar lentos, complicados, difíceis de manter. Aquela brincadeira de programar de maneira desleixada, agora tornou-se uma maçaroca sem início nem fim.

É normal também escrever scripts rápidos, de poucas linhas, que são usados por alguns dias e depois simplesmente esquecidos. Como é fácil perder scripts! Eles são descartáveis. Mas tudo bem, isso não é um problema desde que se tenha em mente que o script recém-escrito não era para ser sério. O problema é quando um script desses é o responsável por uma tarefa importante, vital para o funcionamento de uma empresa, por exemplo.

Os scripts são ótimos para automatizar tarefas repetitivas e um lugar propício de elas acontecerem é no servidor. Os administradores de sistema sempre acabam recorrendo aos scripts para não precisarem fazer manualmente aquelas tarefas chatas, rotineiras. Só que administradores em geral não são programadores, então o resultado é um emaranhado de pequeninos scripts de "5 minutos" que mantêm serviços importantes funcionando, monitoram conexões, fazem becape... Já imaginou que seus dados importantes podem estar passando por um destes scripts?

O que é um programa?

Há várias definições que podem ser usadas para descrever um programa. Em nosso contexto, um programa é um script feito do jeito certo.

Programas não são feitos nas coxas. Eles são pensados, analisados, codificados com cautela, têm comentários, cabeçalho, tratam erros e exceções, são alinhados, bonitos de se ver. Um programa não é uma maçaroca, não é enigmático e tampouco descartável. Um programa é feito para ser funcional, eficiente e principalmente: é feito para ser atualizado.

Um programa não é uma obra de arte imutável. Pelo contrário! Um programa é vivo, mutante, imperfeito, incompleto, que está sempre melhorando, ficando menor/maior, mais rápido/lento. É raro encontrar programas "versão final", em que não há mais o que melhorar. Além da simples automatização de tarefas, programas são feitos para resolver problemas e suprir necessidades. Problemas mudam, necessidades mudam, então programas também mudam!

Um programador não é um scripteiro. Ele reconhece a importância de um código limpo e legível, que facilite o trabalho de manutenção e compreensão de seu funcionamento. De que adianta fazer um código complicadíssimo, rápido e inovador se na hora de atualizá-lo ninguém souber como alterá-lo e não restar outra solução senão o famoso reescrever do zero?

Principais diferenças entre scripts e programas

Script	Programa
Codificação descuidada	Codificação cautelosa

Código feio e sem estrutura	Código limpo
Pouca manutenção, descartável	Vivo, evolução constante
Feito por um usuário	Feito por um ou mais programadores
Bugs são ignorados	Bugs são encontrados e corrigidos

Por que programar em shell?

O primeiro passo de quem aprende shell é fazer scripts. Administradores de sistemas precisam fazer scripts para automatizar tarefas do servidor, usuários fazem scripts para aprender a programar ou para criar pequenas ferramentas de auxílio. Fazer scripts é fácil e tranquilo, com poucas linhas é possível desfrutar das vantagens que a automatização e a padronização nos trazem.

Já programas são um passo maior. É preciso tempo de dedicação e estudo para resolver problemas e codificar soluções. Mas, se existem várias linguagens de programação mais poderosas e flexíveis que o shell, então por que usá-lo para fazer programas? Simples: porque o tempo de aprendizado é reduzido.

Um ponto-chave é o conhecimento prévio em shell script. Se a pessoa investiu seu tempo em aprender os comandos e estruturas do shell, já sabe fazer seus scripts e sente-se à vontade com a linguagem, por que não dar um passo maior e fazer programas completos? Não se sai do zero, e sim se aperfeiçoa um conhecimento que já possui. Com isso, o caminho a percorrer é menor e em pouco tempo o antigo scripteiro poderá estar fazendo programas com P maiúsculo.

Programadores de outras linguagens que são novatos no shell também se beneficiarão do aprendizado rápido. A parte mais difícil que são os algoritmos e os mecanismos de funcionamento de um programa já são conhecidos. Resta apenas aprender a sintaxe e as características do shell, que são facilitados por serem aprendidos diretamente na linha de comando do sistema operacional.

Programar em shell é diferente!

Para quem conhece linguagens de programação tradicionais como C,

Pascal e Cobol, ou as mais moderninhas como Java, PHP, Python e Ruby, logo notará que programar em shell é diferente.

A programação é mais tranquila, de alto nível. Não é preciso se preocupar com o tipo das variáveis, acesso ao hardware, controle de memória, ponteiros, compilação, plataforma, módulos, bibliotecas, bits, bytes, little/big endian e outros complicadores. Para o programador, resta a parte boa: algoritmos. Ele fica livre para criar soluções e deixa de se preocupar com limitações da máquina ou da linguagem.

Programar em shell geralmente envolve a manipulação de texto e o gerenciamento de processos e de arquivos. As tarefas complexas ficam com as ferramentas do sistema como `grep`, `sed`, `dd` e `find` que se encarregam dos bits e bytes e possuem interface amigável via opções de linha de comando.

Além de possuir as funcionalidades básicas de uma linguagem estruturada normal e a integração natural com o sistema operacional e suas ferramentas, há as facilidades de redirecionamento, em que é possível combinar vários programas entre si, multiplicando seus poderes e evitando reescrita de código. Pelo uso intensivo dessas ferramentas e da possibilidade de interoperabilidade entre elas, a programação em shell é chamada do tipo LEGO, onde a maioria das peças necessárias já existe, bastando saber como combiná-las para produzir soluções.

É a filosofia do Unix mesclando-se com a arte da programação.

Capítulo 2

Controle de qualidade

Um bom profissional sabe como fazer um trabalho de qualidade. Isso faz parte do seu dia a dia, não é preciso se esforçar. A atenção aos detalhes e a busca por um resultado perfeito são algumas de suas características. Aprenda quais são os componentes que farão seu programa em shell ser considerado profissional. Destaque-se, faça o melhor, faça o todo.

Por estar sempre evoluindo e precisar de manutenção constante, um programa tem um estilo de codificação diferente de um script. Um ou mais programadores podem estar envolvidos na manutenção de um mesmo programa, então quanto mais limpo e bem-escrito o código, mais fácil será o trabalho de mantê-lo funcionando.

Como regra geral, o programador deve ter em mente que o código que ele está escrevendo pode precisar ser alterado dali a alguns dias ou meses. Passado esse tempo, ele não vai mais ter todo o esquema de funcionamento em sua cabeça. Ou ainda, pode ser que outro programador precise alterá-lo, então quanto mais fácil e informativo o código, menores são as chances de receber uma ligação desesperada durante a madrugada :)

Existem várias maneiras de se deixar o código de um programa mais amigável e informativo. Algumas delas, entre outras que veremos adiante:

- Cabeçalho inicial com uma explicação geral do funcionamento.
- Código alinhado (*indent*) e bem-espaçado verticalmente.
- Comentários explicativos e esclarecedores.
- Nomes descritivos para funções e variáveis.
- Controle de alterações e versões.
- Estrutura interna coesa e padronizada.

Além disso, um programa ainda pode ter extras para torná-lo mais profissional:

- Manual de uso.
- Manual técnico.
- Suíte de testes.
- Registro histórico das mudanças (Changelog).
- Base de problemas conhecidos.
- Versões beta.
- Arquivos de exemplo de comportamento (se aplicável).

Veremos agora em detalhes os componentes de um programa de qualidade, o que ele deve ter para ser visto como um trabalho profissional.

Cabeçalho inicial

O cabeçalho inicial é a primeira parte do programa, o primeiro texto que qualquer um que for ler seu código-fonte irá encontrar. Pelo seu posicionamento, é aconselhável colocar neste cabeçalho todas as informações necessárias para que o leitor tenha uma visão geral do programa.

O cabeçalho é composto por várias linhas de comentário. Um comentário nada mais é do que uma frase normal em português (ou inglês, dependendo do caso), precedida de um caractere "#".

```
# Isto é um comentário
echo Este é um comando
```

Minuto-curiosidades do mundo nerd: Qual o nome do caractere "#"? Gradinha, grelha, cerquilha, sustenido, hash, jogo da velha, lasanha etc.

Ao escrever um cabeçalho, coloque todas as informações que um programador precisaria ter para poder entender do que se trata o programa, o que ele faz e como faz. Coloque-se no lugar desse programador. Que informações você gostaria de ler antes de se debruçar sobre o código?

Também são importantes informações adicionais como o nome do autor e as datas de criação e última modificação. Como sugestão, este é o formato de um cabeçalho informativo:

- Localização (*path*) do interpretador.
- Nome do programa.
- Descrição breve do propósito.
- Site do programa (se aplicável).
- Nome do autor.
- E-mail do autor.

- Nome e e-mail do responsável pela manutenção (caso diferente do autor).
- Descrição detalhada do funcionamento.
- Data de criação.
- Histórico de mudanças (data, nome, descrição).
- Licença (copyright).

Veja como todas estas informações são colocadas em um cabeçalho:

⌨ Exemplo de cabeçalho completo

```
#!/bin/bash
#
# nome_completo.sh - Busca o nome completo de um usuário no Unix
#
# Site      : http://programas.com.br/nomecompleto/
# Autor     : João Silva <joao@email.com.br>
# Manutenção: Maria Teixeira <maria@email.com.br>
#
#  ------------------------------------------------------------
#   Este programa recebe como parâmetro o login de um usuário e
#   procura em várias bases qual o seu nome completo, retornando
#   o resultado na saída padrão (STDOUT).
#
#   Exemplos:
#      $ ./nome_completo.sh jose
#      José Moreira
#      $ ./nome_completo.sh joseee
#      $
#
#   A ordem de procura do nome completo é sequencial:
#
#      1. Arquivo /etc/passwd
#      2. Arquivo $HOME/.plan
#      3. Base de Usuários LDAP
#      4. Base de Usuários MySQL
#
#   Respeitando a ordem, o primeiro resultado encontrado será o
```

```
#   retornado.
#  ----------------------------------------------------------------
#
#
# Histórico:
#
#    v1.0 1999-05-18, João Silva:
#       - Versão inicial procurando no /etc/passwd
#    v1.1 1999-08-02, João Silva:
#       - Adicionada pesquisa no $HOME/.plan
#       - Corrigido bug com nomes acentuados
#    v2.0 2000-04-28, Mário Rocha:
#       - Corrigidos 2.534 bugs (O João não sabe programar!)
#       - Adicionado meu nome em vários lugares hehehe
#    v2.1 2000-04-30, José Moreira:
#       - Desfeitas as "correções" do Mário (ele quebrou o
  programa)
#       - Retirada a frase "Mário é o Maior" de várias partes
#    v2.2 2000-05-02, José Moreira:
#       - Adicionado suporte a LDAP (que trabalheira!)
#       - Aceita nomes de usuários EM MAIÚSCULAS
#       - Retirado suporte a acentuação (estraga meu terminal)
#    v2.3 2000-05-03, José Moreira:
#       - Arrumado o suporte a LDAP (agora vai funcionar)
#    v2.4 2000-05-03, José Moreira:
#       - Arrumado o suporte a LDAP (agora é pra valer)
#    v2.5 2000-05-04, José Moreira:
#       - Retirado o suporte a LDAP (eu odeio LDAP!)
#    v3.0 2000-05-10, Maria Teixeira:
#       - Programa reescrito do zero
#       - Adicionado suporte a LDAP (funcional)
#       - Adicionado suporte a MySQL
#       - Restaurado suporte a acentuação
#    v3.1 2003-05-10, Maria Teixeira:
#       - Adicionado este comentário para comemorar 3 anos sem
#         alterações :)
#
#
# Licença: GPL.
```

```
#
```

Ficou extenso? Sim. Isto é ruim? Não.

Pode parecer feio um cabeçalho grande. De fato, às vezes o cabeçalho pode ficar até maior que o programa... Mas isso não é um problema, pois, para o interpretador, os comentários são ignorados, não influindo no desempenho. Já para o programador que precisar modificar o código, quanta diferença! E cuidado, este programador pode ser você mesmo...

Antes de mergulhar no código, é importante ter uma visão geral do programa, o que facilita muito a sua análise. Não acredita? Então que tal descrever o que faz este programa?

Programa misterioso

```
#!/bin/bash
o=+ a=1 z=${1:-1}; [ "$2" ] && { a=$1; z=$2; } ; [ $a -gt $z ] &&
  o=-
while [ $a -ne $z ]; do echo $a ; eval "a=\$((a $o 1))"; done;
  echo $a
```

Complicado, não? Mesmo executando o programa, ainda assim é difícil dizer precisamente o que ele faz, em todos os detalhes. Que falta faz um cabeçalho! Nem nome o programa tem, tudo o que se sabe é que ele roda no Bash, por causa da primeira linha.

Analise a diferença. Como está agora:

- O **autor** escreveu o código e não se importou em escrever um cabeçalho. O programa é dele, funciona para ele.
- O **programador** que "cair de paraquedas" no código e precisar modificá-lo, vai gastar vários minutos (talvez horas?) só para entender o que ele faz e para que ele serve.
- O **usuário** que encontrar seu programa, vai simplesmente ignorá-lo, pois não saberá para que serve e nem como utilizá-lo.

Por outro lado, a situação poderia ser diferente:

- O **autor** escreveu o código e investiu mais 10 minutos para escrever um cabeçalho descritivo, enquanto toda a ideia está fresca em sua cabeça.

- O **programador** que "cair de paraquedas" no código e precisar modificá-lo, vai gastar um minuto para ler o cabeçalho e imediatamente irá saber o que o programa faz e para que serve.
- O **usuário** que encontrar seu programa vai ler o cabeçalho e saber o que ele faz, podendo utilizá-lo tranquilamente.

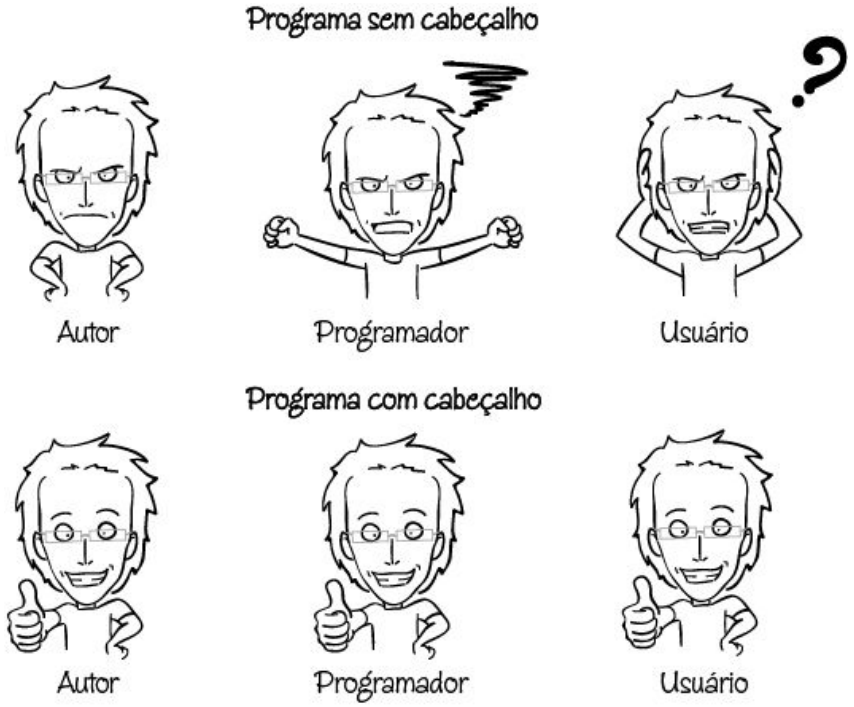
Ou seja, não é trabalho algum para o autor escrever um texto simples, com exemplos. Já para quem precisar olhar o código, ter o cabeçalho pode ser a diferença entre uma apresentação imediata ao programa ou tempo perdido em testes e análises apenas para saber do que se trata.

Que tal inchar um programa de três linhas para 23 e torná-lo mais amigável?

Programa misterioso com cabeçalho

```
#!/bin/bash
#
# seq.sh - Emulação do comando seq em Bash, por Aurelio M. Jargas
#
# O comando seq recebe dois números e mostra na saída padrão todos
  os números
# existentes entre eles, inclusive. Essa sequência está pronta
  para ser usada
# pelo comando FOR. Caso omitido o número inicial, é utilizado o
  1.
#
#       Exemplo 1           Exemplo 2           Exemplo 3
#
#        $ seq 5 10         $ seq 10 5          $ seq 5
#        5                  10                  1
#        6                  9                   2
#        7                  8                   3
#        8                  7                   4
#        9                  6                   5
#        10                 5
#
# Uso em scripts: for i in $(seq 5 10); do comandos ; done
#
o=+ a=1 z=${1:-1}; [ "$2" ] && { a=$1; z=$2; } ; [ $a -gt $z ] &&
```

```
  o=-
while [ $a -ne $z ]; do echo $a ; eval "a=\$((a $o 1))"; done;
  echo $a
```



Aaaaaaah, agora sim... Uma lida rápida no cabeçalho já nos dá todas as informações necessárias para entender o que ele faz. E ainda, com os exemplos, também temos instruções de como utilizá-lo. Então, vale ou não vale a pena aumentar o tamanho do programa para facilitar sua manutenção? Claro que vale!

Código limpo

O programa seq.sh visto no tópico anterior já possui um cabeçalho bem descritivo do que ele faz. Mas logo após temos duas linhas que poderiam ser definidas como "sopa de letrinhas", pois vários comandos foram ajuntados em uma massa de caracteres difícil de se digerir.

Para que um programa seja legível por humanos, o seu código deve ser limpo, ou seja, deve estar visualmente bem organizado para facilitar a leitura. Para o computador não faz diferença, mas para nós, quanto mais fácil encontrar a informação que se procura, melhor.

Existem várias regras simples para melhorar o código e torná-lo legível,

bonito até. Note que estas regras dizem respeito a posicionamento do código apenas, nenhum comando ou nome de variável precisa ser alterado. As regras de um código visualmente limpo, incluem:

- Colocar apenas um comando por linha.
- Alinhar verticalmente comandos de um mesmo bloco, usando espaços ou TABs.
- Deslocar o alinhamento à direita a cada bloco novo.
- Usar linhas em branco para separar trechos.
- Não ultrapassar o limite de 80 colunas por linha.

Todas estas regras são facilmente aplicadas apenas inserindo-se espaços e quebras de linha. Coisa rápida, que ao acostumar-se é feita automaticamente, sem pensar, enquanto se digita o código. É igual trocar a marcha do carro, fica natural. Vamos ver a transformação pela qual o código sopa de letrinhas vai passar:

seq.sh sujo

```
o=+ a=1 z=${1:-1}; [ "$2" ] && { a=$1; z=$2; } ; [ $a -gt $z ] &&
  o=-
while [ $a -ne $z ]; do echo $a ; eval "a=\$((a $o 1))"; done;
  echo $a
```

seq.sh limpo

```
o=+
a=1
z=${1:-1}
[ "$2" ] && {
    a=$1
    z=$2
}
[ $a -gt $z ] && o=-
while [ $a -ne $z ]; do
    echo $a
    eval "a=\$((a $o 1))"
done
```

```
echo $a
```

Que diferença, hein? O programa é exatamente o mesmo, porém agora está mais claro de ler e tentar entender. Basicamente os separadores ponto e vírgula foram trocados por quebras de linha, os comandos dos blocos `{...}` e `while` foram alinhados e deslocados por um TAB e várias linhas em branco foram adicionadas para separar o código em partes distintas. Não há segredo algum em limpar o código, basta fazer!

Para o deslocamento, podem-se usar espaços em branco ou TABs, tanto faz. Fica a critério do programador escolher. Os TABs são mais flexíveis porque os editores de texto bons têm opções para trocar o tamanho de um TAB, podendo ser oito espaços (o padrão) ou qualquer outra quantidade que se queira. Outro motivo é que um programa alinhado com TABs fica menor (em bytes) do que um alinhado com espaços. Se você está em dúvida, use TAB.

As linhas em branco, além de dar um respiro na leitura, ajudam a separar os pedaços do programa. Se você olha para um trecho do programa e pode dizer "este trecho faz tal coisa", então ele é um bloco lógico que merece estar separado dos demais por uma linha em branco. Há blocos bem claros que aparecem na maioria dos programas: definição das variáveis globais, declaração das funções, processamento, formatação, saída. Não os deixe grudados!

Para comandos muito extensos que podem ultrapassar 80 colunas, basta "escapar" a quebra de linha colocando uma barra invertida "\" antes e continuar o comando na linha seguinte. O dialog é um comando que usa linhas bem compridas.

```
dialog --title "Minha telinha" --msgbox "Esta é a minha mensagem"
  0 0
```

Para facilitar a leitura do comando, é melhor quebrá-lo em várias linhas. Com uma opção por linha, é bom alinhá-las com espaços à esquerda, ficando muito clara a organização geral:

```
dialog    \
        --title "Minha telinha" \
        --msgbox "Esta é a minha mensagem" \
        0 0
```

Código comentado

Já foi visto por que é importante o programa ter um cabeçalho completo. Já foi visto como limpar o código, tornando-o legível. Mas isto ainda não é o suficiente para que o programa seja fácil de se alterar. Mesmo estando o `seq.sh` já limpo e com cabeçalho, ainda podem restar perguntas como:

- O que faz exatamente essa linha?
- Qual a função desse trecho do código?
- Por que foi usado o comando X em vez do Y?
- Por que é feito esse teste aqui?

Para o autor do programa, as respostas são óbvias, mas o mesmo não acontece com o resto do mundo, que não codificou aquele programa do zero.

Código é código. O computador sempre o executará corretamente estando limpo ou não. Mas, para os humanos, nenhum código substituirá o bom e velho português, que é a língua que dominamos. Então não seria bom misturar textos explicativos em português no meio do código?

É para isto que existem os comentários, eles são uma maneira do programador deixar recados no código, explicar o que faz uma linha ou um bloco e até deixar lembretes para ele mesmo ler depois. Existem três tipos de comentários em shell:

1. Comentário em bloco, para introduzir uma seção:

```
############################################################
#                                                          #
#  Este é um comentário em bloco.                          #
#                                                          #
#  Visualmente distinto dos comandos, serve para colocar   #
#  textos extensos, introduções e exemplos.                #
#                                                          #
############################################################
```

2. Comentários de uma linha para comandos:

```
# Converte o e-mail para maiúsculas
echo "$email" | tr a-z A-Z
```

3. Comentários de meia linha para comandos:

```
echo "$email"     # Mostra o e-mail na tela
```

O shell simplesmente ignora os comentários, então não há problema algum em utilizá-los. Eles podem ser resumidos para ocupar meia linha, ou podem ser tão extensos quanto necessário para explicar um trecho complexo. O bom programador, além de escrever códigos inteligentes e eficientes, deve saber colocar comentários relevantes e claros para completar a obra.

Alguns programadores gostam de dizer que não precisam comentar seus códigos, pois eles são claros, basta ler os comandos. Esses programadores tiveram a sorte de nunca precisar alterar um código feito por outra pessoa, sem comentários. É claro que todo programador sabe ler código, mas **entender** exatamente o que faz cada pedaço do programa é algo que necessita de tempo, testes e análise. Isso é um custo alto, e dependendo da complexidade do programa, pode ser um impeditivo em seu uso.

Por outro lado, um programa bem-comentado ensina ao programador como utilizá-lo, ele ajuda em vez de complicar, é um amigo em vez de um desafio. Para exemplificar bem a importância de um código bem-comentado, veja como agora fica fácil entender o que faz o `seq.sh`:

seq.sh limpo e comentado

```
#!/bin/bash
#
# seq.sh - Emulação do comando seq em Bash, por Aurelio M. Jargas
#
# O comando seq recebe dois números e mostra na saída padrão todos
  os números
# existentes entre eles, inclusive. Essa sequência está pronta
  para ser usada
# pelo comando FOR. Caso omitido o número inicial, é utilizado o
  1.
#
#       Exemplo 1           Exemplo 2           Exemplo 3
#
#       $ seq 5 10          $ seq 10 5          $ seq 5
#       5                   10                  1
```

```
#        6                   9                   2
#        7                   8                   3
#        8                   7                   4
#        9                   6                   5
#        10                  5
#
# Uso em scripts: for i in $(seq 5 10); do comandos ; done
#
### Inicialização das variáveis
o=+         # Operação a ser feita. Pode ser + ou -
a=1         # Valor padrão de início da contagem
z=${1:-1}   # Valor do final da contagem recebido em $1 (padrão é
 1)
# A variável 'a' é o número de início e a variável 'z' é o final.
#
# O código anterior serve para quando o usuário passar apenas um
 valor na
# linha de comando, que seria o valor do *final* da contagem. Por
 isso
# a=1 e z=$1.
#
# Caso o programa seja chamado sem parâmetros, 'z' também é
 definido
# como 1. Como a=1 e z=1, o programa retorna apenas o número 1.
[ "$2" ] && {

    # Foram passados 2 parâmetros, o $1 é o início e o $2 é o fim.
    a=$1
    z=$2
}
###
# Se o número inicial ($a) for maior que o número final ($z),
# faremos uma contagem regressiva, por isso a operação é definida
# como subtração.
#
[ $a -gt $z ] && o=-
###
# Loop da contagem (progressiva ou regressiva)
#
# A cada volta, adiciona ou subtrai 1 do valor inicial,
```

```
# até que ele se iguale ao final.
#
# O eval executa o resultado da expansão de variáveis.
# Supondo o='+' e a=3, o eval executará o comando a=$((3+1)).
#
while [ $a -ne $z ]; do
    echo $a                # mostra a contagem atual
    eval "a=\$((a $o 1))"  # efetua a operação (+1 ou -1)
done

echo $a                    # mostra o último número
```

Uau! O antigo programa misterioso sopa de letrinhas, que tinha apenas três linhas, agora é um programão de mais de 60 linhas, e está perfeitamente legível. Note que nenhum comando foi alterado, apenas brancos e comentários foram adicionados. Vamos até relembrá-lo para ver a diferença. E que diferença!

```
#!/bin/bash
o=+ a=1 z=${1:-1}; [ "$2" ] && { a=$1; z=$2; } ; [ $a -gt $z ] &&
  o=-
while [ $a -ne $z ]; do echo $a ; eval "a=\$((a $o 1))"; done;
  echo $a
```

Não tem comparação. Códigos compactos são interessantes, intrigantes. Mas na vida real em que não se pode perder tempo analisando maçarocas, o código limpo e comentado é uma necessidade.

Se você não tem o costume de comentar seus códigos ou acha que isto é perda de tempo, espero que o exemplo do `seq.sh` o tenha iluminado. Libere seus neurônios de ter que lembrar de todo o funcionamento do programa. Em vez disso, escreva comentários enquanto o programa ainda está fresco na memória e depois esqueça! Quando precisar mexer no programa novamente, basta ler os comentários. Satisfação garantida.

TODO, FIXME e XXX

Ainda falando sobre comentários, há três tipos especiais que são largamente utilizados em todas as linguagens de programação: `TODO`, `FIXME` e `XXX`. São comentários normais, porém com um prefixo padrão que os identifica como um recado especial. Até os editores de texto que colorem os códigos dão um destaque extra a estes comentários.

Eles servem para a comunicação entre programadores ou mesmo como lembretes feitos pelo autor para ele mesmo ler depois. Cada tipo tem uma finalidade específica, facilitando a categorização de comentários. Acostume-se a usá-los e rapidamente você perceberá sua utilidade como lembretes e como organização de pendências.

Prefixos especiais para comentários

TODO	Indica uma tarefa a ser feita, uma pendência ainda não resolvida. `#TODO Colocar este trecho em uma função`
FIXME	Indica um bug conhecido, que precisa ser arrumado. `#FIXME Este loop não está funcionando para números negativos`
XXX	Chama a atenção, é um recado ou uma notícia importante. `#XXX Não sei como melhorar este trecho, me ajudem!!!`

Como comentar bem

Nenhum programador pode se considerar um bom profissional se não dominar a arte de fazer comentários. Mas embora seja fácil colocar vários "#" no código e sair escrevendo, fazer um comentário **bom** não é tão fácil assim. Acompanhe as dicas seguintes e pratique em seus próprios programas:

- **Escrevo em português ou inglês?**

Como regra geral, sempre comente em português, que é a língua falada. É a língua que você domina, é a língua que os outros programadores dominam. Escrever em inglês só vai trazer uma barreira adicional para quem precisar ler seu código. Programar já tem desafios de sobra, não é preciso um empecilho extra nos comentários. Não é "bonito" nem "chique" escrever em inglês, se você faz isso sem motivo algum, está errado.

Comente em inglês se você tem a intenção de lançar seu programa internacionalmente como um software de código aberto, para poder receber contribuições de programadores de qualquer parte do mundo. Ou, ainda, quando precisar trabalhar em uma equipe com programadores de várias nacionalidades ou quando o código em inglês for um requisito de projeto, o que é muito comum em empresas multinacionais.

■ **Uso acentuação?**

Sim! As letras acentuadas fazem parte do português e a sua falta dificulta a leitura. Os comentários existem para facilitar o trabalho e não para criar novos problemas. Não há motivo técnico para não acentuar, pois o shell simplesmente ignora os comentários durante a execução do programa, eles são inofensivos.

Se o seu sistema está mal configurado ou você tem preguiça de usar os acentos, isto é um problema pessoal que você, como bom profissional, deve resolver.

Escrever sem usar acentuacao passa uma imagem de servico porco e amador, entao salvo situacoes especificas onde limitacoes do ambiente proibem os acentos, acentue sempre!

■ **Como deve ser o texto de um comentário?**

Curto, direto ao ponto, claro, relevante, com contexto, sem erros de ortografia e gramática. Diga tudo o que tiver que dizer, com o menor número possível de palavras. Seja informativo, porém sucinto. Dê respostas em vez de complicar, antecipando possíveis dúvidas do leitor. E principalmente: mantenha atualizado. Melhor nenhum comentário do que um comentário errado.

■ **Faz mal comentar demais?**

Sim. Se você colocar muitos comentários desnecessários, o código ficará poluído, prejudicando sua leitura. Tente encontrar o equilíbrio entre a falta e o excesso. Comente trechos que possam gerar dúvidas, mas não coloque comentários irrelevantes, óbvios para qualquer programador:

```
a="abc"       # Define 'a' com 'abc'

i=$((i + 1))  # Incrementa o contador i

ApagaArquivosTemporarios()  # Função que apaga arquivos
temporários
```

Também não faça dos comentários um canal de expressão de sua angústia ou um local para autopromoção. Os comentários devem falar somente sobre o código, e não sobre o autor dele.

■ **Posso fazer piadinhas? Posso xingar?**

Piadinhas, pode, mas com muita moderação. Uma tirada inteligente ou uma citação engraçada vão fazer o leitor descontrair-se por alguns segundos, mas ao mesmo tempo podem tirar a sua concentração no código. Use com cautela. Se estiver em dúvida, não faça.

Xingar não pode. Nunca. Assim como você não coloca palavrões nas provas do vestibular, nas redações do colégio e no seu currículo, não estrague a qualidade do seu trabalho para expressar uma raiva temporária. É normal perder a paciência com algum bug difícil ou ao ver um código porco. Mas resista à tentação de praguejar, respire fundo, fique zen.

- **"Don't comment bad code, rewrite it."**

Como mensagem final, um conselho famoso que nunca deve ser esquecido. Não perca tempo escrevendo páginas de comentários para tentar explicar um trecho de código obscuro e malfeito. É melhor reescrever o código de maneira clara.

Variáveis padronizadas

O uso racional de variáveis influi diretamente na legibilidade do código. Variáveis são simples, mas a falta de padronização em seu uso pode prejudicar o entendimento dos algoritmos.

Como nomear variáveis

- Dê um nome descritivo para cada variável.
- Use apenas letras, números e o sublinhado "_".
- Prefira MAIÚSCULAS para variáveis globais e minúsculas para variáveis locais e temporárias.
- Decida usar somente português ou somente inglês para nomear as variáveis, não misture!
- Acentuação não é bem-vinda no nome de uma variável.
- `$var1`, `$var2` e `$var3` são definitivamente péssimos nomes para variáveis.
- `$a`, `$m` e `$x` também não ajudam.

Quando criar uma variável

- Quando precisar usar o mesmo texto ou número duas ou mais vezes.
- Quando seu programa possuir dados que podem ser modificados pelo usuário.
- Quando perceber que um texto um dia poderá ser alterado.
- Quando precisar de configuração de funcionalidades (opções).
- Quando precisar ligar/desligar funcionalidades (chaves).
- Quando quiser, variáveis são legais e dificilmente atrapalham.

Detalhes sobre variáveis no shell

- Todas são globais, a não ser quando precedidas pelo comando `local`.
- Não são tipadas, então tanto faz `a=5`, `a="5"` ou `a='5'`.
- As 'aspas simples' impedem a expansão da variável.
- Podem ser colocadas dentro das aspas juntamente com strings, "assim `$PWD`".
- Sempre use "aspas" ao redor de variáveis para manter os espaços em branco e evitar problemas com variáveis vazias.
- Não é erro referenciar uma variável não existente, como `$homersimpson`.
- Coloque todas as variáveis importantes no início do programa, logo após o cabeçalho.
- Também podem ser acessadas como `${nome}` para evitar confusão com nomes grudados, como em `$PWDjoao` ou para acessar mais do que nove parâmetros: `${10}`, `${11}`, ...

Funções funcionais

Como nomear funções

- Dê um nome descritivo para cada função.

- Use apenas letras, números e sublinhado "_".
- Deixe claro quais são as palavras que compõem o nome de uma função, utilizando iniciais em maiúsculas ou separando-as com o sublinhado. Exemplos: `PegaValorNumerico`, `ExtraiNomeUsuario`, `pega_valor_numerico`, `extrai_nome_usuario`.
- Decida usar somente português ou inglês para nomear as funções, não misture!
- Acentuação não é bem-vinda no nome de uma função.
- `func1`, `func2` e `func3` são definitivamente péssimos nomes para funções.
- `a`, `m` e `x` também são um horror.

Quando criar uma função

- Quando precisar usar o mesmo trecho de código duas ou mais vezes.
- Quando quiser pôr em destaque um trecho de código especialista.
- Quando precisar de recursividade.
- Quando quiser, funções sempre ajudam.

Detalhes sobre funções no shell

- Sempre use o comando `local` para proteger todas as variáveis de uma função.
- Funções são como comandos do sistema, podendo ser chamadas diretamente pelo nome.
- A função `cat` tem preferência ao comando `cat` do sistema.
- Sempre coloque todas as funções no início do programa, logo após as variáveis globais.
- Uma função deve ser declarada antes de ser chamada, a ordem importa.
- Funções só podem retornar números de 0 a 255, usando o comando `return`.
- Funções podem retornar strings usando o comando `echo`.

■ Funções podem gravar variáveis globais, mas evite fazer isso.

Versionamento

Como o código de um programa está em constante manutenção, é preciso uma maneira fácil de identificar seu estado ao longo do tempo. Referenciar versões antigas como "código da semana passada" ou "código quando funcionava a detecção de rede" não é muito profissional, concorda?

Por isto programas possuem versões numéricas e sequenciais: versão 1.0, versão 5.2.1, versão 2007-4. A cada mudança o número aumenta, ficando fácil identificar versões passadas e traçar objetivos para versões futuras.

Embora haja algumas tentativas de padronização, na prática cada programador escolhe a forma de numerar/nomear versões que mais o agrada. Veja alguns exemplos de versionamento atualmente utilizados:

Versão	Descrição
1	Sequencial: 1, 2, 3, 4...
1.0	Sequencial com subversões 1.1, 1.2, 1.3...
1.0.0	Sequencial com subsubversões 1.1.1, 1.1.2, ...
1.0a	Sequencial com números e letras.
1.0-carlos	Sequencial com o nome do autor.
1.0-ALPHA	Sequencial ALPHA (instável, primeira tentativa).
1.0-BETA1	Sequencial BETA (para pré-lançamento e testes).
1.0 (mickey)	Sequencial com codinome (geralmente um nome estranho).
20070131	Data no formato AAAAMMDD.
6.0131	Data em outros formatos (A.MMDD).
1.0-20060131	Sequencial e data.
...	...

E a lista segue, em uma infinidade de combinações e significados. O mais comum é o formato N.N, no qual há a versão principal, como 2.0 e as próximas são subversões como 2.1 e 2.2. Costuma-se chamar de série cada conjunto de versões com uma mesma versão principal, neste caso, seria a "série 2.x". Geralmente neste esquema, as versões ponto-zero

trazem grandes novidades ou são reescritas completas da série anterior, podendo até quebrar compatibilidade.

Como regra geral, o formato N.N serve para a grande maioria dos programas e é fácil de memorizar. Caso o programa seja de atualização constante, como uma por semana ou mais, é aconselhável usar a data como versão para facilitar a identificação do programa. Ou, ainda, pode-se misturar formatos, usando a data quando o programa ainda está em desenvolvimento inicial e depois de estável ele partir para o N.N.

Na verdade, para a maioria dos programas de pequeno e médio portes, não importa muito o formato de versão que você escolha. Qualquer um vai servir. O que importa mesmo é que você tenha disciplina e sempre aumente o número da versão a cada alteração (ou conjunto delas).

Outro detalhe é que é após a versão 1.9 pode vir a 2.0, a 1.10 ou a 1.9a. Isto vai depender se o programador teve tempo de produzir a nova série, ou ainda precisa de mais algumas subversões para respirar.

Um exemplo dessa necessidade de mais tempo para terminar a série nova aconteceu com o editor de textos Vim na versão 6.0, que atrasou. Ele esgotou todo o alfabeto de "a" a "z" e ainda não estava pronto, então teve que usar 6.0aa, 6.0ab, 6.0ac... :)

Algumas dicas adicionais sobre versionamento:

- Escolhida a nomenclatura, procure não mudá-la. Os usuários, distribuidores e programas de atualização automática já estão acostumados com o esquema atual e uma mudança pode quebrar várias rotinas.
- É permitido fazer saltos de versão para uma série nova, como da 2.6 para a 3.0. Isso significa que a possível versão 2.7 teve tantas melhorias ou mudanças internas que mereceu iniciar uma série nova. Só não salte números da mesma série (de 2.6 para 2.9) ou outras séries (de 2.6 para 5.8).
- Se o programa é novo e ainda está em desenvolvimento, não tendo sido lançado oficialmente, comece pela versão 0.1 e vá crescendo: 0.2, 0.3, 0.4, ... Quando ele estiver pronto para o lançamento, use a versão 1.0.
- Versões redondas, que iniciam uma série nova (1.0, 2.0, 3.0, ...), tendem a gerar uma expectativa de apresentarem algo revolucionário ou

grandes mudanças de comportamento. Mas não se sinta obrigado a seguir esta regra. O programa é seu, a política de versões é você quem define.

- **Regra sagrada** que nunca deve ser quebrada: se o programa foi alterado, a versão deve ser aumentada. Não podem haver dois códigos diferentes com a mesma versão.

Histórico de mudanças

Assim como acontece com a versão, o histórico das mudanças que um programa sofreu pode ser escrito em vários formatos, e seu conteúdo pode variar desde descrições extremamente técnicas, aproximando-se de um `diff`, até textos corridos no formato de anúncio para massas. Há dois arquivos com nomes padrão que a maioria dos programadores utiliza: `Changelog` e `NEWS`.

Changelog

Usado para registrar todas as mudanças ocorridas nos arquivos do programa, em linguagem técnica e detalhada. A audiência esperada para este arquivo são outros programadores à procura de informações sobre o código. Caso o programa seja composto por apenas um arquivo, o histórico pode estar embutido no próprio cabeçalho, não precisando de um arquivo separado.

Exemplo:

```
2003-01-31 (josé): $ARQUIVO agora pode ser um link simbólico
2003-02-04 (maria): adicionada opção --help
2003-02-05 (maria): adicionadas opções -h, -V e --version
2003-02-06 (josé): Checagem(): adicionado aviso em $USER == 'root'
2003-02-08 (paulo): Corrigidos bugs #498, #367 e #421
2003-02-10 --- lançada versão 1.3
```

NEWS

Usado para registrar todas as mudanças ocorridas no programa como um todo, desde a última versão. A audiência esperada para este arquivo são

usuários que querem saber um resumo de quais são as novidades. A linguagem deve ser acessível e se preciso, didática.

Exemplo:

```
Novidades da versão 1.3:
-    Novas opções de linha de comando -h e --help, que mostram uma
 tela
     de ajuda e -V e --version, que mostram a versão do programa.
-    Adicionado suporte a arquivos que são links simbólicos.
-    Vários bugs reportados pelos usuários foram corrigidos, como
 o
     problema da acentuação no nome do arquivo e extensão em
 letras
     maiúsculas como .TXT.
```

No `Changelog`, várias informações são importantes como a data da alteração, quem fez, o que fez e por que fez. O lançamento de uma versão é apenas uma marcação cronológica, não influindo nas alterações. Caso a equipe utilize uma ferramenta de gerenciamento de problemas, basta informar o número do ticket.

Já no `NEWS`, tudo gira em torno das versões. As mudanças são agrupadas por versões e não importa quem as fez ou quando fez. Algumas mudanças podem até ser omitidas, pois somente as mudanças perceptíveis pelo usuário são importantes. Em resumo:

Arquivo Changelog	Arquivo NEWS
Lido por programadores	Lido por usuários
Linguagem técnica	Linguagem acessível
Agrupado por data	Agrupado por versão
Quem fez o que, quando	Mudanças desde a última versão
Frases curtas, diretas	Texto normal, com exemplos e dicas

Agradecimentos

Um detalhe facilmente esquecido por programadores, mas vital para que as contribuições de usuários aconteçam, são os agradecimentos.

Quando um usuário contribui, indicando um problema ou até mesmo mandando código pronto para ser aplicado, ele faz por vontade própria,

pois geralmente não há remuneração para contribuições em programas.

Pode ser um colega de trabalho, um amigo ou um usuário do outro lado do mundo que baixou seu programa pela Internet, cada um deles vai ficar feliz ao ver seu nome registrado em algum arquivo do seu programa. Isso o motivará a continuar contribuindo e seu programa continuará melhorando, com mais cabeças pensando nele.

Não precisa ser um anúncio em letras piscantes, mas dê um jeito de agradecer e recompensar o trabalho alheio. Todos ganham.

Capítulo 3

Chaves (flags)

Usar chaves (flags) é uma maneira limpa e organizada de programar. O conceito é bem simples, muito fácil de ser assimilado e colocado em prática. Aprenda a separar a coleta de informações e o processamento, fazendo cada um em um passo distinto. Isso simplificará seus algoritmos, melhorando a legibilidade do código e facilitando o trabalho de manutenção futura.

Uma chave é uma variável de apenas dois estados: está ligada ou desligada. Não há meio termo, não há múltiplos valores. É sim ou não, 0 ou 1, verdadeiro ou falso, preto ou branco, dentro ou fora, ON ou OFF. Uma chave funciona exatamente como um interruptor de luz, onde só há dois estados: a lâmpada está acesa ou apagada, não existindo meio acesa ou meio apagada.

Essa limitação de apenas dois estados possíveis é muito útil em programação, para ligar e desligar atributos e funcionalidades.

- Seu programa pode mostrar o texto de saída normal ou colorido?
- Pode mandar o resultado para a tela ou para um arquivo?
- Pode mostrar o resultado dos cálculos em dólares ou reais?
- Pode funcionar em modo quieto ou em modo verboso?
- Pode funcionar em modo normal ou restrito?

Para todas estas situações, basta você ter uma chave em seu programa, que indicará se a funcionalidade em questão está ligada ou desligada. No miolo do programa esta chave pode mudar de estado, dependendo de testes internos, configurações ou opções do usuário. Quando chegar no trecho de código onde a funcionalidade deve ser executada, o estado da chave é conferido para decidir se prossegue ou cancela a tarefa.

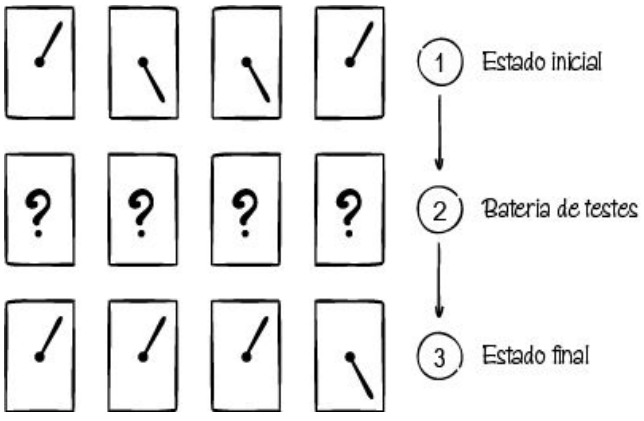


Diagrama de funcionamento das chaves

A grande vantagem desta técnica é separar a coleta de informações da tomada de decisões. Primeiro as informações são obtidas: as chaves são inicializadas e vários testes subsequentes podem alterar o seu estado.

Nenhum processamento "de verdade" é feito neste passo, é a bateria de testes.

Em um segundo passo, os algoritmos do programa tomam decisões analisando os estados dessas chaves. Tais decisões podem ser simples ou complexas, dependendo da própria estrutura e uso das chaves. O bom é que há um universo limitado de possibilidades e combinações, pois cada chave pode ter somente dois estados. Vários IFs, controlando várias condições, conseguem fazer muito de maneira razoavelmente simples e de manutenção facilitada.

Como usar chaves

Vejamos um exemplo de como este esquema de chaves funciona. Há um programa qualquer que mostra uma mensagem na tela. Esta mensagem pode ser colorida ou não, isso vai depender do terminal no qual o programa está sendo executado. Se este terminal tiver a capacidade de mostrar cores, elas serão usadas. Senão a saída é normal, somente texto sem atributos.

Primeiro definiremos uma chave no programa que indicará se as cores podem ser usadas. Se a chave estiver ligada, as cores serão usadas. A chave deve estar logo no início do programa, antes de qualquer processamento. Seu estado inicial deve ser o mais comum de uso, para que só mude em casos de exceção.

```
USAR_CORES=1   # Chave para usar cores (0 desliga, 1 liga)
```

Como a maioria dos terminais atuais sabem como mostrar texto colorido, o estado inicial da chave é LIGADO. Tudo bem, chave criada. Agora lá no meio do código, após feito o processamento inicial, serão realizados os testes que podem afetar o estado da chave. Em nosso caso, os testes devem tentar mudar o estado inicial da chave, desligando-a.

```
# O terminal VT100 não sabe nada de cores
if test "$TERM" = "vt100"
then
    USAR_CORES=0
fi
```

Este bloco detecta um tipo de terminal que não tem suporte a cores (`VT100`), e, quando o encontra, desliga a chave `USAR_CORES`. Perceba que nenhuma atitude é tomada com relação ao estado da chave. O estado mudou, tudo bem. A execução do programa continuará normalmente. Outros testes parecidos podem ser colocados em seguida, todos tentando mudar o estado inicial da chave caso detectem terminais incompatíveis.

Somente lá adiante, passada toda a fase inicial de preparação do programa é que alguma atitude será tomada, dependendo do estado atual da chave. Neste ponto do programa o estado da chave pode inclusive ser o inicial, caso nenhum teste o tenha mudado.

```
if test $USAR_CORES -eq 1
then
    msg_colorida $mensagem   # chama a função "msg_colorida"
else
    echo $mensagem           # usa o echo normal
fi
```

Se após todos os testes a chave permaneceu ligada, a mensagem será colorida. Caso contrário, o texto é mostrado normalmente. O interessante aqui é notar que neste trecho do programa não importa se a chave está em seu estado original ou se foi alterada por algum teste prévio. Tudo o que importa é seu estado atual.

Nessa maneira de programar há uma separação muito clara do que faz cada bloco, sendo independentes uns dos outros. Você pode adicionar vários outros testes, mudar o estado original, não importa. Este trecho final que depende do estado **atual** da chave continuará funcionando da mesma maneira.

Faça chaves robustas

Como você pôde perceber, a chave nada mais é do que uma variável normal. A diferença é que o seu valor tem um significado especial para o programa, podendo mudar o rumo da execução e ativar ou desativar funcionalidades. Dessa maneira, pode-se colocar várias chaves para controlar vários aspectos do programa e apenas definindo valores pode-se alterar todo o comportamento da execução. Um verdadeiro painel de

controle.

Em outras linguagens de programação uma chave é uma variável declarada com o tipo booleano, que somente aceita os valores True ou False. Mas em shell não temos tipagem de variáveis, tudo é string.

Por isso improvisamos, usando 0 e 1 como valores para as chaves. Poderia ser
sim/nao, on/off ou qualquer outro par de palavras que você queira. Mas os números são mais fáceis de alterar, prevenindo erros de digitação.

Outra alternativa seria testar se a variável está vazia ou não. Mas isso pode trazer problemas caso a variável contenha apenas espaços em branco, pode ficar difícil de enxergá-los na depuração de um defeito. E ainda pode dar diferença se você usar ou não as aspas ao redor da variável. Então deixe simples e evite dores de cabeça: use os números.

No exemplo que acabamos de ver, a verificação do estado da chave foi feita usando-se o operador `-eq` do comando `test`. Como este operador é exclusivo para comparações numéricas, caso a chave contenha qualquer outro valor que não um número, seu programa falhará, mostrando erros na tela:

```
$ chave=0
$ test $chave -eq 1 && echo LIGADA
$ chave=1
$ test $chave -eq 1 && echo LIGADA
LIGADA
$ chave=abc
$ test $chave -eq 1 && echo LIGADA
-bash: test: abc: integer expression expected
$ chave=
$ test $chave -eq 1 && echo LIGADA
-bash: test: -eq: unary operator expected
$ test "$chave" -eq 1 && echo LIGADA
-bash: test: : integer expression expected
$
```

Nos dois primeiros exemplos, tudo correu bem. A chave com os valores zero e um comporta-se como esperado: somente o 1 significa ligado. Mas no terceiro exemplo a chave estava com o texto “abc”, quebrando a comparação numérica. No último exemplo a chave estava vazia, também

quebrando a comparação. Perceba no último comando que o uso das aspas ao redor de `$chave` não ajudou.

Antes que você levante o dedo e diga que as chaves não deveriam ter aqueles valores em primeiro lugar, deixe-me lembrar-lhe de um ditado sapientíssimo dito por sei lá quem: *shit happens*. Ou seja, problemas acontecem. São vários os motivos que podem fazer sua chave adquirir algum destes valores incorretos durante a execução, seja por falha do seu programa, falhas de outros programas dos quais ele depende, ou falha do usuário que editou o arquivo.

Independente do culpado, seria interessante que o seu programa não capotasse por causa de uma variável com valor inesperado. Se você puder prever um erro seu ou do usuário, precavendo-se contra ele, seu programa ganha em qualidade! Neste caso, a correção é bem simples, basta fazer uma comparação de strings em vez da numérica:

```
test "$chave" = 1 && echo LIGADA
```

O `-eq` foi trocado pelo sinal de igual, e a chave foi colocada entre aspas, para evitar erro quando estiver vazia. Agora temos uma chave robusta, que liga quando seu valor for 1 e desliga em qualquer outro valor, seja string, numérico ou não definido. Prefira esta forma.

Valor	Estado da chave
`USAR_CORES="abc"`	Chave DESLIGADA
`USAR_CORES=""`	Chave DESLIGADA
`USAR_CORES=`	Chave DESLIGADA
`USAR_CORES=0`	Chave DESLIGADA
`USAR_CORES=2`	Chave DESLIGADA
`USAR_CORES=-1`	Chave DESLIGADA
`USAR_CORES=1`	Chave LIGADA
`USAR_CORES="1"`	Chave LIGADA

Chaves para configuração do usuário

Outra vantagem do uso das chaves é a possibilidade de usá-las como uma maneira rápida de o usuário ter acesso a configurações básicas de seu

programa. Se você está sem tempo de implementar o suporte a arquivo de configuração externo ou opções de linha de comando, as chaves podem ser a sua solução!

Faça chaves com nomes bem descritivos e coloque-as já no início do programa, no primeiro trecho após o cabeçalho. Separe visualmente este bloco de chaves do restante do programa, colocando comentários ao redor. Também descreva em comentários o que faz cada chave, como ligá-las e como desligá-las.

Enfim, deixe tudo bem fácil e amigável para que qualquer pessoa com acesso a um editor de textos possa editar seu programa e mudar o estado inicial das chaves. A separação visual deste bloco do restante do programa evitará que o usuário fique tentado a mexer em outras linhas. E se o fizer, estará por sua própria conta e risco, pois a área de configuração estava bem-sinalizada. É bom pôr um aviso do tipo "Não edite daqui para baixo" com o objetivo de reforçar.

Como basta colocar zeros e uns, o usuário também não precisa ter noções de programação. Ficará intuitivo ao ver o bloco, e os comentários ajudarão a sanar eventuais dúvidas que possam aparecer. Veja um exemplo:

```
############################################################
#
### Configuração do programa mostra_mensagem.sh
### Use 0 (zero) para desligar as opções e 1 (um) para ligar
### O padrão é zero para todas (desligado)
#
USAR_CORES=0      # mostrar cores nas mensagens?
CENTRALIZAR=0     # centralizar a mensagem na tela?
SOAR_BIPE=0       # soar um bipe ao mostrar a mensagem?
CONFIRMAR=0       # pedir confirmação antes de mostrar?
MOSTRA_E_SAI=0    # sair do programa após mostrar?
#
### Fim da configuração - Não edite daqui para baixo
#
############################################################
```

Fácil de entender, não? Claro que fazer o usuário editar seu programa

para configurá-lo não é o ideal, mas esta pode ser uma solução rápida para necessidades básicas.

Ao utilizar este esquema, só cuide para que seus testes não dependam do estado inicial das chaves, pois agora eles podem ser alterados pelo usuário, são imprevisíveis.

Detalhes sobre o uso de chaves

- Seu programa pode ter várias chaves, tantas quantas necessário. Uma lâmpada precisa apenas de um interruptor. Já um avião tem um mundo de chavezinhas e botões para o piloto brincar. Cada caso é um caso.
- O estado inicial de uma chave pode ser mudado por testes internos, estado atual do sistema, horário da execução, resultado de cálculos, configurações do usuário e opções passadas pela linha de comando, entre outros.
- Sempre coloque todas as chaves já no início do programa, com um estado inicial definido. Assim seus testes podem se concentrar em tentar mudar este estado original.
- Resista à tentação de usar outras variáveis do seu programa como chaves, para não misturar os conceitos. Por exemplo, você já tem em seu programa a variável `ARQUIVO_LOG` que serve para guardar a localização do arquivo de log. Você pode assumir que se ela estiver vazia, não fará log. Mas o melhor é criar uma chave chamada `GRAVAR_LOG` e basear esta escolha nela, independente do conteúdo de `ARQUIVO_LOG`.
- Várias chaves podem ser usadas em uma mesma expressão lógica para tomar alguma decisão durante o processamento do programa. Por exemplo, você pode decidir fazer o becape do sistema somente se as chaves `SISTEMA_PARADO` e `BANCO_DADOS_TRAVADO` estiverem ligadas, porém a chave `HORARIO_COMERCIAL` deve estar desligada.
- Algumas chaves podem mudar o estado de outras. Você pode desligar a chave `USAR_CORES` quando a for ligar a chave `GRAVAR_LOG`, pois, ao gravar as mensagens de log em um arquivo texto, as cores são inúteis.

Chaves podem se relacionar!

- Use nomes bem descritivos para suas chaves. Isso tornará seu código mais legível e evitará a necessidade de comentários para explicar o que faz a chave. Quanto mais descritivo e claro o nome da chave, mais rápido de entender para que ela serve. Não tenha medo de usar nomes realmente amigáveis como `GRAVAR_LOG_EM_ARQUIVO` e `ESCONDER_MENSAGENS_DE_ERRO`.

As dicas já vistas em relação aos nomes de variáveis também se aplicam às chaves!

- A língua inglesa tende a ser mais concisa para nomes de chaves, como `SAVE_LOGFILE` e `HIDE_ERROR_MSGS` para os dois exemplos anteriores. Mas aconselha-se que o idioma do nome das chaves siga o padrão das outras variáveis do programa.
- Você também pode usar um prefixo padrão no nome de suas chaves para identificá-las como tal, diferenciando-as das variáveis normais de seu programa. Sugestões são `CH_` ou `F_` (de chave ou flag). Nesse caso, as chaves já citadas ficariam: `F_SAVE_LOGFILE` e `F_HIDE_ERROR_MSGS`.

– Admita, você ficou o capítulo inteiro esperando uma piadinha com o Chaves.
– Sim, pois é, pois é, pois é...
– Bem, chegamos ao final do capítulo, agora é a hora da piadinha, né?
– Zás! Isso, isso, isso!
– Mas não vai ter.
– Mas por que causa, motivo, razão ou circunstância?
– É que eu quero evitar a fadiga.
– Como?
– Ops, é que me escapuliu...
– Olha ele, hein, olha ele! Conta tudo pra sua mãe!
– Tá, tá, tá, tá, tááá!
– Tá bom, mas não se irrite.
– Que que foi, que que foi, que que há?
– Mamãe!!!
– Fala, tesouro.
– Não quer fazer a piadinha!
– Você de novo?
– Não! Não! Não sou eu, é ele quem não quer...

– PAFT!
– Vamos, tesouro, não se misture com essa GENTALHA!
– Sim, mamãe. Gentalha! Gentalha! Pfffff
– E da próxima vez, vá regular piadinhas pra sua avó!
– Humpf!
– Seu Madruga, a sua avozinha era uma regulenta?
– Toma!
– Pipipipipipipipi...
– "Pipipipipi" o quê! Só não te dou outra porque...
– ...minha avozinha era uma santa.

Capítulo 4

Opções de linha de comando (-f, --foo)

Trazer mais opções e possibilidades para o usuário é algo que faz parte da evolução natural de um programa. Mas não é elegante forçar o usuário a editar o código para alterar o valor de variáveis, cada vez que precisar de um comportamento diferente. Aprenda a fazer seu programa reconhecer opções de linha de comando, curtas e longas, tornando-o mais flexível e amigável ao usuário.

Você já está bem-acostumado a usar opções para as ferramentas do sistema. É um `-i` no `grep` para ignorar a diferença das maiúsculas e minúsculas, é um `-n` no `sort` para ordenar numericamente, é um `-d` e um `-f` no `cut` para extrair campos, é um `-d` no `tr` para apagar caracteres. Usar uma opção é rápido e fácil, basta ler a man page ou o `--help` e adicioná-la no comando.

Agora, responda-me rápido: o que impede que os seus próprios programas também tenham opções de linha de comando? Não seria interessante o seu programa agir como qualquer outra ferramenta do sistema?

Neste capítulo aprenderemos a colocar opções em nossos programas. Com isso, além da melhor integração com o ambiente (sistema operacional), melhoramos a interface com o usuário, que já está acostumado a usar opções e poderá facilmente informar dados e alterar o comportamento padrão do programa.

O formato "padrão" para as opções

Não há uma convenção ou padrão internacional que force um programa a usar este ou aquele formato para ler parâmetros e argumentos do usuário. Ao longo das décadas, alguns formatos foram experimentados, porém hoje podemos constatar que a maioria usa o formato adotado pelos aplicativos GNU. Acompanhe a análise.

Em um sistema Unix, que possui aplicativos com mais idade que muitos leitores deste livro, é variada a forma que os programas esperam receber as opções de linha de comando. Uma grande parte usa o formato de opções curtas (de uma letra) com o hífen, mas não há um padrão. Veja alguns exemplos:

Formato das opções de programas no Unix

Comando	Formato	Exemplos
`find`	`-<palavra>`	`-name`,`-type`
`ps`	`<letra>`	`a u w x`
`dd`	`<palavra>=`	`if=`, `of=`, `count=`

Há também os aplicativos GNU, que de uma forma ou outra estão ligados à FSF (Free Software Foundation) e preferem a licença GPL. Estes são programas com código mais recente, que vieram para tentar substituir os aplicativos do Unix. Estão presentes em todos os cantos de um sistema Linux, alguns exemplares habitam no Mac e também podem ser instalados no Windows por intermédio do Cygwin. Eles seguem um padrão que não é forçado, porém recomendado e adotado pela maioria.

Formato das opções de aplicativos GNU

Formato	Exemplos	Descrição
`-<letra>`	`-h`, `-V`	Opções curtas, de uma letra
`--<palavra>`	`--help`, `--version`	Opções longas, de uma ou mais palavras

Por ser um formato mais consistente e hoje amplamente utilizado, acostume-se a usá-lo. Este é um padrão inclusive para outras linguagens de programação como Python e Perl (por meio dos módulos getopt).

Para pensar na cama: Os aplicativos GNU são considerados mais modernos e geralmente possuem mais opções e funcionalidades do que seus parentes do Unix. Essa abundância de opções pode ser benéfica ou não, dependendo do seu ponto de vista. O GNU sort deveria mesmo ter a opção `-u` sendo que já existe o `uniq` para fazer isso? E o GNU `ls` com suas mais de 80 opções, incluindo a pérola `--dereference-command-line-symlink-to-dir`, não seria um exagero? Onde fica o limite do "faça apenas UMA tarefa e a faça bem-feita"? Quantidade reflete qualidade? Evolução ou colesterol?

Opções clássicas para usar em seu programa

Há algumas opções clássicas que são usadas pela grande maioria dos programas para um propósito comum. Por exemplo, o `-h` ou `--help` é usado para mostrar uma tela de ajuda. Você já está acostumado a usar esta opção em várias ferramentas do sistema, então, quando for implementar uma tela de ajuda em seu programa, não invente! É `-h` e `--help`. Não surpreenda o usuário ou force-o a aprender novos conceitos sem necessidade.

Algumas opções estão tão presentes por todo o sistema que é considerado um abuso utilizá-las para outros propósitos que não os já estabelecidos. Algumas exceções são programas antigos do Unix que já usavam estas letras para outras funções. Mas você, como novato na

vizinhança, siga as regras e faça seu programa o mais parecido com os já existentes. O uso das seguintes opções é fortemente recomendado (caso aplicável):

Opções estabelecidas

Opção curta	Opção longa	Descrição
`-h`	`--help`	Mostra informações resumidas sobre o uso do programa e sai.
`-V`	`--version`	Mostra a versão do programa e sai (V maiúsculo).
`-v`	`--verbose`	Mostra informações adicionais na saída, informando o usuário sobre o estado atual da execução.
`-q`	`--quiet`	Não mostra nada na saída, é uma execução quieta.
	`--`	Terminador de opções na linha de comando, o que vem depois dele não é considerado opção.

Há outras opções que não são um padrão global, mas que são vistas em vários aplicativos com o mesmo nome e função similar. Então, se aplicável ao seu programa, é aconselhável que você use estes nomes em vez de inventar algo novo:

Opções recomendadas

Opção curta	Opção longa	Descrição	Exemplo
`-c`	`--chars`	Algo com caracteres	`cut -c, od -c, wc -c`
`-d`	`--delimiter`	Caractere(s) usado(s) como delimitador, separador	`cut -d, paste -d`
`-f`	`--file`	Nome do arquivo a ser manipulado	`grep -f, sed -f`
`-i`	`--ignore-case`	Trata letras maiúsculas e minúsculas como iguais	`grep -i, diff -i`
`-n`	`--number`	Algo com números	`cat -n, grep -n, head -n, xargs -n`
`-o`	`--output`	Nome do arquivo de saída	`sort -o, gcc -o`
`-w`	`--word`	Algo com palavras	`grep -w, wc -w`

Como adicionar opções a um programa

Chega de teoria, não é mesmo? Está na hora de fazer a parte divertida

desse negócio: programar. Vamos acompanhar passo a passo a inclusão de várias opções a um programa já existente. Por falar nele, aqui está:

usuarios.sh

```
#!/bin/bash
# usuarios.sh
#
# Mostra os logins e nomes de usuários do sistema
# Obs.: Lê dados do arquivo /etc/passwd
#
# Aurelio, Novembro de 2007
#
cut -d : -f 1,5 /etc/passwd | tr : \\t
```

Este é um programa bem simples que mostra uma listagem dos usuários do sistema, no formato "login TAB nome completo". Seu código resume-se a uma única linha com um `cut` que extrai os campos desejados do arquivo de usuários
(`/etc/passwd`) e um `tr` filtra esta saída transformando todos os dois-pontos em TABs. Veja um exemplo de sua execução:

```
$ ./usuarios.sh
root    System Administrator
daemon  System Services
uucp    Unix to Unix Copy Protocol
lp      Printing Services
postfix Postfix User
www     World Wide Web Server
eppc    Apple Events User
mysql   MySQL Server
sshd    sshd Privilege separation
qtss    QuickTime Streaming Server
mailman Mailman user
amavisd Amavisd User
jabber  Jabber User
tokend  Token Daemon
unknown Unknown User
$
```

Realmente simples, não? Nas próximas páginas, esta pequena gema de

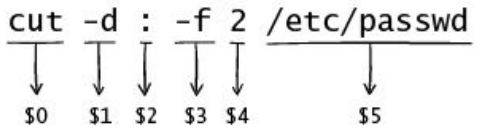
uma linha crescerá para suportar opções e aumentar um pouco a sua gama de funcionalidades. Mas, claro, sem perder o foco inicial: mostrar a lista de usuários. Nada de ler e-mail ou trazer um Wiki embutido ;)

Adicionando as opções -h, -V, --help e --version

Nossa primeira tarefa será adicionar as duas opções mais clássicas de qualquer programa: a `-h` para ajuda e a `-V` para obter a versão. Alguma ideia de como fazer isso? Vou dar um tempinho para você pensar. Como pegar as opções que o usuário digitou? Como reconhecer e processar estas opções? E se o usuário passar uma opção inválida, o que acontece?

E aí, muitas ideias? Vamos começar simples, fazendo a opção `-h` para mostrar uma tela de ajuda para o programa. O primeiro passo é saber exatamente quais foram as opções que o usuário digitou, e, caso tenha sido a `-h`, processá-la.

Para saber o que o usuário digitou na linha de comando, basta conferir o valor das variáveis posicionais `$1`, `$2`, `$3` e amigos. Em `$0` fica o nome do programa, em `$1` fica o primeiro argumento, em `$2` o segundo, e assim por diante.



Parâmetros posicionais $0, $1, $2, ...

Então, se nosso programa foi chamado como `usuarios.sh -h`, a opção estará guardada em `$1`. Aí ficou fácil. Basta conferir se o valor de `$1` é `-h`, em caso afirmativo, mostramos a tela de ajuda e saímos do programa. Traduzindo isso para shell fica:

usuarios.sh (v2)

```
#!/bin/bash
# usuarios.sh
#
# Mostra os logins e nomes de usuários do sistema
# Obs.: Lê dados do arquivo /etc/passwd
#
```

```
# Versão 1: Mostra usuários e nomes separados por TAB
# Versão 2: Adicionado suporte à opção -h
#
# Aurelio, Novembro de 2007
#
MENSAGEM_USO="
Uso: $0 [-h]
  -h      Mostra esta tela de ajuda e sai
"

# Tratamento das opções de linha de comando
if test "$1" = "-h"
then
    echo "$MENSAGEM_USO"
    exit 0
fi

# Processamento
cut -d : -f 1,5 /etc/passwd | tr : \\t
```

Lá nos cabeçalhos deixamos claro que esta é a segunda versão do programa e que ela traz de novidade a opção `-h`. A mensagem de uso é guardada em uma variável, usando o `$0` para obter o nome do programa. Isso é bom porque podemos mudar o nome do programa sem precisar nos preocupar em mudar a mensagem de ajuda junto. Em seguida, um `if` testa o conteúdo de `$1`, se for o `-h` que queremos, mostra a ajuda e sai com código de retorno zero, que significa: tudo certo.

```
$ ./usuarios-2.sh -h
Uso: ./usuarios-2.sh [-h]
-h            Mostra esta tela de ajuda e sai
$ ./usuarios-2.sh -X
root    System Administrator
daemon  System Services
uucp    Unix to Unix Copy Protocol
lp      Printing Services
postfix Postfix User
www     World Wide Web Server
eppc    Apple Events User
mysql   MySQL Server
```

```
sshd    sshd Privilege separation
qtss    QuickTime Streaming Server
mailman Mailman user
amavisd Amavisd User
jabber  Jabber User
tokend  Token Daemon
unknown Unknown User
$
```

Funciona! Quando passamos o `-h`, foi mostrada somente a mensagem de ajuda e o programa foi terminado. Já quando passamos a opção `-X`, ela foi ignorada e o programa continuou sua execução normal. Como é fácil adicionar opções a um programa em shell!

Na mensagem de ajuda, o `[-h]` entre colchetes indica que este parâmetro é opcional, ou seja, você pode usá-lo, mas não é obrigatório.

De maneira similar, vamos adicionar a opção `-V` para mostrar a versão do programa. Novamente testaremos o conteúdo de `$1`, então basta adicionar mais um teste ao `if`, desta vez procurando por `-V`:

```
# Tratamento das opções de linha de comando
if test "$1" = "-h"
then
    echo "$MENSAGEM_USO"
    exit 0
elif test "$1" = "-V"
then
    # mostra a versão
    ...
fi
```

E a cada nova opção, o `if` vai crescer mais. Mmmm, espera. Em vez de fazer um `if` monstruoso, cheio de braços, aqui é bem melhor usar o `case`. De brinde ainda ganhamos a opção "`*`" para pegar as opções inválidas. Veja como fica melhor e mais legível:

```
# Tratamento das opções de linha de comando
case "$1" in
    -h)
        echo "$MENSAGEM_USO"
        exit 0
```

```
    ;;
    -V)
        # mostra a versão
    ;;
    *)
        # opção inválida
    ;;
esac
```

Para a opção inválida é fácil, basta mostrar uma mensagem na tela informando ao usuário o erro e sair do programa com código de retorno 1. Um `echo` e um `exit` são suficientes. A versão basta mostrar uma única linha com o nome do programa e a sua versão atual, então sai com retorno zero. Mais um `echo` e um `exit`. Parece fácil.

usuarios.sh (v3)

```
#!/bin/bash
# usuarios.sh
#
# Mostra os logins e nomes de usuários do sistema
# Obs.: Lê dados do arquivo /etc/passwd
#
# Versão 1: Mostra usuários e nomes separados por TAB
# Versão 2: Adicionado suporte à opção -h
# Versão 3: Adicionado suporte à opção -V e opções inválidas
#
# Aurelio, Novembro de 2007
#
MENSAGEM_USO="
Uso: $0 [-h | -V]
  -h    Mostra esta tela de ajuda e sai
  -V    Mostra a versão do programa e sai
"
# Tratamento das opções de linha de comando
case "$1" in
    -h)
        echo "$MENSAGEM_USO"
        exit 0
    ;;
```

```
    -V)
        echo $0 Versão 3
        exit 0
    ;;
    *)
        echo Opção inválida: $1
        exit 1
    ;;
esac
# Processamento
cut -d : -f 1,5 /etc/passwd | tr : \\t
```

Mais uma vez o cabeçalho foi atualizado para informar as mudanças. Isso deve se tornar um hábito, uma regra que não pode ser quebrada. Acostume-se desde já. Em seguida, a mensagem de uso agora mostra que o programa também possui a opção `-V`. O pipe em `[-h | -V]` informa que você pode usar a opção `-h` ou a opção `-V`, e ambas são opcionais (indicado pelos colchetes). Depois vem o `case` com cada opção bem alinhada, tornando a leitura agradável. No `-V` é mostrada a versão atual e a saída é normal. O asterisco vale para qualquer outra opção fora o `-h` e o `-V`, e além de mostrar a mensagem informando que a opção digitada é inválida, sai com código de erro (1).

```
$ ./usuarios-3.sh -h
Uso: ./usuarios-3.sh [-h | -V]
  -h    Mostra esta tela de ajuda e sai
  -V    Mostra a versão do programa e sai
$ ./usuarios-3.sh -V
./usuarios-3.sh Versão 3
$ ./usuarios-3.sh -X
Opção inválida: -X
$ ./usuarios-3.sh
Opção inválida:
$
```

Tudo funcionando! Ops, quase. Quando não foi passada nenhuma opção, o programa deveria mostrar a lista de usuários normalmente, mas acabou caindo no asterisco do `case`... Primeira melhoria a ser feita: só testar pela opção inválida se houver `$1`; caso contrário, continue.

```
    *)
        if test -n "$1"
        then
            echo Opção inválida: $1
            exit 1
        fi
    ;;
```

Outra alteração interessante seria eliminar o "`./`" do nome do programa tanto na opção `-h` quanto na `-V`. Ele é mostrado porque o `$0` sempre mostra o nome do programa exatamente como ele foi chamado, inclusive com `PATH`. O comando `basename` vai nos ajudar nesta hora, arrancando o `PATH` e deixando somente o nome do arquivo.

```
MENSAGEM_USO="
Uso: $(basename "$0") [-h | -V]
...
"
```

Já que estamos aqui, com o `-h` e o `-V` funcionando, que tal também adicionar suas opções equivalentes `--help` e `--version`? Vai ser muito mais fácil do que você imagina. Lembre-se de que dentro do `case` é possível especificar mais de uma alternativa para cada bloco? Então, alterando somente duas linhas do programa ganhamos de brinde mais duas opções.

```
case "$1" in
    -h | --help)
        ...
    ;;
    -V | --version)
        ...
    ;;
    *)
        ...
    ;;
esac
```

Uma última alteração seria melhorar este esquema de mostrar a versão do programa. Aquele número ali fixo dentro do `case` tende a ser esquecido. Você vai modificar o programa, lançar outra versão e não vai se lembrar de aumentar o número da opção `-V`. Por outro lado, o número da

versão está sempre lá no cabeçalho, no registro das mudanças. Será que...

```
$ grep '^# Versão ' usuarios-3.sh
# Versão 1: Mostra usuários e nomes separados por TAB
# Versão 2: Adicionado suporte à opção -h
# Versão 3: Adicionado suporte à opção -V e opções inválidas
$ grep '^# Versão ' usuarios-3.sh | tail -1
# Versão 3: Adicionado suporte à opção -V e opções inválidas
$ grep '^# Versão ' usuarios-3.sh | tail -1 | cut -d : -f 1
# Versão 3
$ grep '^# Versão ' usuarios-3.sh | tail -1 | cut -d : -f 1 | tr -
  d \#
Versão 3
$
```

Ei, isso foi legal! Além de extrair a versão do programa automaticamente, ainda nos forçamos a sempre registrar nos cabeçalhos o que mudou na versão nova, para não quebrar o `-V`. Simples e eficiente. Vejamos como ficou esta versão nova do código.

⌨ usuarios.sh (v4)

```
#!/bin/bash
# usuarios.sh
#
# Mostra os logins e nomes de usuários do sistema
# Obs.: Lê dados do arquivo /etc/passwd
#
# Versão 1: Mostra usuários e nomes separados por TAB
# Versão 2: Adicionado suporte à opção -h
# Versão 3: Adicionado suporte à opção -V e opções inválidas
# Versão 4: Arrumado bug quando não tem opções, basename no
#           nome do programa, -V extraindo direto dos cabeçalhos,
#           adicionadas opções --help e --version
#
# Aurelio, Novembro de 2007
#
MENSAGEM_USO="
Uso: $(basename "$0") [-h | -V]
  -h, --help        Mostra esta tela de ajuda e sai
  -V, --version     Mostra a versão do programa e sai
```

```
"
# Tratamento das opções de linha de comando
case "$1" in
    -h | --help)
        echo "$MENSAGEM_USO"
        exit 0
    ;;
    -V | --version)
        echo -n $(basename "$0")
        # Extrai a versão diretamente dos cabeçalhos do programa
        grep '^# Versão ' "$0" | tail -1 | cut -d : -f 1 | tr -d
  \#
        exit 0
    ;;
    *)
        if test -n "$1"
        then
            echo Opção inválida: $1
            exit 1
        fi
    ;;
esac
# Processamento
cut -d : -f 1,5 /etc/passwd | tr : \\t
```

Agora, sim, podemos considerar que o programa ficou estável após a adição de quatro opções de linha de comando, além do tratamento de opções desconhecidas. Antes de continuar adicionando opções novas, é bom conferir se está mesmo tudo em ordem com o código atual.

```
$ ./usuarios-4.sh -h
Uso: usuarios-4.sh [-h | -V]
  -h, --help        Mostra esta tela de ajuda e sai
  -V, --version     Mostra a versão do programa e sai
$ ./usuarios-4.sh --help
Uso: usuarios-4.sh [-h | -V]
  -h, --help        Mostra esta tela de ajuda e sai
  -V, --version     Mostra a versão do programa e sai
$ ./usuarios-4.sh -V
usuarios-4.sh Versão 4
```

```
$ ./usuarios-4.sh --version
usuarios-4.sh Versão 4
$ ./usuarios-4.sh --foo
Opção inválida: --foo
$ ./usuarios-4.sh -X
Opção inválida: -X
$ ./usuarios-4.sh
root    System Administrator
daemon  System Services
uucp    Unix to Unix Copy Protocol
lp      Printing Services
postfix Postfix User
www     World Wide Web Server
eppc    Apple Events User
mysql   MySQL Server
sshd    sshd Privilege separation
qtss    QuickTime Streaming Server
mailman Mailman user
amavisd Amavisd User
jabber  Jabber User
tokend  Token Daemon
unknown Unknown User
$
```

Adicionando opções específicas do programa

Colocar as opções clássicas no programa foi fácil, não? Um `case` toma conta de tudo, cada opção tendo seu próprio cantinho dentro do programa. Faz o que tem que fazer e termina com um `exit` de zero ou um para informar se está tudo bem. Se o fluxo de execução não entrou em nenhum destes cantinhos, o programa segue sua rota normal, mostrando a lista de usuários do sistema.

Agora vamos adicionar algumas opções que possuem um comportamento diferente. Elas vão ter seu cantinho dentro do `case`, mas em vez de terminar a execução do programa, vão definir variáveis e no final a lista de usuários também deverá ser mostrada. Isso vai requerer uma alteração estrutural em nosso programa. Mas vamos com calma.

Primeiro, o mais importante é avaliar: que tipo de opção seria útil adicionar ao nosso programa? Como programadores talentosos e criativos que somos, podemos adicionar qualquer coisa ao código, o céu é o limite. Mas será que uma enorme quantidade de opções reflete qualidade? Adicionar toda e qualquer opção que vier à mente é realmente benéfico ao programa e aos usuários? Avalie o seguinte:

- A opção `--FOO` vai trazer benefícios à maioria dos usuários ou apenas a um grupo muito seleto de usuários avançados? Estes usuários avançados já não conseguem se virar sem esta opção?
- A opção `--FOO` vai trazer muita complexidade ao código atual? Sua implementação será muito custosa ao programador?
- A opção `--FOO` vai influir no funcionamento da opção `--BAR` já existente?
- A opção `--FOO` está dentro do escopo do programa? Ela não vai descaracterizar seu programa?
- A opção `--FOO` é auto-explicável? Ou é preciso um parágrafo inteiro para descrever o que ela faz? Se está difícil dar um nome, pode ser um sintoma que ela nem deveria existir em primeiro lugar...
- A opção `--FOO` é realmente necessária? Não é possível fazer a mesma tarefa usando uma combinação das opções já existentes?
- A opção `--FOO` é realmente necessária? Não seria ela apenas "cosmética", não adicionando nenhum real valor ao seu programa?
- A opção `--FOO` é **REALMENTE** necessária? :)

A ênfase na real necessidade de se colocar uma opção é justificada pela tendência que nós, programadores, temos de ir adicionando funcionalidades no código, sem pensar muito nelas. Afinal, programar é divertido! É muito melhor ficar horas programando novidades do que "perder tempo" avaliando se aquilo realmente será útil.

Essa tendência leva a transformar em um monstro indomável aquele programinha rápido e eficiente das primeiras versões. No início, o programa tinha poucas opções, mas as executava instantaneamente e com perfeição. Com o passar do tempo, muitas opções foram adicionadas e

hoje ele demora para dar uma resposta, às vezes interrompendo sua execução sem aviso prévio. Deu pau!

Para não ser apenas mais um personagem dessa história que se repete diariamente, faça um favor a si mesmo: leia a lista anterior toda vez que pensar em adicionar uma opção nova ao seu programa. Pense, analise, avalie, questione. Se a opção passar por todas estas barreiras e provar ser realmente útil, implemente-a.

Cara chato, né? Mas se não tiver um chato para te ensinar as coisas chatas (porém importantes), como você vai evoluir como programador? A chatice de hoje é a qualidade de amanhã em seu trabalho. Invista no seu potencial e colha os frutos no futuro!

Agora que acabaram os conselhos da terceira-idade, podemos continuar :)

Vamos decidir quais opções incluir em nosso programa. Não seremos tão rigorosos quanto à utilidade das opções, visto que o objetivo aqui é ser didático. Mas no seu programa, já sabe... Vejamos, o que o `usuarios.sh` faz é listar os usuários do sistema. Quais variações desta listagem poderiam ser interessantes ao usuário? Talvez uma opção para que a listagem apareça ordenada alfabeticamente? Isso pode ser útil.

```
$ ./usuarios-4.sh | sort
amavisd Amavisd User
daemon  System Services
eppc    Apple Events User
jabber  Jabber User
lp      Printing Services
mailman Mailman user
mysql   MySQL Server
postfix Postfix User
qtss    QuickTime Streaming Server
root    System Administrator
sshd    sshd Privilege separation
tokend  Token Daemon
unknown Unknown User
uucp    Unix to Unix Copy Protocol
www     World Wide Web Server
$
```

Sabemos que existe o comando sort e que ele ordena as linhas. Mas o usuário não é obrigado a saber disso. Ele no máximo lerá a nossa tela de ajuda e ali saberá das possibilidades de uso do programa. Então, apesar de ter uma implementação trivial, o usuário se beneficiará com esta opção. Nosso programa terá duas opções novas `-s` e `--sort` que serão equivalentes e servirão para que a lista seja ordenada.

Sim, poderia ser `-o` e `--ordena` para que as opções ficassem em português. Mas como já temos `--help` e `--version` em inglês, é preferível manter o padrão do que misturar os idiomas. Outra opção seria deixar tudo em português alterando as opções atuais para `--ajuda` e `--versao` (sem acentos para evitar problemas!). Avalie o perfil de seus usuários e decida qual idioma utilizar.

Para adicionar esta opção nova temos que incluí-la na mensagem de ajuda, e também dentro do `case`. Alguma variável global será necessária para indicar se a saída será ou não ordenada, e esta opção mudará o valor desta variável. Um esqueleto deste esquema seria assim:

```
ordenar=0    # A saída deverá ser ordenada?
...
# Tratamento das opções de linha de comando
case "$1" in
    -s | --sort)
        ordenar=1
    ;;
    ...
esac
...
if test "$ordenar" = 1
then
    # ordena a listagem
fi
```

No início do programa desligamos a chave de ordenação (`$ordenar`), colocando um valor padrão zero. Então são processadas as opções de linha de comando dentro do case. Caso o usuário tenha passado a opção `--sort`, a chave é ligada. Lá no final do programa há um `if` que testa o valor da chave, se ela estiver ligada a listagem é ordenada.

Este é o jeito limpo de implementar uma opção nova. Tudo o que ela faz é mudar o valor de uma variável de nosso programa. Lá na frente o

programa sabe o que fazer com essa variável. O efeito é exatamente o mesmo de o usuário editar o programa e mudar o valor padrão da chave (`ordenar=1`) no início. A vantagem em se usar uma opção na linha de comando é evitar que o usuário edite o código, pois assim corre o risco de bagunçá-lo.

Para implementar a ordenação em si, a primeira ideia que vem à cabeça é fazer tudo dentro do `if`. Bastaria repetir a linha do `cut | tr` colocando um `sort` no final e pronto, ambas as possibilidades estariam satisfeitas. Acompanhe:

```
if test "$ordenar" = 1
then
    cut -d : -f 1,5 /etc/passwd | tr : \\t | sort
else
    cut -d : -f 1,5 /etc/passwd | tr : \\t
fi
```

Mas isso traz vários problemas. O primeiro é a repetição de código, o que nunca é benéfico. Serão dois lugares para mudar caso alguma alteração precise ser feita na dupla `cut | tr`. Esta solução também não irá ajudar quando precisarmos adicionar outras opções que também manipulem a listagem. O melhor é fazer cada tarefa isoladamente, para que a independência entre elas garanta que todas funcionem simultaneamente: primeiro extrai a lista e guarda em uma variável. Depois ordena, se necessário.

```
# Extrai a listagem
lista=$(cut -d : -f 1,5 /etc/passwd)

# Ordena a listagem (se necessário)
if test "$ordenar" = 1
then
    lista=$(echo "$lista" | sort)
fi

# Mostra o resultado para o usuário
echo "$lista" | tr : \\t
```

Assim o código fica maior, porém muito mais flexível e poderoso. A vantagem da separação das tarefas ficará evidente quando adicionarmos

mais opções ao programa. Já são muitas mudanças, hora de lançar uma versão nova:

⌨ usuarios.sh (v5)

```
#!/bin/bash
# usuarios.sh
#
# Mostra os logins e nomes de usuários do sistema
# Obs.: Lê dados do arquivo /etc/passwd
#
# Versão 1: Mostra usuários e nomes separados por TAB
# Versão 2: Adicionado suporte à opção -h
# Versão 3: Adicionado suporte à opção -V e opções inválidas
# Versão 4: Arrumado bug quando não tem opções, basename no
#           nome do programa, -V extraindo direto dos cabeçalhos,
#           adicionadas opções --help e --version
# Versão 5: Adicionadas opções -s e --sort
#
# Aurelio, Novembro de 2007
#
ordenar=0     # A saída deverá ser ordenada?
MENSAGEM_USO="
Uso: $(basename "$0") [-h | -V | -s]
  -s, --sort     Ordena a listagem alfabeticamente
  -h, --help      Mostra esta tela de ajuda e sai
  -V, --version    Mostra a versão do programa e sai
"
# Tratamento das opções de linha de comando
case "$1" in
    -s | --sort)
        ordenar=1
    ;;
    -h | --help)
        echo "$MENSAGEM_USO"
        exit 0
    ;;
    -V | --version)
        echo -n $(basename "$0")
        # Extrai a versão diretamente dos cabeçalhos do programa
```

```
        grep '^# Versão ' "$0" | tail -1 | cut -d : -f 1 | tr -d
  \#
        exit 0
    ;;
    *)
        if test -n "$1"
        then
            echo Opção inválida: $1
            exit 1
        fi
    ;;
esac
# Extrai a listagem
lista=$(cut -d : -f 1,5 /etc/passwd)
# Ordena a listagem (se necessário)
if test "$ordenar" = 1
then
    lista=$(echo "$lista" | sort)
fi
# Mostra o resultado para o usuário
echo "$lista" | tr : \\t
```

O código está simples de entender, é certo que a opção nova vai funcionar. Mas não custa testar, são apenas alguns segundos investidos. E vá que aparece algum bug alienígena que nossos olhos terráqueos não consigam captar?

```
$ ./usuarios-5.sh --sort
amavisd Amavisd User
daemon  System Services
eppc    Apple Events User
jabber  Jabber User
lp      Printing Services
mailman Mailman user
mysql   MySQL Server
postfix Postfix User
qtss    QuickTime Streaming Server
root    System Administrator
sshd    sshd Privilege separation
tokend  Token Daemon
```

```
unknown Unknown User
uucp    Unix to Unix Copy Protocol
www     World Wide Web Server
$
```

Ufa, não foi desta vez que os ETs tiraram o nosso sono... Agora que o código do programa está com uma boa estrutura, fica fácil adicionar duas opções novas: uma para inverter a ordem da lista e outra para mostrar a saída em letras maiúsculas. Digamos `--reverse` e `--uppercase`. Como programadores experientes em shell, sabemos que o `tac` inverte as linhas de um texto e que o `tr` pode converter um texto para maiúsculas. Assim, o código para suportar estas opções novas será tão trivial quanto o do `--sort`.

```
$ ./usuarios-5.sh --sort | tac
www     World Wide Web Server
uucp    Unix to Unix Copy Protocol
unknown Unknown User
tokend  Token Daemon
sshd    sshd Privilege separation
root    System Administrator
qtss    QuickTime Streaming Server
postfix Postfix User
mysql   MySQL Server
mailman Mailman user
lp      Printing Services
jabber  Jabber User
eppc    Apple Events User
daemon  System Services
amavisd Amavisd User
$ ./usuarios-5.sh --sort | tac | tr a-z A-Z
WWW     WORLD WIDE WEB SERVER
UUCP    UNIX TO UNIX COPY PROTOCOL
UNKNOWN UNKNOWN USER
TOKEND  TOKEN DAEMON
SSHD    SSHD PRIVILEGE SEPARATION
ROOT    SYSTEM ADMINISTRATOR
QTSS    QUICKTIME STREAMING SERVER
POSTFIX POSTFIX USER
MYSQL   MYSQL SERVER
```

```
MAILMAN MAILMAN USER
LP      PRINTING SERVICES
JABBER  JABBER USER
EPPC    APPLE EVENTS USER
DAEMON  SYSTEM SERVICES
AMAVISD AMAVISD USER
$
```

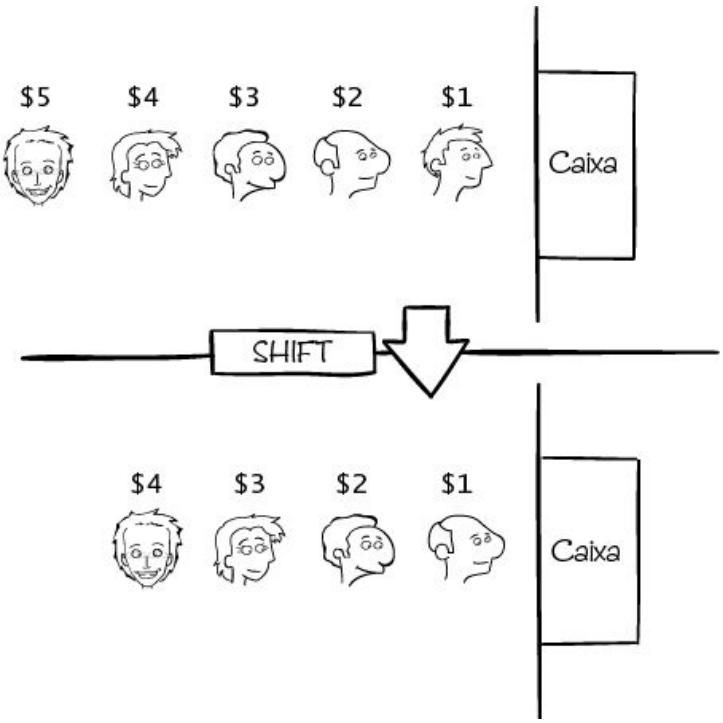
Ah, como é bom programar em shell e ter à mão todas estas ferramentas já prontas que fazem todo o trabalho sujo, não é mesmo? Um único detalhe que está faltando no código de nosso programa é que ele sempre verifica o valor de `$1`, não estando pronto para receber múltiplas opções. E agora ele precisará disso, pois o usuário pode querer usar todas ao mesmo tempo:

```
usuarios.sh --sort --reverse --uppercase
```

Volte algumas páginas e analise o código-fonte do programa. O que precisa ser feito para que `$2`, `$3` e outros também sejam interpretados pelo `case`? Como processar um número variável de opções de linha de comando?

O segredo da resposta está no comando `shift`. Ele remove o `$1`, fazendo com que todos os parâmetros posicionais andem uma posição na fila. Assim o `$2` vira `$1`, o `$3` vira `$2`, e assim por diante.

Imagine uma fila de banco onde você é o quinto (`$5`) cliente. Quando o primeiro da fila for atendido, a fila anda (`shift`) e você será o quarto (`$4`). Depois o terceiro, e assim por diante, até você ser o primeiro ($1) para finalmente ser atendido. Essa é a maneira shell de fazer loop nos parâmetros posicionais: somente o primeiro da fila é atendido, enquanto os outros esperam. Em outras palavras: lidaremos sempre com o `$1`, usando o `shift` para fazer a fila andar.



Funcionamento do comando shift

Traduzindo este comportamento para códigos, teremos um loop `while` que monitorará o valor de `$1`. Enquanto esta variável não for nula, temos opções de linha de comando para processar. Dentro do loop fica o `case` que já conhecemos. Ele já sabe consultar o valor de `$1` e tomar suas ações. Ele é como se fosse o caixa do banco. Uma vez processada a opção da vez, ela é liberada para que a fila ande (`shift`) e o loop continue até não haver mais o `$1`:

```
# Tratamento das opções de linha de comando
while test -n "$1"
do
    case "$1" in
        -s | --sort)
            ordenar=1
        ;;
        ...
    esac
    # Opção $1 já processada, a fila deve andar
    shift
done
```

Veja como ficou a versão nova do código com este loop implementado, bem como as opções novas `--reverse` e `--uppercase`. Perceba também que a tela de ajuda mudou um pouco, usando o formato `[OPÇÕES]` para simplificar a sintaxe de uso, indicando que todas as opções seguintes não são obrigatórias (colchetes). Outra mudança dentro do `case`, foi a retirada do teste de existência do parâmetro `$1`, que era feito na opção padrão (asterisco). Ele não é mais necessário visto que o `while` já está fazendo esta verificação.

usuarios.sh (v6)

```
#!/bin/bash
# usuarios.sh
#
# Mostra os logins e nomes de usuários do sistema
# Obs.: Lê dados do arquivo /etc/passwd
#
# Versão 1: Mostra usuários e nomes separados por TAB
# Versão 2: Adicionado suporte à opção -h
# Versão 3: Adicionado suporte à opção -V e opções inválidas
# Versão 4: Arrumado bug quando não tem opções, basename no
#           nome do programa, -V extraindo direto dos cabeçalhos,
#           adicionadas opções --help e --version
# Versão 5: Adicionadas opções -s e --sort
# Versão 6: Adicionadas opções -r, --reverse, -u, --uppercase,
#           leitura de múltiplas opções (loop)
#
# Aurelio, Novembro de 2007
#

ordenar=0          # A saída deverá ser ordenada?
inverter=0       # A saída deverá ser invertida?
maiusculas=0     # A saída deverá ser em maiúsculas?
MENSAGEM_USO="
Uso: $(basename "$0") [OPÇÕES]
OPÇÕES:
  -r, --reverse     Inverte a listagem
  -s, --sort          Ordena a listagem alfabeticamente
  -u, --uppercase   Mostra a listagem em MAIÚSCULAS
  -h, --help          Mostra esta tela de ajuda e sai
```

```
  -V, --version       Mostra a versão do programa e sai
"
# Tratamento das opções de linha de comando
while test -n "$1"
do
    case "$1" in
        -s | --sort)
            ordenar=1
        ;;

        -r | --reverse)
            inverter=1
        ;;

        -u | --uppercase)
            maiusculas=1
        ;;
        -h | --help)
            echo "$MENSAGEM_USO"
            exit 0
        ;;
        -V | --version)
            echo -n $(basename "$0")
            # Extrai a versão diretamente dos cabeçalhos do
  programa
            grep '^# Versão ' "$0" | tail -1 | cut -d : -f 1 | tr
  -d \#
            exit 0
        ;;
        *)
            echo Opção inválida: $1
            exit 1
        ;;
    esac

    # Opção $1 já processada, a fila deve andar
    shift
done
# Extrai a listagem
```

```
lista=$(cut -d : -f 1,5 /etc/passwd)
# Ordena a listagem (se necessário)
if test "$ordenar" = 1
then
    lista=$(echo "$lista" | sort)
fi
# Inverte a listagem (se necessário)
if test "$inverter" = 1
then
    lista=$(echo "$lista" | tac)
fi
# Converte para maiúsculas (se necessário)
if test "$maiusculas" = 1
then
    lista=$(echo "$lista" | tr a-z A-Z)
fi
# Mostra o resultado para o usuário
echo "$lista" | tr : \\t
```

Antes de passar para a próxima opção a ser incluída, cabe aqui uma otimização de código que melhorará a sua legibilidade. Alteraremos dois trechos, sem incluir nenhuma funcionalidade nova, apenas o código será reformatado para ficar mais compacto e, ao mesmo tempo, mais legível. Isso não acontece sempre, geralmente compactar significa tornar ruim de ler. Mas como neste caso as linhas são praticamente iguais, mudando apenas uma ou outra palavra, o alinhamento beneficia a leitura:

Código atual	Código compacto

`case "$1" in`
`  -s | --sort)`
`  ordenar=1`
`  ;;`
`  -r | --reverse)`
`  inverter=1`
`  ;;`
`  -u | --uppercase)`
`  maiusculas=1`
`  ;;`
`  ...`
`esac`
	`case "$1" in`
	`  # Opções que ligam/desligam chaves`
	`  -s | --sort  ) ordenar=1  ;;`
	`  -r | --reverse  ) inverter=1  ;;`
	`  -u | --uppercase) maiusculas=1 ;;`
	`  ...`
	`esac`

`# Ordena a listagem (se`	`# Ordena, inverte ou converte para`

```
  necessário)
if test "$ordenar" = 1
then
    lista=$(echo "$lista" |
  sort)
fi
# Inverte a listagem (se
  necessário)
if test "$inverter" = 1
then
    lista=$(echo "$lista" |
  tac)
fi
# Converte para maiúsculas
  (se nece...
if test "$maiusculas" = 1
then
    lista=$(echo "$lista" |
  tr a-z A-Z)
fi
```

```
  maiúsculas (se necessário)
test "$ordenar"    = 1 && lista=$(echo
  "$lista" | sort)
test "$inverter"   = 1 && lista=$(echo
  "$lista" | tac)
test "$maiusculas" = 1 && lista=$(echo
  "$lista" | tr a-z A-Z)
```

Adicionando opções com argumentos

Até agora vimos opções que funcionam como chaves que ligam funcionalidades. Elas não possuem argumentos, sua presença basta para indicar que tal funcionalidade deve ser ligada (ou em alguns casos, desligada).

A última opção que adicionaremos ao nosso programa será diferente. Ela se chamará `--delimiter`, e assim como no `cut`, esta opção indicará qual caractere será usado como delimitador. Em nosso contexto, isso se aplica ao separador entre o login e o nome completo do usuário, que hoje está fixo no TAB. Veja a diferença na saída do programa, com esta opção nova:

```
$ ./usuarios-7.sh | grep root
root    System Administrator
$ ./usuarios-7.sh --delimiter , | grep root
root,System Administrator
$
```

Assim o usuário ganha flexibilidade no formato de saída, podendo adaptar o texto ao seu gosto. A implementação da funcionalidade é tranquila, basta usar uma nova variável que guardará o delimitador. Mas

e dentro do `case`, como fazer para obter o argumento da opção?

```
# Tratamento das opções de linha de comando
while test -n "$1"
do
    case "$1" in
        -d | --delimiter)
            shift
            delim="$1"
        ;;
        ...
    esac
    # Opção $1 já processada, a fila deve andar
    shift
done
```

Primeiro é feito um `shift` "manual" para que a opção `-d` (ou `--delimiter`) seja descartada, fazendo a fila andar e assim seu argumento fica sendo o `$1`, que é então salvo em `$delim`. Simples, não? Sempre que tiver uma opção que tenha argumentos, use esta técnica. Ah, mas e se o usuário não passou nenhum argumento, como em `usuarios.sh -d`?

```
# Tratamento das opções de linha de comando
while test -n "$1"
do
    case "$1" in
        -d | --delimiter)
            shift
            delim="$1"
            if test -z "$delim"
            then
                echo "Faltou o argumento para a -d"
                exit 1
            fi
        ;;
        ...
    esac
    # Opção $1 já processada, a fila deve andar
```

```
    shift
done
```

Agora sim, a exceção foi tratada e o usuário informado sobre seu erro. Chega, né? Quase 100 linhas está bom demais para um programa que inicialmente tinha apenas 10. Mas, em compensação, agora ele possui 12 opções novas (seis curtas e suas alternativas longas) e muita flexibilidade de modificação de sua execução, além de estar mais amigável com o usuário por ter uma tela de ajuda. Veja como ficou a última versão:

usuarios.sh (v7)

```
#!/bin/bash
# usuarios.sh
#
# Mostra os logins e nomes de usuários do sistema
# Obs.: Lê dados do arquivo /etc/passwd
#
# Versão 1: Mostra usuários e nomes separados por TAB
# Versão 2: Adicionado suporte à opção -h
# Versão 3: Adicionado suporte à opção -V e opções inválidas
# Versão 4: Arrumado bug quando não tem opções, basename no
#           nome do programa, -V extraindo direto dos cabeçalhos,
#           adicionadas opções --help e --version
# Versão 5: Adicionadas opções -s e --sort
# Versão 6: Adicionadas opções -r, --reverse, -u, --uppercase,
#           leitura de múltiplas opções (loop)
# Versão 7: Melhorias no código para que fique mais legível,
#           adicionadas opções -d e --delimiter
#
# Aurelio, Novembro de 2007
#
ordenar=0        # A saída deverá ser ordenada?
inverter=0       # A saída deverá ser invertida?
maiusculas=0     # A saída deverá ser em maiúsculas?
delim='\t'       # Caractere usado como delimitador de saída
MENSAGEM_USO="
Uso: $(basename "$0") [OPÇÕES]
OPÇÕES:
  -d, --delimiter C  Usa o caractere C como delimitador
```

```
  -r, --reverse      Inverte a listagem
  -s, --sort         Ordena a listagem alfabeticamente
  -u, --uppercase    Mostra a listagem em MAIÚSCULAS
  -h, --help         Mostra esta tela de ajuda e sai
  -V, --version      Mostra a versão do programa e sai
"
# Tratamento das opções de linha de comando
while test -n "$1"
do
    case "$1" in
        # Opções que ligam/desligam chaves
        -s | --sort     ) ordenar=1    ;;
        -r | --reverse  ) inverter=1   ;;
        -u | --uppercase) maiusculas=1 ;;

        -d | --delimiter)
            shift
            delim="$1"

            if test -z "$delim"
            then
                echo "Faltou o argumento para a -d"
                exit 1
            fi
        ;;
        -h | --help)
            echo "$MENSAGEM_USO"
            exit 0
        ;;
        -V | --version)
            echo -n $(basename "$0")
            # Extrai a versão diretamente dos cabeçalhos do
  programa
            grep '^# Versão ' "$0" | tail -1 | cut -d : -f 1 | tr
  -d \#
            exit 0
        ;;
        *)
            echo Opção inválida: $1
```

```
            exit 1
        ;;
    esac

    # Opção $1 já processada, a fila deve andar
    shift
done
# Extrai a listagem
lista=$(cut -d : -f 1,5 /etc/passwd)
# Ordena, inverte ou converte para maiúsculas (se necessário)
test "$ordenar"    = 1 && lista=$(echo "$lista" | sort)
test "$inverter"   = 1 && lista=$(echo "$lista" | tac)
test "$maiusculas" = 1 && lista=$(echo "$lista" | tr a-z A-Z)
# Mostra o resultado para o usuário
echo "$lista" | tr : "$delim"
```

Ufa, como cresceu! Você pode usar este programa como modelo para seus próprios programas que terão opções de linha de comando, toda essa estrutura é reaproveitável. Para fechar de vez com chave de ouro, agora faremos o teste final de funcionamento, usando todas as opções, inclusive misturando-as. Espero que não apareça nenhum bug :)

```
$ ./usuarios-7.sh --help
Uso: usuarios-7.sh [OPÇÕES]
OPÇÕES:
  -d, --delimiter C  Usa o caractere C como delimitador
  -r, --reverse      Inverte a listagem
  -s, --sort         Ordena a listagem alfabeticamente
  -u, --uppercase    Mostra a listagem em MAIÚSCULAS
  -h, --help         Mostra esta tela de ajuda e sai
  -V, --version      Mostra a versão do programa e sai
$ ./usuarios-7.sh -V
usuarios-7.sh Versão 7
$ ./usuarios-7.sh
root    System Administrator
daemon  System Services
uucp    Unix to Unix Copy Protocol
lp      Printing Services
postfix Postfix User
```

```
www     World Wide Web Server
eppc    Apple Events User
mysql   MySQL Server
sshd    sshd Privilege separation
qtss    QuickTime Streaming Server
mailman Mailman user
amavisd Amavisd User
jabber  Jabber User
tokend  Token Daemon
unknown Unknown User
$ ./usuarios-7.sh --sort
amavisd Amavisd User
daemon  System Services
eppc    Apple Events User
jabber  Jabber User
lp      Printing Services
mailman Mailman user
mysql   MySQL Server
postfix Postfix User
qtss    QuickTime Streaming Server
root    System Administrator
sshd    sshd Privilege separation
tokend  Token Daemon
unknown Unknown User
uucp    Unix to Unix Copy Protocol
www     World Wide Web Server
$ ./usuarios-7.sh --sort --reverse
www     World Wide Web Server
uucp    Unix to Unix Copy Protocol
unknown Unknown User
tokend  Token Daemon
sshd    sshd Privilege separation
root    System Administrator
qtss    QuickTime Streaming Server
postfix Postfix User
mysql   MySQL Server
mailman Mailman user
lp      Printing Services
jabber  Jabber User
```

```
eppc    Apple Events User
daemon  System Services
amavisd Amavisd User
$ ./usuarios-7.sh --sort --reverse --uppercase
WWW     WORLD WIDE WEB SERVER
UUCP    UNIX TO UNIX COPY PROTOCOL
UNKNOWN UNKNOWN USER
TOKEND  TOKEN DAEMON
SSHD    SSHD PRIVILEGE SEPARATION
ROOT    SYSTEM ADMINISTRATOR
QTSS    QUICKTIME STREAMING SERVER
POSTFIX POSTFIX USER
MYSQL   MYSQL SERVER
MAILMAN MAILMAN USER
LP      PRINTING SERVICES
JABBER  JABBER USER
EPPC    APPLE EVENTS USER
DAEMON  SYSTEM SERVICES
AMAVISD AMAVISD USER
$ ./usuarios-7.sh -s -d ,
amavisd,Amavisd User
daemon,System Services
eppc,Apple Events User
jabber,Jabber User
lp,Printing Services
mailman,Mailman user
mysql,MySQL Server
postfix,Postfix User
qtss,QuickTime Streaming Server
root,System Administrator
sshd,sshd Privilege separation
tokend,Token Daemon
unknown,Unknown User
uucp,Unix to Unix Copy Protocol
www,World Wide Web Server
$
```

Como (e quando) usar o getopts

Nas páginas anteriores vimos em detalhes como processar opções e argumentos informados pelo usuário na linha de comando. Vimos também que não existe um padrão, mas o uso da maioria indica um padrão estabelecido na prática. Tentando preencher estas duas lacunas (padronização e processamento), o Bash criou um comando interno (builtin) chamado de `getopts`.

O comando `getopts` serve para processar opções de linha de comando. Ele toma conta do procedimento de identificar e extrair cada opção informada, além de fazer as duas verificações básicas de erro: quando o usuário digita uma opção inválida e quando uma opção vem sem um argumento obrigatório.

Por ser rápido e lidar com a parte mais chata da tarefa de lidar com opções, o getopts pode ser uma boa escolha para programas simples que desejam usar apenas opções curtas. Porém, a falta de suporte às --opções-longas é sua maior limitação, impossibilitando seu uso em programas que precisam delas.

Prós e contras do getopts

getopts	"na mão"
Funciona somente no Bash	Funciona em qualquer shell
Somente opções curtas (`-h`)	Opções curtas e longas (`-h`, `--help`)
Opções curtas juntas ou separadas (`-abc` e `-a -b -c`)	Opções curtas somente separadas (`-a -b -c`)
Verificação automática de argumento nulo	Verificação de argumentos para opções é feita manualmente
As opções válidas devem ser registradas em dois lugares	As opções válidas são registradas somente em um lugar
Loop nas opções é feito automaticamente	Loop nas opções é feito manualmente com o `shift`

Se você já tem um programa e quer incluir rapidamente o suporte a algumas opções de linha de comando, comece com o getopts e opções curtas que é a solução mais rápida. Depois se você julgar importante incluir também as opções longas, mude para a solução caseira.

O `getopts` é um comando diferente, que leva um certo tempo para acostumar-se com os seus detalhes. Para começar, veremos um exemplo com muitos comentários, para que você tenha uma visão geral de seu uso.

Leia com atenção:

getopts-teste.sh

```
#!/bin/bash
# getopts-teste.sh
#
# Aurelio, Novembro de 2007

# Loop que processa todas as opções da linha de comando.
# Atenção ao formato das opções válidas ":sa:"
# - Os dois-pontos do início ligam o modo silencioso
# - As opções válidas são 'sa:', que são -s e -a
# - Os dois-pontos de 'a:' representam um argumento: -a FOO
#
while getopts ":sa:" opcao
do
    # $opcao guarda a opção da vez (ou ? e : em caso de erro)
    # $OPTARG guarda o argumento da opção (se houver)
    #
    case $opcao in
         s) echo "OK Opção simples (-s)";;
         a) echo "OK Opção com argumento (-a), recebeu: $OPTARG";;
        \?) echo "ERRO Opção inválida: $OPTARG";;
         :) echo "ERRO Faltou argumento para: $OPTARG";;
    esac
done

# O loop termina quando nenhuma opção for encontrada.
# Mas ainda podem existir argumentos, como um nome de arquivo.
# A variável $OPTIND guarda o índice do resto da linha de
# comando, útil para arrancar as opções já processadas.
#
echo
shift $((OPTIND - 1))
echo "INDICE: $OPTIND"
echo "RESTO: $*"
echo
```

Antes de entrarmos nos detalhes do código, veja alguns exemplos de sua execução, que ajudarão a compreender o que o `getopts` faz. No primeiro

exemplo foram passadas duas opções (uma simples e outra com argumento) e um argumento solto, que é um nome de arquivo qualquer. No segundo exemplo temos apenas o nome de arquivo, sem opções. Por fim, no terceiro fizemos tudo errado :)

```
$ ./getopts-teste.sh -a FOO -s /tmp/arquivo.txt
OK Opção com argumento (-a), recebeu: FOO
OK Opção simples (-s)
INDICE: 4
RESTO: /tmp/arquivo.txt
$ ./getopts-teste.sh /tmp/arquivo.txt
INDICE: 1
RESTO: /tmp/arquivo.txt
$ ./getopts-teste.sh -z -a
ERRO Opção inválida: z
ERRO Faltou argumento para: a
INDICE: 3
RESTO:
$
```

Entendeu? Funcionar ele funciona, mas tem uma série de detalhes que são importantes e devem ser respeitados. Estes são os passos necessário para usar o `getopts` corretamente:

- Colocar o `getopts` na chamada de um `while`, para que ele funcione em um loop.
- O primeiro argumento para o `getopts` é uma string que lista todas as opções válidas para o programa:
  - Se o primeiro caractere desta string for dois-pontos ":", o `getopts` funcionará em modo silencioso. Neste modo, as mensagens de erro não são mostradas na tela. Isso é muito útil para nós, pois elas são em inglês. Outra vantagem deste modo é no tratamento de erros, possibilitando diferenciar quando é uma opção inválida e quando falta um argumento. **Use sempre este modo silencioso**.
  - Liste as letras todas grudadas, sem espaços em branco. Lembre-se que o getopts só entende opções curtas, de uma letra. Por exemplo, "`hVs`" para permitir as opções `-h`, `-V` e `-s`.

    - Se uma opção requer argumento (como a `-d` do nosso `usuarios.sh`), coloque dois-pontos logo após. Por exempo, em "`hVd:sru`" somente a opção `-d` precisa receber argumento.
- O segundo argumento do `getopts` é o nome da variável na qual ele irá guardar o nome da opção atual, útil para usar no `case`. Aqui você pode usar o nome que preferir.
- Dentro do `case`, liste todas as opções válidas. Note que não é necessário o hífen! Se a opção requer um argumento, ele estará guardado na variável `$OPTARG`. Trate também as duas alternativas especiais que o `getopts` usa em caso de erro:
    - Uma interrogação é utilizada quando o usuário digita uma opção inválida. A opção digitada estará dentro de `$OPTARG`.
    - Os dois-pontos são utilizados quando faltou informar um argumento obrigatório para uma opção. A opção em questão estará dentro de `$OPTARG`.
- Nenhum `shift` é necessário, pois o `getopts` cuida de todo o processo de avaliação de cada opção da linha de comando.
- Por fim, se processadas todas as opções e ainda sobrar algum argumento, ele também pode ser obtido. Isso geralmente acontece quando o aplicativo aceita receber um nome de arquivo como último argumento, assim como o `grep`. A variável `$OPTIND` guarda o índice com a posição deste argumento. Como podem ser vários argumentos restantes, o mais fácil é usar um `shift N` para apagar todas as opções, sobrando apenas os argumentos. Onde `N = $OPTIND - 1`.

Como você percebeu, são vários detalhes, o `getopts` não é um comando trivial. Mas você se acostuma. E usado uma vez, você pode aproveitar a mesma estrutura para outro programa. Analise o exemplo que acabamos de ver, digite-o, faça seus testes. O `getopts` facilita sua vida, mas primeiro você deve ficar mais íntimo dele. Segue uma tabelinha bacana que tenta resumir todos estes detalhes:

Resumo das regras do getopts

```
while getopts OPÇÕES VAR; do ... done
```

OPÇÕES (string definida pelo programador)	
a	Opção simples: -a
a:	Opção com argumento: -a FOO
:	Liga modo silencioso (se for o primeiro caractere)
$VAR (gravada automaticamente em cada laço do loop)	
a-z	Opção que está sendo processada agora
?	Opção inválida ou falta argumento (modo normal) Opção inválida (modo silencioso)
:	Falta argumento (modo silencioso)
Variáveis de ambiente	
$OPTERR	Liga/desliga as mensagens de erro (o padrão é OPTERR=1)
$OPTARG	Guarda o argumento da opção atual (se houver)
$OPTIND	Guarda o índice do resto da linha de comando (não-opções)

O usuarios.sh, que estudamos bastante, pode ser modificado para funcionar usando o getopts. A desvantagem é que somente as opções curtas podem ser utilizadas. O interessante aqui é ver o diff das duas versões, para ficar bem fácil enxergar as diferenças das duas técnicas:

usuarios.sh (diff da versão 7 para usar getopts)

```
--- usuarios-7.sh    2007-11-20 18:19:22.000000000 -0200
+++ usuarios-7-getopts.sh    2007-11-21 00:23:18.000000000 -0200
@@ -1,92 +1,88 @@
#!/bin/bash
# usuarios.sh
#
# Mostra os logins e nomes de usuários do sistema
# Obs.: Lê dados do arquivo /etc/passwd
#
# Versão 1: Mostra usuários e nomes separados por TAB
# Versão 2: Adicionado suporte à opção -h
# Versão 3: Adicionado suporte à opção -V e opções inválidas
# Versão 4: Arrumado bug quando não tem opções, basename no
#           nome do programa, -V extraindo direto dos cabeçalhos,
#           adicionadas opções --help e --version
# Versão 5: Adicionadas opções -s e --sort
# Versão 6: Adicionadas opções -r, --reverse, -u, --uppercase,
```

```
#           leitura de múltiplas opções (loop)
# Versão 7: Melhorias no código para que fique mais legível,
#           adicionadas opções -d e --delimiter
+# Versão 7g: Modificada para usar o getopts
#
# Aurelio, Novembro de 2007
#
ordenar=0       # A saída deverá ser ordenada?
inverter=0      # A saída deverá ser invertida?
maiusculas=0    # A saída deverá ser em maiúsculas?
delim='\t'      # Caractere usado como delimitador de saída
MENSAGEM_USO="
Uso: $(basename "$0") [OPÇÕES]
OPÇÕES:
-  -d, --delimiter C  Usa o caractere C como delimitador
-  -r, --reverse      Inverte a listagem
-  -s, --sort         Ordena a listagem alfabeticamente
-  -u, --uppercase    Mostra a listagem em MAIÚSCULAS
+  -d C     Usa o caractere C como delimitador
+  -r       Inverte a listagem
+  -s       Ordena a listagem alfabeticamente
+  -u       Mostra a listagem em MAIÚSCULAS
-  -h, --help         Mostra esta tela de ajuda e sai
-  -V, --version      Mostra a versão do programa e sai
+  -h       Mostra esta tela de ajuda e sai
+  -V       Mostra a versão do programa e sai
"
# Tratamento das opções de linha de comando
-while test -n "$1"
+while getopts ":hVd:rsu" opcao
do
-    case "$1" in
+    case "$opcao" in
        # Opções que ligam/desligam chaves
-       -s | --sort     ) ordenar=1    ;;
-       -r | --reverse  ) inverter=1   ;;
-       -u | --uppercase) maiusculas=1 ;;
-
-       -d | --delimiter)
```

```
-            shift
-            delim="$1"
-
-            if test -z "$delim"
-            then
-                echo "Faltou o argumento para a -d"
-                exit 1
-            fi
+        s) ordenar=1    ;;
+        r) inverter=1   ;;
+        u) maiusculas=1 ;;
+
+        d)
+            delim="$OPTARG"
        ;;

-        -h | --help)
+        h)
            echo "$MENSAGEM_USO"
            exit 0
        ;;
-        -V | --version)
+        V)
            echo -n $(basename "$0")
            # Extrai a versão diretamente dos cabeçalhos do
  programa
            grep '^# Versão ' "$0" | tail -1 | cut -d : -f 1 | tr
  -d \#
            exit 0
        ;;
-        *)
-            echo Opção inválida: $1
+        \?)
+            echo Opção inválida: $OPTARG
+            exit 1
+        ;;
+
+        :)
+            echo Faltou argumento para: $OPTARG
```

```
            exit 1
        ;;
    esac
-
-    # Opção $1 já processada, a fila deve andar
-    shift
done
# Extrai a listagem
lista=$(cut -d : -f 1,5 /etc/passwd)
# Ordena, inverte ou converte para maiúsculas (se necessário)
test "$ordenar"    = 1 && lista=$(echo "$lista" | sort)
test "$inverter"   = 1 && lista=$(echo "$lista" | tac)
test "$maiusculas" = 1 && lista=$(echo "$lista" | tr a-z A-Z)
# Mostra o resultado para o usuário
echo "$lista" | tr : "$delim"
```

A mudança mais marcante é a eliminação do código de verificação do argumento para a opção `-d`. Ele já está pronto para ser usado em `$OPTARG`. Caso o argumento não tenha sido informado, o `case` cairá direto na alternativa ":", que mostra uma mensagem de erro genérica. O uso do `shift` também foi abolido. Fora isso, são apenas mudanças bem pequenas.

Para finalizarmos este capítulo, um exemplo da execução desta versão modificada, com a limitação de não poder usar opções longas, porém com a vantagem de poder juntar várias opções curtas em um mesmo argumento `-sru`.

```
$ ./usuarios-7-getopts.sh -h
Uso: usuarios-7-getopts.sh [OPÇÕES]
OPÇÕES:
  -d C    Usa o caractere C como delimitador
  -r      Inverte a listagem
  -s      Ordena a listagem alfabeticamente
  -u      Mostra a listagem em MAIÚSCULAS
  -h      Mostra esta tela de ajuda e sai
  -V      Mostra a versão do programa e sai
$ ./usuarios-7-getopts.sh -V
usuarios-7-getopts.sh Versão 7g
$ ./usuarios-7-getopts.sh -sru -d ,
```

```
WWW,WORLD WIDE WEB SERVER
UUCP,UNIX TO UNIX COPY PROTOCOL
UNKNOWN,UNKNOWN USER
TOKEND,TOKEN DAEMON
SSHD,SSHD PRIVILEGE SEPARATION
ROOT,SYSTEM ADMINISTRATOR
QTSS,QUICKTIME STREAMING SERVER
POSTFIX,POSTFIX USER
MYSQL,MYSQL SERVER
MAILMAN,MAILMAN USER
LP,PRINTING SERVICES
JABBER,JABBER USER
EPPC,APPLE EVENTS USER
DAEMON,SYSTEM SERVICES
AMAVISD,AMAVISD USER
$
```

Capítulo 5

Depuração (debug)

À medida que os programas crescem e sua complexidade aumenta, começa a ficar difícil memorizar todo o funcionamento. Quando aparece um problema, pode demorar até encontrar o ponto exato onde ele acontece. Aprenda a usar as técnicas de depuração para monitorar tudo o que acontece durante a execução do seu programa. Veja quais comandos foram executados, inspecione o conteúdo de variáveis e estados de chaves. Levante o capô do carro e veja como tudo funciona.

Depurar é purificar, limpar. Em nosso contexto, depurar é resolver problemas (bugs) e melhorar a performance de um código. Para isso, o programa é colocado em modo de depuração, também chamado de modo debug. Também é comum dizer-se que é preciso fazer um debug ou, ainda mais simples, debugar.

A depuração é necessária quando há um problema cuja causa é muito difícil de encontrar durante a execução normal do programa ou na pura inspeção visual do código. Nestes casos, é preciso uma visão de raio X para ver o que acontece por trás das cortinas durante a execução e, assim, encontrar o trecho problemático.

Usando algumas técnicas que veremos a seguir, ao rodar o programa, é possível saber exatamente em qual ponto do código ele se encontra, vendo inclusive o conteúdo das variáveis naquele momento. Sabe quando o médico abre o peito do paciente em uma cirurgia e é possível ver o coração ali batendo? É isso :)

Em shell, temos várias maneiras de depurar programas, desde o debug simples de colocar echos em pontos estratégicos até a execução passo a passo e a depuração personalizada com funções. Veja a coleção de técnicas que aprenderemos nos parágrafos seguintes:

- Verificação de sintaxe.
- Execução passo a passo.
- Debug simples.
- Debug global.
- Debug setorizado.
- Debug personalizado.
- Debug categorizado.

Mas, antes de começar, primeiro precisamos de uma cobaia. Não adianta querer depurar um programa funcional, precisamos de um código defeituoso para poder analisá-lo e encontrar o ponto de falha:

⌨ grita.sh (com defeito)

```
#!/bin/bash
# grita.sh
#
# Mostra uma palavra ($TXT) em maiúsculas e com exclamações
# Exemplo: foo -> !!!!!FOO!!!!!

TXT="gritaria"
TXT="     $TXT     "            # Adiciona 5 espaços ao redor
TXT=$(echo $TXT | tr ' ' '!')   # Troca os espaços por exclamações
TXT=$(echo $TXT | tr a-z A-Z)   # Deixa o texto em maiúsculas
echo "$TXT"                     # Mostra a mensagem
```

O código é bem simples e aparentemente está correto. O conteúdo da variável `$TXT` é manipulado para adicionar exclamações ao redor e deixar todas as letras em maiúsculas. O programa deveria mostrar a mensagem "!!!!!GRITARIA!!!!!", mas ao rodá-lo não é exatamente isso o que acontece:

```
$ ./grita.sh
GRITARIA
$
```

Ué, para onde foram as exclamações que o `tr` colocou? Quem as removeu? Ou será que nem foram de fato colocadas? O código aparentemente está certo, como saber o que aconteceu de errado? Entra em cena a depuração. É preciso ver o que acontece com a variável `$TXT` durante a execução do programa para identificar o problema.

Como diria o Lion dos Thundercats: "Espada justiceira, dê-me a visão além do alcance!"

Verificação de sintaxe (-n)

Ao encontrar algum problema, antes de começar todo o processo de depuração e escolher qual técnica utilizar, é prudente primeiro verificar o básico: está tudo certo com a sintaxe dos comandos? Será que de repente não falta fechar um parênteses ou colocar um ponto e vírgula em algum lugar?

Sabe quando o computador não liga e a pessoa ao lado pergunta: "Está ligado na tomada?". Você tem vontade de massacrar o indivíduo por perguntar algo tão óbvio, mas a raiva vira vergonha quando você olha para a tomada e... Pois é,

acontece.

O shell tem uma opção de linha de comando chamada `-n`, que serve para testar se há erros de sintaxe em seu código. Útil para programas de execução demorada como rotinas de becape. Todo o código é testado, mas nada é executado, então o resultado é instantâneo.

```
$ bash -n grita.sh
$
```

Caso algum erro seja encontrado, será mostrada uma mensagem na tela, informando o número da linha e o problema com ela. Se nada aparecer e o prompt voltar, isso significa que seu programa está com a sintaxe correta (*No news are good news*). Em nosso caso, não há erros de sintaxe no `grita.sh`.

Debug simples (echo)

A maneira mais simples de se depurar um programa é colocar comandos adicionais para mostrar o conteúdo de variáveis importantes no trecho de código problemático. Esta técnica não é exclusiva do shell e pode ser usada em qualquer linguagem de programação.

O problema de nosso programa é que as exclamações não estão aparecendo no resultado. Precisamos usar o debug simples ao redor da linha responsável por colocar as exclamações no texto, ela é a suspeita. Basta colocar um `echo` antes e outro depois, mostrando o conteúdo da variável para analisarmos o que acontece:

grita.sh (debug simples)

```
#!/bin/bash
# grita.sh
#
# Mostra uma palavra ($TXT) em maiúsculas e com exclamações
# Exemplo: foo -> !!!!!FOO!!!!!
TXT="gritaria"
TXT="     $TXT     "            # Adiciona 5 espaços ao redor
echo "TXT com espaços    : [$TXT]"
TXT=$(echo $TXT | tr ' ' '!')   # Troca os espaços por exclamações
echo "TXT com exclamações: [$TXT]"
```

```
TXT=$(echo $TXT | tr a-z A-Z)   # Deixa o texto em maiúsculas
echo "$TXT"                     # Mostra a mensagem
```

É só isso mesmo, simples. O que precisamos ver é o conteúdo da variável `$TXT`. Mas em vez de colocar somente `echo $TXT`, foi adicionado também um texto identificando o que era esperado encontrar naquela variável. Neste exemplo, isso pode parecer desnecessário, mas, à medida que você for colocando mais echos de debug, essa identificação ajudará muito na análise da saída. Vejamos:

```
$ ./grita.sh
TXT com espaços    : [     gritaria     ]
TXT com exclamações: [gritaria]
GRITARIA
$
```

No primeiro `echo`, a variável `$TXT` estava com os espaços ao redor (perceba a importância dos colchetes colocados para podermos enxergar estes espaços). Porém, após o comando que deveria trocar estes espaços por exclamações, não temos mais nada, nem exclamações nem espaços! Encontramos onde está o problema.

Com a depuração simples, colocamos apenas dois echos ao redor da linha suspeita e tivemos a confirmação: aquele é o ponto exato do código onde o erro acontece. Mas ainda não é possível saber ao certo qual o erro, pois aparentemente a linha está correta. Precisamos de uma depuração mais detalhada.

Debug global (-x, -v)

Se o debug simples não foi suficiente, você precisa de uma depuração mais agressiva. Algo que mostre o que acontece a cada comando executado, mesmo quando há vários comandos em uma mesma linha, unidos por um redirecionamento (pipe) ou agrupados em uma subshell. Conheça a opção `-x`.

Chamado com a opção `-x`, o shell mostra na tela os comandos conforme eles são executados, após feitas todas as expansões internas de nomes de arquivo e variáveis. Estas linhas especiais são precedidas pelo sinal de mais "+", ou por qualquer outro caractere que esteja armazenado na

variável de ambiente `$PS4`. Dois sinais de mais "++" indicam que o comando foi executado em uma subshell. Quanto mais fundo, mais sinais.

```
$ bash -x grita.sh
+ TXT=gritaria
+ TXT=     gritaria
++ echo gritaria     <------ aqui
++ tr ' ' '!'
+ TXT=gritaria
++ echo gritaria
++ tr a-z A-Z
+ TXT=GRITARIA
+ echo GRITARIA
GRITARIA
$
```

Pelas informações adicionais podemos perceber que, antes do primeiro comando `tr`, o `echo` mostrou apenas "gritaria", sem os espaços em branco ao redor. Como o `tr` não recebeu os espaços, ele não pôde trocá-los pelas exclamações. Legal, estamos isolando o problema com a ajuda da depuração. Agora é uma boa hora para testar o trecho defeituoso na linha de comando e descobrimos de vez o que causa este erro:

```
$ TXT="gritaria"
$ TXT="     $TXT     "
$ echo $TXT | tr ' ' '!'       # é, nada ainda...
gritaria
$ echo $TXT                   # cadê os espaços?
gritaria
$ echo "$TXT"                   # ah!!! malditas aspas!
     gritaria
$ echo "$TXT" | tr ' ' '!'     # foi
!!!!!gritaria!!!!!
$
```

Ah, as aspas... Sempre elas! Se não colocar aspas ao redor da variável, `"$assim"`, o shell faz uma limpa em seu conteúdo: os espaços em branco do início e do final são cortados e os espaços consecutivos internos são reduzidos a apenas um. Veja mais um exemplo deste comportamento:

```
$ TXT='     um     dois     três     '
$ echo $TXT
um dois três
$ echo "$TXT"
     um     dois     três
$ echo '$TXT'
$TXT
$
```

O último comando pode ser ignorado. Foi só para você lembrar que dentro das aspas simples (não duplas) as variáveis não são expandidas :)

Este comportamento do shell é útil quando você realmente deseja ignorar todos os espaços em branco de uma string, como no caso de fazer um loop nas palavras de um texto. Tanto faz a quantidade de espaços, o que você quer mesmo são as palavras. Então você fará `for palavra in $TXT; do ... done`, sem as aspas. Mas como esta é uma exceção e na prática geralmente queremos o conteúdo real de nossa variável, acostume-se a **sempre** usar aspas. Nem pense se é necessário ou não, use sempre.

Sempre use "aspas" ao redor das variáveis. Isso evitará problemas em seus programas.

Voltando ao nosso programa-encrenca, a correção é simples, basta colocar aspas ao redor da variável `$TXT` e tudo vai funcionar corretamente. A linha problemática vai então ficar assim:

```
TXT=$(echo "$TXT" | tr ' ' '!') # Troca os espaços por exclamações
```

Com a certeza de que agora tudo irá funcionar corretamente, podemos rodar o programa corrigido com um sorriso discreto de satisfação no rosto, apertando o Enter do teclado com vontade, batendo forte a ponta do dedo, fazendo aquele barulho seco que ecoa pela sala:

```
$ bash -x grita.sh
+ TXT=gritaria
+ TXT=     gritaria
++ echo '     gritaria     '
++ tr ' ' '!'
+ TXT=!!!!!gritaria!!!!!
++ echo '!!!!!gritaria!!!!!'
++ tr a-z A-Z
+ TXT=!!!!!GRITARIA!!!!!
```

```
+ echo '!!!!!GRITARIA!!!!!'
!!!!!GRITARIA!!!!!
$
```

Outro recurso útil para se usar é a opção `-v`, que mostra a linha atual do programa, a que está sendo executada naquele momento. Em programas muitos extensos, onde é fácil perder-se no meio do código, é muito útil depurar usando o `-v`. Vamos usar um outro arquivo de exemplo para facilitar:

⌨ cinco.sh

```
#!/bin/bash
# cinco.sh
#
# Conta até cinco :)
echo $((0+1))
echo $((0+2))
echo $((0+3))
echo $((0+4))
echo $((0+5))
```

Nenhum segredo. Os cálculos só foram usados para que fique fácil de você perceber a diferença entre a linha original `echo $((0+1))` e a linha executada pelo shell `echo 1`, que, por sua vez, é diferente da linha mostrada na tela: `1`. Acompanhe a execução deste programa utilizando as opções de depuração:

```
$ bash cinco.sh      # sem depuração
1
2
3
4
5
$ bash -x cinco.sh   # mostra os comandos
+ echo 1
1
+ echo 2
2
+ echo 3
3
```

```
+ echo 4
4
+ echo 5
5
$ bash -v cinco.sh   # mostra as linhas do código
#!/bin/bash
# cinco.sh
#
# Conta até cinco :)
echo $((0+1))
1
echo $((0+2))
2
echo $((0+3))
3
echo $((0+4))
4
echo $((0+5))
5
```

Percebeu a diferença? Com a opção `-x` os comandos são mostrados na tela no momento em que são executados, precedidos pelo sinal de mais "+". Logo em seguida vem a saída normal do programa, mostrando o resultado do comando. Por isso, a saída fica intercalada entre debug e resultado, para cada linha. Já o `-v` mostra a linha original do código, sem prefixo, seguida pelo resultado.

Existem situações nas quais será mais eficiente usar o `-x` para entender como os comandos foram executados. Em outras é mais prático simplesmente saber qual a linha original para saber em que ponto do código estamos. Mas o melhor é que você pode usar as duas opções ao mesmo tempo, se assim desejar:

```
$ bash -xv cinco.sh
#!/bin/bash
# cinco.sh
#
# Conta até cinco :)
echo $((0+1))
+ echo 1
```

```
1
echo $((0+2))
+ echo 2
2
echo $((0+3))
+ echo 3
3
echo $((0+4))
+ echo 4
4
echo $((0+5))
+ echo 5
5
$
```

Pode parecer confuso ter tantas informações na tela, mas quando for o seu programa que estiver com um bug cabeludo, o chefe o pressionando, o cliente pressionando seu chefe e todos estiverem de cabeça quente, essa sopa de letrinhas será muito bem-vinda!

Debug setorizado (liga/desliga)

O debug global do tópico anterior é uma técnica eficiente para desvendar todos os segredos da execução do seu programa. Mas esta eficiência cai à medida que aumenta o número de linhas do programa. Quanto maior o programa, mais difícil fica encontrar as informações relevantes em meio a tantas linhas de depuração. O ideal nestes casos é ligar o debug apenas para algumas partes do programa, aquelas onde você desconfia que se encontra o problema.

O shell lhe dá a opção de ligar e desligar o debug em qualquer ponto do código, usando o comando `set`. Basta colocá-lo antes do trecho desejado, com a opção `-x` (ou `-v`) para ligar o debug. Todos os comandos posteriores serão executados com mensagens de depuração. Para desligar o debug a qualquer momento, basta usar o set mais uma vez, porém com a opção `+x`. Isso mesmo, o sinal de mais desliga a opção.

Comando	Descrição
`set -x`	Liga o modo depuração de comandos.

`set +x`	Desliga o modo de depuração de comandos.
`set -v`	Liga o modo depuração de linhas.
`set +v`	Desliga o modo depuração de linhas.

Vamos brincar um pouco na linha de comando, ligando e desligando o modo de depuração para ver a diferença. Lembre-se que tanto faz usar estes comandos no console ou dentro do seu programa, o resultado é o mesmo:

```
$ set -x              # liga o debug
$ i=$((3+4))
+ i=7
$ date | cut -c1-3    # note como cada comando é mostrado
  isoladamente
+ date
+ cut -c1-3
Thu
$ echo $(echo $(echo $(echo oi))) # subshells! um + para
  cada nível
+++++ echo oi
++++ echo oi
+++ echo oi
++ echo oi
+ echo oi
oi
$ set +x              # desliga o debug
+ set +x
$ echo $(echo $(echo $(echo oi))) # agora não tem
  informações extras
oi
$
```

Agora vamos aplicar estes comandos no nosso `cinco.sh`, delimitando a área de depuração para apenas uma linha do programa. Vamos aproveitar e incluir a opção `-v` para que a linha original do código também seja mostrada na saída.

⌨ cinco.sh (debug setorizado)

```
#!/bin/bash
# cinco.sh
```

```
#
# Conta até cinco :)
echo $((0+1))
echo $((0+2))
set -xv       # liga debug
echo $((0+3))
set +xv       # desliga debug
echo $((0+4))
echo $((0+5))
```

Acompanhe na execução como apenas a terceira linha é dotada de informações de debug, tanto do comando (`-x`) quanto da linha original (`-v`). Observe que também aparece a linha que desliga o debug, pois ela é primeiro mostrada, e somente então executada.

```
$ ./cinco.sh
1
2
echo $((0+3))
+ echo 3
3
set +xv       # desliga debug
+ set +xv
4
5
$
```

Os comandos de debug podem ser colocados em qualquer ponto do programa, inclusive podem haver vários pares de liga/desliga no código, revelando somente as informações dos trechos mais críticos. Veja este exemplo, com trechos diferentes para debug de comandos e de linhas:

⌨ cinco.sh (debug setorizado múltiplo)

```
#!/bin/bash
# cinco.sh
#
# Conta até cinco :)
set -v        # liga debug de linhas
echo $((0+1))
```

```
echo $((0+2))
set +v        # desliga debug de linhas
echo $((0+3))
set -x        # liga debug de comandos
echo $((0+4))
echo $((0+5))
set +x        # desliga debug de comandos
```

Na execução do programa fica fácil perceber que aparecem as duas primeiras linhas, então o debug é desligado. A terceira linha é executada normalmente e a quarta e a quinta mostram o debug de comandos:

```
$ ./cinco.sh
echo $((0+1))
1
echo $((0+2))
2
set +v    # desliga debug de linhas
3
+ echo 4
4
+ echo 5
5
+ set +x
$
```

Execução passo a passo

O shell não possui as facilidades de um depurador poderoso, que permite criar breakpoints e a lhe dá a possibilidade de "pular" entre trechos específicos durante a execução. Porém, uma alternativa que pode ajudar quando não se sabe exatamente em qual ponto o programa capota, é fazer uma execução pausada, onde a próxima linha só é executada quando o usuário pressiona a tecla Enter. Assim a execução caminha passo a passo, no ritmo que o usuário desejar.

```
trap read DEBUG
```

Basta colocar esta linha mágica no programa para, daquele ponto em diante, condicionar a execução à resposta do usuário. Note, porém, que é

aconselhável o modo de depuração estar ligado para que as mensagens sejam mostradas na tela e o usuário saiba em que ponto está. Use o `set -x` ou `-v`, qual preferir. Para desligar a qualquer momento este modo assistido, coloque outra linha mágica:

```
trap "" DEBUG
```

Mais uma vez o `cinco.sh` será alterado, agora ligando o debug e o passo a passo já no início. Após o segundo comando, o passo a passo é desligado, porém o debug continua até o final. Note que não é preciso desligá-lo com o `set +x`, quando o programa terminar, o modo de depuração é encerrado também.

⌨ cinco.sh (debug passo a passo)

```
#!/bin/bash
# cinco.sh
#
# Conta até cinco :)
set -x           # liga debug
trap read DEBUG  # liga passo a passo
echo $((0+1))
echo $((0+2))
trap "" DEBUG    # desliga passo a passo
echo $((0+3))
echo $((0+4))
echo $((0+5))
```

Durante a execução o programa para e espera pelo Enter do usuário. Os pontos de parada estão indicados pelo comando `read`, seguido da linha em branco, causada pela tecla do usuário. Desligado o trap, do terceiro comando em diante o programa continua sem pausas até o final:

```
$ ./cinco.sh
+ trap read DEBUG
++ read

+ echo 1
1
++ read

+ echo 2
```

```
2
++ read
+ trap '' DEBUG
+ echo 3
3
+ echo 4
4
+ echo 5
5
$
```

Esta técnica do passo a passo não funciona no Bourne Shell (sh).

Debug personalizado

Depois que o programa atinge um certo nível de complexidade, mostrar a depuração para todas as linhas fica pouco eficiente, pois a informação desejada perde-se em um mar de linhas de depuração. A técnica de debug simples com o `echo` também se torna incômoda por precisar comentar e descomentar várias linhas de debug cada vez que precisar testar algo. Neste estágio, a solução é criar uma função específica para cuidar da depuração:

```
Debug(){
    [ "$DEBUG" = 1 ] && echo "$*"
}
```

A função `Debug()` é uma evolução do debug simples com o `echo`. Continua sendo um `echo`, porém ele só vai mostrar a mensagem de depuração quando a variável global `$DEBUG` estiver definida com o valor 1. Assim, trocando o valor de uma única variável, o programador pode ligar e desligar a depuração de todo o programa. Quer depurar? Faça `DEBUG=1` e pronto. Não quer mais? Comente a linha ou mude o valor para zero. Chaves, você sabe.

Voltemos ao primeiro exemplo deste capítulo, o `grita.sh`, e vamos implementar o debug personalizado nele. Não é nada complicado, basta definir a variável `$DEBUG` no início do programa, declarar a função `Debug()` e utilizá-la no código, no lugar dos echos. Veja:

⌨ grita.sh (debug personalizado)

```
#!/bin/bash
# grita.sh
#
# Mostra uma palavra ($TXT) em maiúsculas e com exclamações
# Exemplo: foo -> !!!!!FOO!!!!!
DEBUG=1       # depuração: 0 desliga, 1 liga
# Função de depuração
Debug(){
    [ "$DEBUG" = 1 ] && echo "$*"
}
TXT="gritaria"
TXT="     $TXT     "    # Adiciona 5 espaços ao redor
Debug "TXT com espaços    : [$TXT]"
TXT=$(echo $TXT | tr ' ' '!')  # Troca os espaços por exclamações
Debug "TXT com exclamações: [$TXT]"
TXT=$(echo $TXT | tr a-z A-Z)  # Deixa o texto em maiúsculas
echo "$TXT"                    # Mostra a mensagem
```

O conceito é simples, a implementação é simples e é superútil. Mas pode ficar ainda melhor. Percebeu que o nome é debug "personalizado"? Então, com as mensagens de depuração centralizadas em uma função, é possível formatá-las de maneira uniforme, como, por exemplo, adicionar um prefixo padrão para que elas fiquem bem visíveis durante a execução:

```
Debug(){
    [ "$DEBUG" = 1 ] && echo "------------{ $*"
}
```

Veja como fica fácil identificar as mensagens de depuração com este prefixo que chama bastante a atenção:

```
$ ./grita.sh
------------{ TXT com espaços    : [     gritaria     ]
------------{ TXT com exclamações: [gritaria]
GRITARIA
$
```

Outra opção bem bacana é mostrar as mensagens de debug em uma cor diferente, como vermelho ou amarelo. Isso é possível usando caracteres de

controle, um tema que estudaremos em um capítulo seguinte. Por enquanto não precisa preocupar-se em entender o que são estes caracteres, apenas registre em sua mente que debug com mensagens coloridas é muito prático!

```
# Mostra as mensagens de depuração em amarelo
Debug(){
    [ "$DEBUG" = 1 ] && echo -e "\033[33;1m$*\033[m"
}
```

Debug categorizado

Neste momento você já é um expert em depuração, já tem a sua função `Debug()` toda turbinada e há vários pontos de depuração no programa. Mas você quer mais. Já há tantas mensagens de depuração que está ficando confuso novamente. Seria bom poder categorizar essas mensagens e mostrar apenas algumas, e não todas.

Uma ideia é utilizar números para identificar o tipo da mensagem. Cada número representaria um nível de debug, sendo que quanto maior o número, mais detalhada será a depuração. Mensagens de nível 1 seriam bem genéricas como "Vou iniciar o cálculo do total", enquanto mensagens de nível 4 trariam detalhes mais específicos como o conteúdo de variáveis secundárias do programa.

Cada programador deve fazer sua própria divisão conforme o código em que está trabalhando, porém segue uma sugestão que pode ser usada como ponto de partida:

Nível	Descrição
1	Mensagens genéricas, de localização ("estou aqui").
2	Mensagens de localização de fluxo ("entrei no loop").
3	Mensagens com conteúdo de variáveis importantes.
4	Mensagens com conteúdo de variáveis secundárias.

Assim, se a variável `$DEBUG` for configurada com o valor 2, serão mostradas as mensagens de localização genérica e localização de fluxo (níveis 1 e 2). Se o valor da variável for 4, as mensagens de todos os níveis serão mostradas. Partindo de mensagens genéricas para específicas,

quanto maior for o valor de `$DEBUG`, mais mensagens de depuração aparecerão na tela.

E sabe qual a melhor parte desse esquema numérico? É que a nossa função `Debug()` só precisa de uma modificação bem pequena para que tudo funcione. Em vez de verificar se o valor de `$DEBUG` é 1 ou 0, agora ela verifica se a mensagem recebida está em um nível menor ou igual (-le) ao valor de `$DEBUG`.

```
Debug(){
    [ $1 -le $DEBUG ] && echo "--- DEBUG $*"
}
```

Assim, se `DEBUG=3`, somente as mensagens de nível 4 não serão mostradas. No restante do código do programa, basta colocar o nível correto em todas as chamadas da função de debug. O nível deve ser o primeiro parâmetro, veja alguns exemplos:

```
Debug 1 Mensagem nível um
Debug 2 Mensagem nível dois
Debug 3 Mensagem nível três
Debug 4 Mensagem nível quatro
```

Para ilustrar este conceito, segue um programa que conta até cinco, porém possui diversas mensagens de debug que são ativadas pelo argumento da linha de comando. Se nada for passado, não mostra as mensagens. Se um número for passado, mostra as mensagens que se encaixam naquele nível.

debug-categorizado.sh

```
#!/bin/bash
# debug-categorizado.sh
#
# Exemplo de Debug categorizado em três níveis
DEBUG=${1:-0}    # passe o nível pelo $1
Debug(){
    [ $1 -le $DEBUG ] && echo "--- DEBUG $*"
}
Debug 1 "Início do Programa"
i=0
```

```
max=5
echo "Contando até $max"
Debug 2 "Vou entrar no WHILE"
while [ $i -ne $max ]; do
    Debug 3 "Valor de \$i antes  de incrementar: $i"
    let i=$i+1
    Debug 3 "Valor de \$i depois de incrementar: $i"
    echo "$i..."
done
Debug 2 "Saí do WHILE"
echo 'Terminei!'
Debug 1 "Fim do Programa"
```

O código é simples, atente para as mensagens de debug com os níveis no início e a declaração de `$DEBUG` que pega o número informado pelo usuário na linha de comando (`$1`). Vejamos como este programa se comporta, cada vez sendo executado com um nível de debug diferente:

```
$ ./debug-categorizado.sh
Contando até 5
1...
2...
3...
4...
5...
Terminei!
$ ./debug-categorizado.sh 1
--- DEBUG 1 Início do Programa
Contando até 5
1...
2...
3...
4...
5...
Terminei!
--- DEBUG 1 Fim do Programa
$ ./debug-categorizado.sh 2
--- DEBUG 1 Início do Programa
Contando até 5
--- DEBUG 2 Vou entrar no WHILE
```

```
1...
2...
3...
4...
5...
--- DEBUG 2 Saí do WHILE
Terminei!
--- DEBUG 1 Fim do Programa
$ ./debug-categorizado.sh 3
--- DEBUG 1 Início do Programa
Contando até 5
--- DEBUG 2 Vou entrar no WHILE
--- DEBUG 3 Valor de $i antes  de incrementar: 0
--- DEBUG 3 Valor de $i depois de incrementar: 1
1...
--- DEBUG 3 Valor de $i antes  de incrementar: 1
--- DEBUG 3 Valor de $i depois de incrementar: 2
2...
--- DEBUG 3 Valor de $i antes  de incrementar: 2
--- DEBUG 3 Valor de $i depois de incrementar: 3
3...
--- DEBUG 3 Valor de $i antes  de incrementar: 3
--- DEBUG 3 Valor de $i depois de incrementar: 4
4...
--- DEBUG 3 Valor de $i antes  de incrementar: 4
--- DEBUG 3 Valor de $i depois de incrementar: 5
5...
--- DEBUG 2 Saí do WHILE
Terminei!
--- DEBUG 1 Fim do Programa
$
```

Tenha em mente que a função `Debug()` pode crescer tanto quanto se deseje. Conforme suas necessidades forem mudando, a função muda junto. Algumas das possibilidades de crescimento são:

- Gravar as mensagens em um arquivo para ser analisado posteriormente.
- Usar uma cor diferente para cada nível (muito útil!).

- Mudar o alinhamento da mensagem, deslocando o texto mais à direita quanto maior for seu nível.
- Fazer um pré-processamento que verifica outras condições, além do valor de `$DEBUG` para decidir se mostra a mensagem ou não.

Não quero estragar sua diversão nem podar sua criatividade, então segue apenas um esqueleto de como seria uma função de debug mais parruda:

```
Debug(){
    [ $1 -le $DEBUG ] || return
    local prefixo
    case "$1" in
         1) prefixo="-- ";;
        2) prefixo="---- ";;
        3) prefixo="------ ";;
        *) echo "Mensagem não categorizada: $*"; return;;
    esac
    shift
    echo $prefixo$*
}
```

Capítulo 6

Caracteres de controle

Você pode programar vários anos sem saber que os caracteres de controle existem. Mas uma vez descobertos, abrem as portas para inúmeras possibilidades de melhoria para seus programas. Aprenda a mostrar textos coloridos na tela, posicionar o cursor, fazer animações simples e até emitir som. Use estes recursos com sabedoria e eleve a experiência do usuário a um novo nível.

Desconhecidos por uns, temidos por outros, os caracteres de controle são peças essenciais para um sistema de computação poder posicionar e desenhar texto na tela. São eles que informam ao sistema a posição do cursor, além de alterarem propriedades do texto, como a cor da letra e do fundo.

Por exemplo, como mostrar um texto exatamente na décima linha da tela? Ou como desenhar uma caixa na tela colocando um texto qualquer dentro dela? Ou, ainda, como dar pulos com o cursor, reescrevendo sempre na mesma linha para fazer uma barra de progresso? Isso tudo é feito com caracteres de controle.

Com a liberdade de movimentos que se consegue após dominar estes caracteres especiais, o limite é a sua imaginação, pois virtualmente qualquer tipo de tela com caracteres pode ser desenhada: animações, caixas, jogos, interfaces, botões...

Os caracteres de controle são "sequências de escape", caracteres normais precedidos por um caractere Esc. Por exemplo, a sequência `ESC [ 2 J` limpa toda a tela, não importando a posição atual do cursor.

As sequências de escape devem ser enviadas diretamente à tela, então basta usar o `echo` ou o `printf`, que mandam o texto para a saída padrão. Experimente:

```
echo -ne '\033[2J'
```

O caractere Esc é obtido usando seu código octal 033 e a opção `-e` do `echo` serve para interpretar este código. Também é possível ecoar um Esc literal apertando Ctrl+V seguido de Esc, aparecerá um ^[na tela para representar o caractere. Mas como comandos com o Esc literal não podem ser copiados/colados, a notação em octal será a utilizada.

Também foi usada a opção `-n`, para que o `echo` não quebrasse a linha no final. Ela sempre deve ser usada ao ecoar caracteres de controle, pois a quebra de linha atrapalha as operações de posicionamento do cursor. Outra alternativa é usar o `printf`, que também não quebra a linha:

```
printf '\033[2J'
```

Como não precisamos das funções avançadas de formatação do `printf`,

o `echo -ne` será o comando utilizado nos exemplos. Todas as sequências começam com um ESC seguido de um colchete, então `ESC[` é o início padrão de todas as sequências de caracteres de controle que veremos.

Mostrando cores na tela

Para começar, nada de pulos com o cursor ou operações complicadas. A maior necessidade dos programadores que procuram os caracteres de controle é mostrar letras coloridas na tela, ou trocar a cor do fundo. A sequência para códigos de cor tem a seguinte sintaxe:

```
ESC [ n1 ; n2 ; ... m
```

Começa com o `ESC[` padrão e termina com a letra m. Entre os dois são colocados números separados por ponto e vírgula, que indicam as cores a serem utilizadas. Caso nenhum número seja informado (`ESC[m`), o zero é assumido.

Códigos das cores da sequência ESC[...m

Código	Descrição		Código	Descrição
0	Texto normal, sem cores		5	Pisca-pisca
1	Cor brilhante		7	Vídeo reverso (invertido)
30	Texto preto (ou cinza)		40	Fundo preto (ou cinza)
31	Texto vermelho		41	Fundo vermelho
32	Texto verde		42	Fundo verde
33	Texto marrom (ou amarelo)		43	Fundo marrom (ou amarelo)
34	Texto azul		44	Fundo azul
35	Texto roxo		45	Fundo roxo
36	Texto ciano		46	Fundo ciano
37	Texto cinza (ou branco)		47	Fundo cinza (ou branco)

Se você tem as Funções ZZ (www.funcoeszz.net) instaladas, use a função `zzcores` que mostra todas as combinações de cores do console.

Basta consultar esta tabela e pronto. Então, para mostrar uma mensagem usando letras vermelhas, basta fazer:

```
echo -e '\033[31m MENSAGEM IMPORTANTE!!! \033[m'
```

A primeira sequência é o `ESC[...m` usando a cor 31 (vermelho) e no final usa-se a sequência vazia `ESC[m` para voltar as cores ao normal, senão a tela continuará com cores vermelhas. Mas no fundo preto, as cores normais ficam meio apagadas, muito escuras. É preciso também colocar o atributo brilhante na primeira sequência, para facilitar a visualização:

```
echo -e '\033[31;1m MENSAGEM IMPORTANTE!!! \033[m'
```

E assim vão se adicionando os códigos desejados, separados entre si por ponto e vírgula. Em um exemplo mais elaborado, `44;31;1;5` define fundo azul com letra vermelha e brilhante e ainda tem o código 5 que indica texto que pisca:

```
echo -e '\033[44;31;1;5m fundo azul, letra vermelha \033[m'
```

O atributo de piscar não funciona em alguns tipos de terminal.

A cor brilhante geralmente é a mesma cor normal, porém mais clara: vermelho claro, verde claro, e assim vai. As exceções são o marrom que vira amarelo, o preto que vira cinza e o cinza que vira branco.

Posicionando o cursor

Agora que já sabemos como colorir um texto, vamos ver como colocá-lo exatamente onde queremos na tela. O formato padrão dos comandos de movimentação é
`ESC[<quantidade><comando>`. Vamos começar com os comandos de movimentação simples, os mais comuns:

Comandos de movimentação do cursor

Comando	Ação (o padrão é n=1 e m=1)
`ESC[`*n*`A`	Move o cursor *n* linhas para cima, na mesma coluna.
`ESC[`*n*`B`	Move o cursor *n* linhas para baixo, na mesma coluna.
`ESC[`*n*`C`	Move o cursor *n* colunas para a direita, na mesma linha.
`ESC[`*n*`D`	Move o cursor *n* colunas para a esquerda, na mesma linha.
`ESC[`*n*`E`	Move o cursor *n* linhas para baixo, na coluna 1.
`ESC[`*n*`F`	Move o cursor *n* linhas para cima, na coluna 1.
`ESC[`*n*`G`	Move o cursor para a coluna *n* da linha atual.

ESC[*n*;*m*H	Move o cursor para a coluna *m* da linha *n*.

É fácil, basta ecoar o comando e o cursor vai dar o pulo, por exemplo, ESC[5E pula para o começo da 5ª linha abaixo da posição atual.

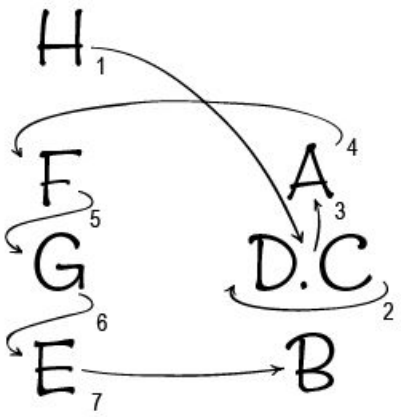
Para ficar mais visual o que cada comando faz, aqui vai um "gráfico" de exemplo de movimentação do cursor. Os comandos são executados com seus valores padrão (n=1,m=1) à partir da posição marcada pelo ponto, que está na linha 4 coluna 7:

```
  123456789
 +----------
1|H
2|
3|F      A
4|G    D.C
5|E     B
6|
```

Quer ver algo realmente bacana? Então execute o comando seguinte, que contém vários pulos de cursor, e no final de todo o vai-e-vem o resultado é... O gráfico de exemplo que acabamos de ver :)

```
echo -e
  '\033c\033[HH\033[4;7H.C\033[3DD\033[AA\033[GF\nG\nE\033[7GB'
```

Comando	Descrição
\033c	Limpa a tela. A sequência ESCc é igual a ESC[2J já vista.
\033[HH	Coloca a letra H na canto superior esquerdo da tela.
\033[4;7H.	Pula para a linha 4 coluna 7 e coloca o ponto.
C	Com o cursor após o ponto, aproveita para colocar a letra C.
\033[3DD	Volta 3 posições à esquerda para colocar a letra D.
\033[AA	Com o cursor sobre o ponto, sobe uma linha e coloca a letra A.
\033[GF	Pula para o início da linha e coloca a letra F.
\nG\nE	Usa o \n para descer linhas e coloca as letras G e E.
\033[7GB	Finalmente pula para a coluna 7 da mesma linha e coloca a letra B.



Sequência de pulos do cursor

Comandos de som

Também é possível alterar propriedades do bipe do computador (*beep*, *speaker*) com os caracteres de controle. Há dois comandos de som, um para definir a duração de um bipe e outro para definir a sua frequência.

Comandos de Som (Speaker)	
`ESC[10;`*n*`]`	Define a frequência para *n* (o padrão é 750).
`ESC[11;`*n*`]`	Define a duração para *n* milisegundos (o padrão é 100).

O caractere especial que emite um bipe é o Ctrl+G. Para digitá-lo faça Ctrl+V seguido de Ctrl+G, ele aparecerá na tela como ^G. Por exemplo, se quiser que cada bipada em sua máquina dure dois segundos, basta fazer:

```
$ echo -e '\033[11;2000]'   # Define a duração para 2000
  milisegundos
$ echo -e '^G'              # Emitindo o bipe
$ echo -e '\033[11;100]'    # Restaurando a duração padrão
```

Dependendo do seu sistema ou do tipo de terminal (`$TERM`), o bipe não será emitido. No Linux, mude para o modo texto (Ctrl+Alt+F1) caso no Xterm não esteja funcionando.

Em alguns sistemas, o escape `\a` também pode ser utilizado no lugar do `^G`.

Com criatividade e muita paciência é até possível fazer músicas com o `echo`, alterando a frequência e duração de cada "nota". Um exemplo simples, que simula uma sirene:

Sirene com caracteres de controle

```
#!/bin/bash
# sirene.sh
# Bom para chamar a atenção dos colegas de trabalho :)
echo -ne "\033[11;900]"   # Cada bipe dura quase um segundo
while :                   # Loop eterno
do
    echo -ne "\033[10;500]\a" ; sleep 1  # Tom alto (agudo)
    echo -ne "\033[10;400]\a" ; sleep 1  # Tom baixo (grave)
done
```

Outros comandos

Além de poder posicionar o cursor e mostrar caracteres coloridos na tela, os caracteres de controle ainda possuem outros comandos, que servem para:

- Apagar trechos da linha ou tela.
- Inserir espaços ou linhas em branco.
- Mover a tela.

Comandos de movimentação de tela e edição

Comando	Ação
ESC[0J	Apaga até o fim da tela.
ESC[1J	Apaga até o começo da tela.
ESC[2J	Apaga a tela toda.
ESC[0K	Apaga até o fim da linha.
ESC[1K	Apaga até o começo da linha.
ESC[2K	Apaga a linha toda.
Comando	**Ação (o padrão é n=1)**
ESC[*n*M	Apaga *n* linhas para baixo.
ESC[*n*P	Apaga *n* caracteres à direita.
ESC[*n*X	Limpa *n* caracteres à direita (coloca espaços).
ESC[*n*@	Insere *n* espaços em branco.
ESC[*n*L	Insere *n* linhas em branco.

ESC[*n*S	Move a tela *n* linhas para cima.
ESC[*n*T	Move a tela *n* linhas para baixo.

Antes de terminar, vale lembrar que alguns comandos como o `tput` ou bibliotecas como `ncurses` visam facilitar a tarefa de posicionamento de texto e desenho de caixas e botões, com funções já prontas para fazer isso. Mas se for assim, qual é a graça? :)

Exemplos

Paleta de cores

```
#!/bin/bash
# cores.sh
# Mostra todas as combinações de cores do console
for letra in 0 1 2 3 4 5 6 7; do      # LINHAS: cores das letras
  for brilho in '' ';1'; do           # liga/desliga cor brilhante
    for fundo in 0 1 2 3 4 5 6 7; do  # COLUNAS: cores dos fundos
      seq="4$fundo;3$letra"           # compõe código de cores
      echo -e "\033[$seq${brilho}m\c" # liga a cor
      echo -e " $seq${brilho:-  } \c" # mostra o código na tela
      echo -e "\033[m\c"              # desliga a cor
    done; echo                        # quebra a linha
  done
done
```

Barra de progresso

```
#!/bin/bash
# gauge.sh
# Barra de progresso usando caracteres de controle
#
#    [..................................................]  0%
#    [##########################........................] 50%
#    [##################################################] 100
#
# barra vazia
echo -n '[..................................................]  0%'
passo='#####'
for i in 10 20 30 40 50 60 70 80 90 100; do
```

```
    sleep 1
    pos=$((i/2-5))           # calcula a posição atual da barra
    echo -ne '\033[G'        # vai para o começo da linha
    echo -ne "\033[${pos}C"  # vai para a posição atual da barra
    echo -n  "$passo"        # preenche mais um passo
    echo -ne '\033[53G'      # vai para a posição da porcentagem
    echo -n  "$i"            # mostra a porcentagem
done
echo
```

⌨ Teste de som

```
#!/bin/bash
# som.sh
# Demonstração de mudança da frequência do Speaker
# Dica: Digite Ctrl+C para finalizar.
#
# Restaura o bipe padrão (f=750, t=100ms)
echo -e '\033[10;750]\033[11;100]'
freq=0                          # frequência inicial
while : ; do                    # loop infinito
  freq=$((freq+1))              # aumenta frequência
  echo -e "\033[10;$freq]"      # muda frequência no Speaker
  echo    "frequência=$freq"    # mostra frequência atual
  echo -e '\a'                  # emite um bipe
  usleep 100                    # espera 100us
done
```

⌨ Encaixotador de texto

```
#!/bin/bash
# caixa.sh
# Encaixota um texto qualquer vindo da STDIN
#
# Usando caracteres de controle, primeiro desenha uma caixa:
#             +----------+
#             |          |
#             |          |
#             +----------+
#
# Depois coloca o texto recebido via STDIN dentro dessa caixa:
```

```
#             +----------+
#             | O texto  |
#             | recebido |
#             +----------+
#
# A separação do código em dois passos permite personalizar
# separadamente a caixa e o texto, podendo-se facilmente
# adicionar cores ou fazer outras formatações.
#
# Configuração do usuário
caixa_largura=60
caixa_coluna_inicio=5
caixa_linha_inicio=5
texto_max_linhas=20
distancia_borda_texto=1
caixa_cor='33;1'
texto_cor='32'
#caixa_cor='33;43'      # descomente essa linha para uma surpresa!
#-------------------------------------------------------------------
  ----
# Daqui para baixo não mexa
### Configuração Dinâmica
caixa_coluna_fim=$((   caixa_coluna_inicio+caixa_largura-1))
texto_coluna_inicio=$((caixa_coluna_inicio+distancia_borda_texto+1
  ))
texto_largura=$((      caixa_largura-distancia_borda_texto))
texto=$(               fmt -sw $texto_largura)
num_linhas=$(          echo "$texto" | wc -l)
total_linhas=$((       num_linhas+2))    # texto + bordas
  horizontais
### Checagem do tamanho do texto
if [ $num_linhas -gt $texto_max_linhas ];then
   echo "Texto muito extenso, não vai caber na tela"
   exit 1
fi
### Compõe a linha horizontal da caixa
# É $caixa_largura-2 porque os "cantos" serão feitos com o +
for i in $(seq $((caixa_largura-2))); do
   linha_caixa="$linha_caixa-"
done
```

```
### Limpa a tela
echo -ne '\033c'
### Desenha a caixa
echo -ne "\033[$caixa_linha_inicio;0H"       # pula p/a linha
  inicial
echo -ne "\033[${caixa_cor}m"               # liga a cor da caixa
for i in $(seq $total_linhas); do           # para cada linha...
   echo -ne "\033[${caixa_coluna_inicio}G"      # vai p/coluna
  inicial
   if [ $i -eq 1 -o $i -eq $total_linhas ]; then # 1a ou última
  linha
      echo +$linha_caixa+                       # borda
  horizontal
   else                                         # senão
      echo -e "|\033[${caixa_coluna_fim}G|"     # bordas
  verticais
   fi
done
echo -e '\033[m'                            # desliga as cores
### Coloca o texto dentro da caixa
echo -ne "\033[$((caixa_linha_inicio+1));0H" # pula p/a linha
  inicial
echo -ne "\033[${texto_cor}m"               # liga a cor do texto
echo "$texto" | while read LINHA; do        # para cada linha...
   echo -e "\033[${texto_coluna_inicio}G$LINHA"  # posiciona e
  mostra
done
echo -e '\033[m'                            # desliga as cores
echo
```

Capítulo 7

Expressões regulares

"Conheci as expressões regulares e minha vida mudou. De repente, várias linhas de código com loops, testes e contadores puderam ser trocadas por uma única linha mágica que faz tudo." Este é um depoimento comum de se ouvir de um recém-convertido. Aprenda a dominar todos os símbolos que compõem uma expressão regular, entendendo até mesmo aquelas mais complicadas. E prepare-se, seus programas nunca mais serão os mesmos.

Expressões Regulares. Um assunto para o qual muitos torcem o nariz ao ouvir falar, mas que sempre acaba aparecendo na resolução dos mais diversos problemas. Para quem não conhece ou não domina o tema, é difícil perceber a utilidade de saber escrever todos aqueles símbolos estranhos. Mas à medida que vai se aprendendo, aplicando, tudo começa a clarear. E vicia. Você sempre vai querer usar as expressões, para tudo. É um caminho sem volta :)

Várias linhas de código com algoritmos de verificação, loops e varreduras podem ser trocadas por uma única linha de `grep` ou `sed` que usa uma expressão regular esperta. Quanto mais você aprender sobre o assunto, mais tempo poupa. Seus códigos ficarão menores e mais eficientes. Você conseguirá ser mais produtivo, resolvendo problemas complicados com várias expressões eficientes.

Neste capítulo aprenderemos o que são e como utilizar estas expressões. Não tenha medo, você vai perceber que não é complicado. É como uma minilinguagem com suas próprias regras. São poucas regras, em poucos minutos você aprenderá tudo. No capítulo seguinte colocaremos o conhecimento em prática, utilizando as expressões para formatar códigos HTML e XML.

Então respire fundo e prepare-se para conhecer algo que vai mudar sua vida. Sério!

O que são expressões regulares

Uma expressão regular (ER) é um método formal de se especificar um padrão de texto. É uma máscara, um modelo, um padrão de pesquisa que serve para encontrar trechos de um texto. A expressão, quando aplicada em um texto qualquer, retorna sucesso caso este texto obedeça a todas as suas condições. Diz-se que o texto "casou" com a expressão.

As expressões servem para se dizer algo abrangente de forma específica. Definido o padrão de busca, tem-se uma lista (finita ou não) de possibilidades de casamento. Em um exemplo rápido, a expressão regular `[rgp]ato` pode casar com rato, gato e pato.

As expressões regulares são úteis para buscar ou validar textos de

formato conhecido, porém de conteúdo variável, como:

- Data e horário.
- Número IP.
- Endereço de e-mail, endereço de Internet.
- Declaração de uma função().
- Dados na coluna N de um texto.
- Dados que estão entre <tags></tags>.
- Número de telefone, RG, CPF, cartão de crédito.

Metacaracteres

Uma expressão regular é formada por caracteres normais como "a", "9" e "M" e por caracteres especiais chamados metacaracteres, como "$", "?" e "*".

Cada metacaractere é uma ferramenta que tem uma função específica. Eles servem para dar mais poder às pesquisas, informando padrões e posições impossíveis de se especificar usando somente caracteres normais. Com eles você pode especificar padrões como "um número de três dígitos" e "uma data no formato DD/MM/AAAA".

Os metacaracteres são pequenos pedacinhos simples que, agrupados entre si, ou com caracteres normais, formam algo maior, uma expressão. O importante é compreender bem cada um individualmente e depois apenas lê-los em sequência. Então o ponto mais importante do aprendizado das expressões é entender o que faz cada um destes caracteres especiais.

Lista dos principais metacaracteres

Meta	Nome	Posicionamento
^	Circunflexo	Representa o começo da linha.
$	Cifrão	Representa o fim da linha.
Meta	**Nome**	**Texto**
[abc]	Lista	Casa as letras "a" ou "b" ou "c".

`[a-d]`	Lista	Casa as letras “a” ou “b” ou “c” ou “d”.
`[^abc]`	Lista negada	Casa qualquer caractere, exceto “a”, “b” e “c”.
`(esse|aquele)`	Ou	Casa as strings “esse” ou “aquele”.
Meta	**Nome**	**Quantidade**
`a{2}`	Chaves	Casa a letra “a” duas vezes.
`a{2,4}`	Chaves	Casa a letra “a” de duas a quatro vezes.
`a{2,}`	Chaves	Casa a letra “a” no mínimo duas vezes.
`a?`	Opcional	Casa a letra “a” zero ou uma vezes.
`a*`	Asterisco	Casa a letra “a” zero ou mais vezes.
`a+`	Mais	Casa a letra “a” uma ou mais vezes.
Meta	**Nome**	**Curingas**
`.`	Ponto	Casa um caractere qualquer.
`.*`	Curinga	Casa qualquer coisa, é o tudo e o nada.

Cuidado para não confundir estes metacaracteres com os curingas do shell (glob) que você já está acostumado a usar na linha de comando e nas opções do comando `case`. O asterisco por exemplo, no shell usamos `*.txt` para dizer que queremos todos os arquivos com a extensão txt. Já em expressões regulares o asterisco sozinho não diz nada, ele é apenas um repetidor que precisa de algo antes, como em `.*` ou `[0-9]*`. Uma coisa é uma coisa, outra coisa é outra coisa :)

Equivalência entre glob e expressões regulares

Glob (shell)	Expressões regulares
`*`	`.*`
`?`	`.`
`{a,b}`	`(a|b)`
`[abc]`	`[abc]`
`[^abc]`	`[^abc]`
`[0-9]`	`[0-9]`
`*.txt`	`.*\.txt`
`arquivo-??.txt`	`arquivo-..\.txt`
`arquivo.{txt|html}`	`arquivo\.(txt|html)`
`[Aa]rquivo.txt`	`[Aa]rquivo\.txt`

Conhecendo cada um dos metacaracteres

Para testar cada um dos metacaracteres disponíveis e conhecer seu funcionamento e detalhes, vamos usar um comando que você já domina há muito tempo: o `grep`. Talvez até hoje você só tenha usado o grep para pesquisar textos, mas saiba que ele já traz de fábrica o suporte às expressões regulares. Quando você faz:

```
grep Maria nomes.txt
```

Você na verdade está pesquisando a expressão regular "Maria" dentro do arquivo `nomes.txt`. Isso mesmo, o comportamento padrão do grep já é pesquisar usando as expressões. Como "Maria" tem apenas letras normais, não utilizando nenhum dos caracteres especiais que veremos a seguir, pode-se dizer que o `grep` procurou, na prática, pela **palavra** Maria. Mas internamente este texto é considerado uma expressão regular.

Para forçar o `grep` a pesquisar por palavras normais, desligando o suporte às expressões regulares, use a opção `-F` (`--fixed-strings`) ou chame-o como `fgrep`. Pode ser útil quando a palavra que você quer pesquisar possui algum dos símbolos especiais considerados metacaracteres.

Em nosso aprendizado usaremos um único arquivo de texto para testar as expressões com o `grep`. Este arquivo será o tradicional `/etc/passwd`, a base de dados de usuários de um sistema Unix. Caso você ainda não o conheça, este é um exemplo de seu conteúdo:

Arquivo /etc/passwd

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
news:x:9:13:news:/etc/news:
uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
games:x:12:100:games:/usr/games:/sbin/nologin
```

```
gopher:x:13:30:gopher:/var/gopher:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
nobody:x:99:99:Nobody:/:/sbin/nologin
ntp:x:38:38::/etc/ntp:/sbin/nologin
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
carlos:x:500:500:carlos:/home/carlos:/bin/bash
ana:x:501:501:Ana Paula Moreira:/home/ana:/bin/bash
acs:x:502:502:Antonio Carlos Silva:/home/acs:/bin/bash
```

São várias linhas, uma para cada usuário. Cada linha possui sete campos, separados entre si pelo caractere dois-pontos ":", no seguinte formato:

```
login : senha : UID : GID : Nome Completo : Diretório $HOME :
  shell
```

Todas as senhas estão protegidas em outro arquivo, por isso há somente uma letra "x" em seu lugar. Analisando a primeira linha, sabemos que ela é referente ao administrador do sistema (root). Seu número de usuário (UID) e seu número de grupo são iguais: zero. Seu nome é `root` mesmo, seu diretório padrão é o `/root` e seu shell é o Bash.

Nos exemplos seguintes, o `grep` fará pesquisas neste arquivo, utilizando expressões regulares. O mais divertido no aprendizado das expressões é que você pode testá-las diretamente na linha de comando e o resultado é instantâneo. Em poucos minutos você testa vários metacaracteres. É exatamente isso que faremos agora.

Digite em sua máquina os exemplos seguintes para ver a mágica das expressões acontecendo ao vivo. Mude os comandos, experimente, descubra novas possibilidades. Tente juntar vários metacaracteres diferentes para formar expressões maiores. Somente com a prática você irá aprender, não fique só na leitura!

O circunflexo ^

O primeiro metacaractere que veremos é o circunflexo "^" (ou chapeuzinho), que simboliza o início de uma linha. Basta colocá-lo no início do texto de pesquisa. Veja a diferença:

```
$ grep root /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
```

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

No primeiro comando, o grep procurou pela palavra root e ela aparece em duas linhas. No segundo comando, adicionamos o circunflexo para forçar que a palavra esteja no início da linha, então o grep retornou apenas uma linha.

Não confunda, não procuramos pelo texto ^root, mas, sim, pela palavra root no início da linha. Que diferença faz um simples ^ hein?

O cifrão $

O primo do circunflexo é o cifrão "`$`", que representa o fim de uma linha. Lembra que o último campo de cada linha do arquivo passwd é o shell do usuário? Agora com o metacaractere cifrão podemos pesquisar quais são os usuários que usam o Bash como shell:

```
$ grep bash$ /etc/passwd
root:x:0:0:root:/root:/bin/bash
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
carlos:x:500:500:carlos:/home/carlos:/bin/bash
ana:x:501:501:Ana Paula Moreira:/home/ana:/bin/bash
acs:x:502:502:Antonio Carlos Silva:/home/acs:/bin/bash
$
```

Acostume-se a ler o comando para entender melhor o que acontece: a expressão regular `bash$` procura pela palavra bash no final da linha. Ou, ainda, a palavra bash seguida de um fim de linha.

Sabemos que esse cifrão não é o mesmo cifrão usado para identificar variáveis no shell. Mas será que o shell sabe disso? Para evitar problemas, é simples: sempre use aspas. Isso evitará confusões com a expansão de variáveis do shell e ajudará a preservar a sua sanidade.

Juntando os dois metacaracteres já vistos, temos na mão uma expressão muito útil que serve para encontrar linhas vazias. O que é uma linha em branco senão um começo de linha seguido de um fim de linha?

```
$ grep '^$' /etc/passwd
$
```

A lista [ ]

Mudando um pouco de tarefa, que tal pesquisar os usuários chamados Carlos? No arquivo de exemplo há dois. Vamos tentar obter ambos com o `grep`:

```
$ grep 'carlos' /etc/passwd
carlos:x:500:500:carlos:/home/carlos:/bin/bash
$ grep 'Carlos' /etc/passwd
acs:x:502:502:Antonio Carlos Silva:/home/acs:/bin/bash
$
```

Puxa vida... Um está em minúsculas e o outro com a primeira letra maiúscula, sendo preciso usar o grep duas vezes para extraí-los. Para resolver este problema, vamos utilizar o metacaractere lista. Basta colocar entre colchetes "[]" todos os caracteres que podem aparecer em uma determinada posição. Em nosso caso, apenas dois bastam, "C" e "c":

```
$ grep '[Cc]arlos' /etc/passwd
carlos:x:500:500:carlos:/home/carlos:/bin/bash
acs:x:502:502:Antonio Carlos Silva:/home/acs:/bin/bash
$
```

A expressão `[Cc]arlos` serve para pesquisar por "Carlos" e "carlos" ao mesmo tempo. Repare bem no detalhe: toda a lista (os colchetes e seu conteúdo) vale para apenas uma posição, um caractere, uma letra. Dentro dos colchetes, pode-se colocar tantos caracteres quantos necessários, mas ela continua representando apenas um único caractere. Podemos combinar a lista com o circunflexo, produzindo uma expressão mais poderosa:

```
$ grep '^[aeiou]' /etc/passwd
adm:x:3:4:adm:/var/adm:/sbin/nologin
uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
ana:x:501:501:Ana Paula Moreira:/home/ana:/bin/bash
acs:x:502:502:Antonio Carlos Silva:/home/acs:/bin/bash
$
```

Este comando procura por linhas que começam com vogais. Note o uso do circunflexo para amarrar a pesquisa no início da linha. A leitura dessa

expressão fica assim: busque por linhas que começam com a letra “a”, ou “e”, ou “i”, ou “o”, ou “u”. Para fazer o inverso, obtendo as linhas que começam com consoantes, poderíamos listar todas elas dentro dos colchetes:

```
grep '^[bcdfghjklmnpqrstvwxyz]' /etc/passwd
```

Mas há outra possibilidade. Também é possível negar o conteúdo de uma lista, dizendo que aqueles caracteres são inválidos, permitindo todos os outros. Assim, para buscar as consoantes, podemos dizer: busque por linhas que não comecem com vogais.

```
$ grep '^[^aeiou]' /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
news:x:9:13:news:/etc/news:
games:x:12:100:games:/usr/games:/sbin/nologin
gopher:x:13:30:gopher:/var/gopher:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
nobody:x:99:99:Nobody:/:/sbin/nologin
ntp:x:38:38::/etc/ntp:/sbin/nologin
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
carlos:x:500:500:carlos:/home/carlos:/bin/bash
$
```

Caso o primeiro caractere dentro da lista seja um circunflexo, ele inverterá o sentido dessa lista, tornando-a uma lista negada. Ela casará qualquer caractere, **exceto** os listados após o "^". Então, essa expressão casa qualquer letra exceto as vogais.

Use a lista negada com cautela. Negar alguns caracteres significa permitir todos os outros! Números, símbolos, TAB e espaço em branco também fazem parte de “qualquer caractere, exceto vogais”. Em nosso exemplo atual isso não atrapalha, mas tenha sempre em mente que essa negação é bem abrangente.

Outra facilidade da lista é o intervalo. Basta colocar um hífen entre duas

letras. Por exemplo [a-f] é interpretado como "todas as letras entre a e f, inclusive", ou seja: abcdef. De maneira similar [0-9] representa todos os números de zero a nove. Dessa maneira, é fácil procurar por linhas que contenham números:

```
grep '[0-9]' /etc/passwd
```

O ponto .

Às vezes, é necessário permitir **qualquer** caractere em uma certa posição. Por exemplo, para procurar usuários em que a segunda letra do login seja uma vogal, como mario e jose, mas não ana. Não importa qual é a primeira letra, pode ser qualquer uma, mas a segunda deve ser uma vogal:

```
$ grep '^.[aeiou]' /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
news:x:9:13:news:/etc/news:
uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
games:x:12:100:games:/usr/games:/sbin/nologin
gopher:x:13:30:gopher:/var/gopher:/sbin/nologin
nobody:x:99:99:Nobody:/:/sbin/nologin
carlos:x:500:500:carlos:/home/carlos:/bin/bash
$ grep '^.[aeiou]' /etc/passwd | cut -c 2
o
i
a
a
a
e
u
a
o
o
a
$
```

O ponto casa qualquer caractere, seja letra, número, símbolo e até espaço em branco. Esta expressão diz: A partir do começo da linha, procure qualquer caractere seguido de uma vogal. Usado junto com as âncoras de começo e fim de linha, o ponto também serve para se procurar por linhas de um tamanho fixo, independente do seu conteúdo. Que tal se obter apenas as linhas que possuam exatamente 27 caracteres?

```
$ grep '^.........................$' /etc/passwd
news:x:9:13:news:/etc/news:
$
```

Sim, o ponto é um caractere especial. Para casar um ponto literal, escape-o com a contrabarra "\."

As chaves { }

Será que no exemplo anterior há 27 pontos mesmo? Como saber ao certo? Para evitar este tipo de dúvida, as expressões regulares nos dão ferramentas mais flexíveis para fazer a pesquisa. Colocando um número entre chaves, indica-se uma quantidade de repetições do caractere (ou metacaractere) anterior:

```
$ egrep '^.{27}$' /etc/passwd
news:x:9:13:news:/etc/news:
$
```

Essa expressão faz o mesmo que a anterior: procura por linhas que tenham exatamente 27 caracteres, quaisquer que sejam eles. A vantagem é que basta olhar o número de repetições, não precisa ficar contando os pontinhos.

Note que desta vez foi usado o egrep e não o grep. É porque as chaves fazem parte de um conjunto avançado de expressões regulares (*extended*) e o egrep lida melhor com elas. Se fosse para usar o grep normal, teria que escapar as chaves:

```
grep '^.\{27\}$' /etc/passwd
```

Feio! Vamos usar somente o egrep daqui para a frente para evitar tais escapes. As chaves também aceitam intervalos, sendo possível procurar por linhas que tenham de 20 a 40 caracteres:

```
$ egrep '^.{20,40}$' /etc/passwd
```

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
halt:x:7:0:halt:/sbin:/sbin/halt
news:x:9:13:news:/etc/news:
nobody:x:99:99:Nobody:/:/sbin/nologin
ntp:x:38:38::/etc/ntp:/sbin/nologin
$
```

Se omitir o segundo número e manter a vírgula, fica um intervalo aberto, sem fim. Assim, pode-se informar repetições de "no mínimo N vezes". Para obter as linhas que possuem 40 caracteres ou mais:

```
$ egrep '^.{40,}$' /etc/passwd
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
games:x:12:100:games:/usr/games:/sbin/nologin
gopher:x:13:30:gopher:/var/gopher:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
carlos:x:500:500:carlos:/home/carlos:/bin/bash
ana:x:501:501:Ana Paula Moreira:/home/ana:/bin/bash
acs:x:502:502:Antonio Carlos Silva:/home/acs:/bin/bash
$
```

E sabe qual a melhor parte dessa brincadeira? É que todos estes metacaracteres que estamos aprendendo podem ser combinados entre si! É possível, por exemplo, repetir uma lista. Assim podemos obter uma lista de todos os usuários cujo número de identificação (UID ou GID) tenha três dígitos ou mais:

```
$ egrep '[0-9]{3,}' /etc/passwd
games:x:12:100:games:/usr/games:/sbin/nologin
carlos:x:500:500:carlos:/home/carlos:/bin/bash
ana:x:501:501:Ana Paula Moreira:/home/ana:/bin/bash
```

```
acs:x:502:502:Antonio Carlos Silva:/home/acs:/bin/bash
$ egrep '[0-9]{3,}' /etc/passwd | cut -d : -f 1
games
carlos
ana
acs
$
```

O curinga .* (AND)

Quando se procura por dois trechos específicos de uma mesma linha, não importando o que há entre eles, usa-se o curinga ponto-asterisco ".*" para significar "qualquer coisa". Um exemplo é procurar os usuários que começam com vogais e usam o shell Bash:

```
$ egrep '^[aeiou].*bash$' /etc/passwd
ana:x:501:501:Ana Paula Moreira:/home/ana:/bin/bash
acs:x:502:502:Antonio Carlos Silva:/home/acs:/bin/bash
$
```

Procuramos por uma linha que comece com uma vogal e termine com a string bash, não importando o que tem no meio, pode ser qualquer coisa. Esse curinga funcionou como um AND lógico, onde só casa se encontrar as duas pesquisas.

O ou | (OR)

Para fazer o OR lógico, onde se procura por uma coisa ou outra, deve-se usar o pipe e delimitar as opções com os parênteses:

```
$ egrep '^(ana|carlos|acs):' /etc/passwd
carlos:x:500:500:carlos:/home/carlos:/bin/bash
ana:x:501:501:Ana Paula Moreira:/home/ana:/bin/bash
acs:x:502:502:Antonio Carlos Silva:/home/acs:/bin/bash
$
```

Essa expressão casa apenas as linhas dos três usuários citados. Ela começa procurando no início da linha "^", depois procura as palavras ana, carlos ou acs, seguido pelo dois-pontos.

Os outros repetidores * + ?

Outros metacaracteres que podem ser usados são o asterisco, o mais e a interrogação (chamado de opcional). Eles definem quantidades e funcionam exatamente como as chaves, porém com uma sintaxe mais prática de usar:

Meta	Nome	Equivalente	Descrição
?	opcional	{0,1}	Pode aparecer ou não (opcional)
*	asterisco	{0,}	Pode aparecer em qualquer quantidade
+	mais	{1,}	Deve aparecer no mínimo uma vez

Detalhes, detalhes, detalhes

- O intervalo `a-z` não inclui os caracteres acentuados! Então para procurar por letras minúsculas em um texto em português, deve-se fazer `[a-záéíóúàâêôãõç]`, ou caso o sistema esteja corretamente configurado para o português, usar a classe especial `[[:lower:]]`. Similarmente, use `[[:upper:]]` para as maiúsculas.
- Dependendo do aplicativo, a sintaxe das expressões pode ser diferente da que usamos no `egrep`. No `sed` e no `grep`, por exemplo, as chaves e o "ou" devem ser escapados para serem especiais. Então, o último exemplo ficaria assim no `grep`:

```
grep '^\(ana\|carlos\|acs\):' /etc/passwd
```

Programa	Opcional	Mais	Chaves	Ou	Grupo
awk	?	+	-	|	()
egrep, gawk	?	+	{,}	|	()
emacs, find	?	+	-	|	\(\)
grep, sed	\?	\+	\{,\}	|	\(\)
vim	\=	\+	\{,}	|	\(\)

- Há programas para auxiliar o aprendizado de expressões regulares. O txt2regex (http://txt2regex.sourceforge.net) é um programa modo texto feito 100% com builtins do Bash e constrói expressões do zero, apenas mostrando menus ao usuário, que escolhe o que quer. Se não estiver no console, use o excelente RegexPal (http://regexpal.com), que testa suas expressões online, diretamente no navegador.

■ Para aprofundar-se no assunto, leia meu outro livro *Expressões Regulares – Uma abordagem divertida* 5a edição(ISBN 85-7522-474-8, Editora Novatec), que é um guia completo sobre o assunto, cobrindo deste o básico até o uso avançado das expressões nas mais diversas linguagens e programas. Há também um guia disponível gratuitamente na Internet, ótimo para pesquisas e consultas rápidas. Tanto o livro quanto o guia podem ser encontrados no seguinte endereço:
http://aurelio.net/er.

Capítulo 8

Extração de dados da Internet

Atualmente, muitas das informações necessárias para uso pessoal e profissional estão disponíveis na Internet. Este é um novo campo de atuação para seus programas, que podem automatizar o processo de obtenção e tratamento destas informações. Aprenda a baixar, manipular e extrair dados de sites, blogs e feeds RSS/Atom.

Com o conhecimento adquirido em expressões regulares, abre-se um mundo de possibilidades no tratamento e na manipulação de texto. Uma tarefa muito comum nesses tempos de Internet é a necessidade de manipular arquivos com marcações, como páginas HTML e arquivos XML.

Agora usaremos as expressões na prática, com exemplos reais de extração de dados de arquivos com marcações. Ao final deste capítulo você saberá como fazer seu próprio leitor de notícias (*feeds*) em shell, eliminando a necessidade de acessar os sites para se manter informado.

Parsing de código HTML, XML e semelhantes

A Internet é composta por uma infinidade de páginas com textos, links, imagens, tabelas e outros componentes. Estas páginas são feitas usando-se um código chamado HTML, que é uma linguagem dita de marcação. Algumas palavras de um texto normal podem ser marcadas, ou seja, circundadas por tags que definem a sua relevância ou a sua aparência.

Exemplo de código HTML

```
<p align="right">Este é um parágrafo em HTML, que
está alinhado à direita. Podemos ter palavras em
<b>negrito</b> ou em <i>itálico</i>.</p>
```

Cada marcação (tag) é feita no formato `<...>`, usando os sinais de maior e menor para delimitar o nome da tag. Uma tag também pode ter atributos e argumentos, vide a tag `<p>` do exemplo, que possui definido o atributo de alinhamento. A tag é fechada no formato `</...>`, ou seja, adicionando uma barra logo após o sinal de menor. Assim, tudo o que está dentro do par `<tag>...</tag>` é afetado por esta marcação. Por isso, o texto `<i>itálico</i>` fica em itálico, pois a tag `i` foi aberta e fechada ao seu redor.

Arquivos XML utilizam o mesmo formato, porém os nomes das tags são diferentes. Em vez de indicar a aparência como negrito ou itálico, as tags do XML indicam o tipo do texto, se ele é um número de série, um endereço, um e-mail ou uma descrição.

Para nós, programadores, que precisaremos manipular e extrair dados que estão em arquivos HTML e XML, a diferença entre ambos pode ser ignorada. O que importa é saber que os dados estão marcados por tags no formato `<...>`. Cabe a nós pensar em maneiras de como lidar com estes dados.

Como remover todas as tags (curinga guloso)

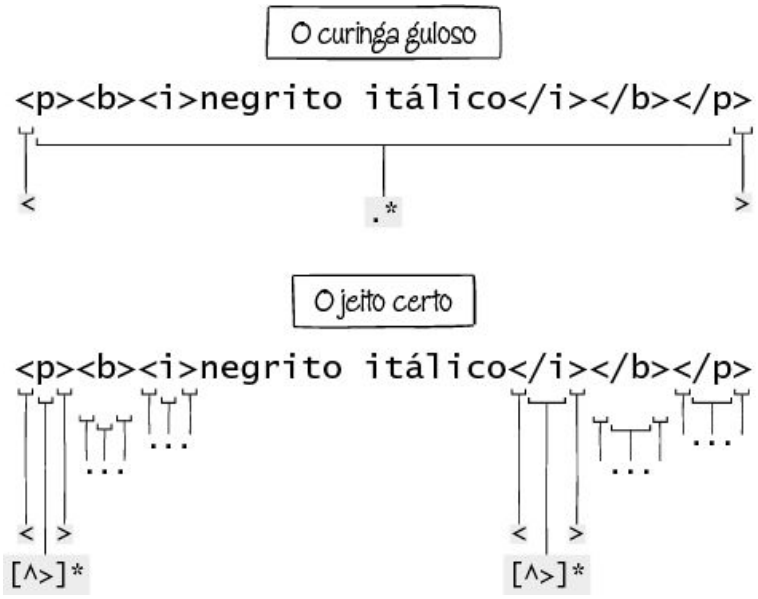
A pergunta mais frequente quando alguém precisa lidar com um arquivo marcado, é como remover todas as tags, deixando apenas o texto. Na obtenção de informações, as tags acabam sendo apenas poluição para quem quer apenas texto. Veja como livrar-se delas com facilidade:

```
$ echo '<p><b><i>negrito itálico</i></b></p>' | sed 's/<[^>]*>//g'
negrito itálico
$
```

Esse comandinho minúsculo do sed, que usa expressões regulares, diz: apague todas as marcações no formato `<...>`. E pronto, as tags foram removidas. Você pode usá-lo com arquivos completos, que todas as tags serão apagadas. Perceba que o conteúdo da tag foi indicado como `[^>]*` que significa "qualquer coisa fora um >, em qualquer quantidade". Aqui não é possível usar o curinga ponto-asterisco, veja o porquê:

```
$ echo '<p><b><i>negrito itálico</i></b></p>' | sed 's/<.*>//g'
$
```

Toda a linha foi apagada! Isso aconteceu porque o curinga ponto-asterisco é guloso. Ele sempre tomará para si a maior quantidade possível de caracteres. Como ele significa qualquer coisa, ele foi pegando tudo até o fim da linha, liberando apenas o último ">" para que a expressão regular fosse válida: case um <, seguido de qualquer coisa, seguido de um >. Na prática, o curinga casou `p><b><i>negrito itálico</i></b></p`.



O curinga guloso entrando em ação

Como extrair links (URL)

Outra tarefa muito comum ao lidar-se com códigos HTML é extrair a URL de determinado link, ou seja, o endereço de Internet para onde ele aponta.

A tag do link é `<a>`, que aceita um atributo chamado `href`. Este atributo guarda o endereço que será acessado ao se clicar no link. Dentro do par de tags está o nome do link, que é o que aparecerá para o usuário. Exemplo:

```
<a href="http://google.com.br">Site do Google</a>
```

Assim, no navegador aparecerá o link "Site do Google" (em azul, sublinhado) e ao clicar nele o endereço http://google.com.br será acessado. Um tarefa rotineira para o programador é extrair somente o endereço desta tag. É o que faremos agora.

Muita atenção deste trecho adiante, pois veremos vários detalhes sobre a manipulação de arquivos HTML. O conteúdo avançará de maneira sequencial, então cuidado para não perder o fio da meada! Estique os braços, gire os pulsos e respire fundo. Tudo certo? Vamos lá.

pagina1.html

```
<html>
```

```
<body>
<p>Acesse a página do
<a href="http://www.google.com.br">Google</a>.
</p>
</body>
</html>
```

Temos como exemplo uma página HTML simples, que possui apenas um link. A tarefa agora é extrair o endereço do texto Google. Começaremos pelo básico, que é extrair somente a linha desejada, ignorando as demais:

```
$ fgrep Google pagina1.html
<a href="http://www.google.com.br">Google</a>.
$
```

Lembre-se de que o fgrep procura apenas por palavras normais, não expressões regulares. Por enquanto usaremos assim, já que nossas amigas expressões ainda não são necessárias. Mas não por muito tempo...

Fácil, facílimo. Aqui está a linha da qual precisamos extrair o endereço, que está dentro do atributo `href`, entre aspas. Para obter este endereço deve haver umas 74 maneiras diferentes de fazer isso, usando `sed`, `awk`, `IFS`... Mas o `cut` parece ser a alternativa mais eficiente:

```
$ fgrep Google pagina1.html | cut -d \" -f 2
http://www.google.com.br
$
```

Usando as aspas como delimitador, extraia o segundo campo. As aspas precisam ser escapadas para evitar problemas com o shell. Tem algo errado aqui, não acha? Foi fácil demais! Geralmente, se algo foi muito fácil de resolver, é porque o problema foi subestimado. A página de exemplo estava muito simples. Vamos complicá-la um pouco:

pagina2.html

```
<html>
<body>
Você conhece o Google?
<p>Acesse a página do
"buscador", o <a href="http://www.google.com.br">Google</a>.
</p>
```

```
</body>
</html>
```

Agora a palavra Google aparece em outro trecho do arquivo, e na mesma linha do link há um texto normal entre aspas. Pronto, estas pequenas mudanças foram suficientes para quebrar nossa solução tosca:

```
$ fgrep Google pagina2.html | cut -d \" -f 2
Você conhece o Google?
buscador
$
```

O primeiro problema é que o `fgrep` está muito genérico e pega linhas além da desejada. Não basta procurar pelo texto desejado, é preciso especificar que este texto deve estar dentro de tags de link `<a></a>`. Para não complicar muito, vamos pensar simples: o texto deve terminar com um `</a>` e deve ter um > antes, que é o final do `<a href...>`.

```
$ fgrep '>Google</a>' pagina2.html
"buscador", o <a href="http://www.google.com.br">Google</a>.
$
```

Certo, conseguimos pescar somente a linha desejada. O esquema antigo do `cut` já não funciona mais, pois podemos ter várias aspas na mesma linha, não há como prever. A missão agora é tirar o lixo. O `sed` é um bom candidato. Primeiro, tudo o que vier até o `href="` deve ser apagado:

```
$ fgrep '>Google</a>' pagina2.html | sed 's/.*href="//'
http://www.google.com.br">Google</a>.
$
```

O `sed` trocou "qualquer coisa" seguida da string `href="` por nada. Legal, o endereço já está no começo da linha, então agora falta arrancar o lixo do final. Aparentemente apagar tudo o que vier depois das aspas é o suficiente.

```
$ fgrep '>Google</a>' pagina2.html | sed 's/.*href="// ; s/".*//'
http://www.google.com.br
$
```

Feito! O extrator de links está começando a ficar mais esperto, mas ainda está longe de ser perfeito. Vamos complicar um pouco mais a página de exemplo, mudando a linha do link para:

```
"buscador", o <a hReF=HTTP://www.GOOGLE.com.br>Google</A>.
```

O HTML permite que os nomes das tags e dos atributos sejam colocados em maiúsculas ou minúsculas, tanto faz, então `<a></a>` e `<A></A>` são iguais. No exemplo, o fechamento da tag `<a>` foi colocado em maiúsculas e seu atributo foi colocado como `hReF`. Outra mudança, que somente os leitores com visão de raio X perceberam, é que agora o endereço foi colocado sem as aspas. Isso também é permitido.

```
$ fgrep -i '>Google</a>' pagina3.html
"buscador", o <a hReF=HTTP://www.GOOGLE.com.br>Google</A>.
$
```

Falou em maiúsculas e minúsculas, falou em `-i` no `fgrep`. Assim conseguimos voltar ao primeiro passo de extrair a linha do link. Já para retirar o lixo, algumas versões do `sed` não sabem ignorar a diferença na caixa das letras automaticamente, forçando-nos a escrever uma expressão regular feia como `[Hh][Rr][Ee][Ff]` para casar qualquer variação da palavra `href`. Neste caso, fica mais fácil eliminar o problema em vez de remendar a solução: basta converter a linha toda para minúsculas:

```
$ fgrep -i '>Google</a>' pagina3.html | tr A-Z a-z
"buscador", o <a href=http://www.google.com.br>google</a>.
$
```

Como só precisamos do endereço, não será um problema usar este "artifício técnico criativo". O que falta agora para deixarmos a linha somente com o endereço é fazer o `sed` reconhecê-lo sem aspas:

```
$ fgrep -i '>Google</a>' pagina3.html | tr A-Z a-z |
> sed 's/.*href=// ; s/>.*//'
http://www.google.com.br
$
```

O comando foi quebrado em duas linhas para não ficar muito extenso. O "> " é o prompt secundário do shell, que mostra que ele está esperando mais comandos.

O comando `sed` foi modificado para apagar os lixos quando o link vier sem as aspas. Mas e quando vier com aspas? É, não vai funcionar. Mas bem, quem precisa de aspas afinal de contas, basta apagá-las!

```
$ fgrep -i '>Google</a>' pagina[23].html | tr A-Z a-z | tr -d \" |
> sed 's/.*href=// ; s/>.*//'
```

```
http://www.google.com.br
http://www.google.com.br
$
```

Foi adicionado mais um `tr` para varrer as aspas e o comando funcionou tanto com a página 2 (link com aspas) como com a página 3 (link sem aspas). Lembre-se de que o `pagina[23].html` é um curinga do shell e não uma expressão regular!

Tudo bem, neste caso a sintaxe do glob e das expressões regulares é exatamente a mesma. Mas ainda assim vale o recado, para tentar manter sua sanidade com essa avalanche de siglas, símbolos, tags etc.

Parece que já temos uma solução bem poderosa, que lida com várias pegadinhas do HTML, não é mesmo? É, quase. Não quero que você se exalte, mas sabia que esse tal de HTML é tão bonzinho com os escrevedores de página que ele permite que uma tag ocupe várias linhas? Bonzinho para uns, cruel para outros. Dessa vez fazemos parte do time dos sofredores, pois temos que contemplar esta "facilidade" em nosso programa.

pagina4.html

```
<html>
<body>
Você conhece o Google?
<p>Acesse a página do
"buscador", o <a
hReF="HTTP://www.GOOGLE.com.br"
>Google</A>.
</p>
</body>
</html>
```

Não se desespere. Em vez disso, prepare-se para aprender uma importante Tática Ninja: Se o problema mudar de forma e seu código parar de funcionar, não mude o código, mude o problema. Como assim?

Pense comigo: já temos um código que funciona, extraindo os links dos arquivos. De repente, chega um arquivo diferente que traz o link em múltiplas linhas. Há dois caminhos que levam em direção à solução:

- Mudar o código que atualmente funciona, fazendo-no contemplar esta situação nova, adicionando complexidade e possivelmente, bugs.
- Se o problema são as quebras de linha, basta removê-las e a solução atual continuará funcionando.

Shell é diferente, pense diferente! Acostume-se a pensar sobre o problema antes de sair codificando a solução. Às vezes uma ideia nova pode facilitar muito a sua vida, reduzindo o tempo de desenvolvimento. Não se prenda às velhas práticas, liberte-se!

Após esta lição preciosa vinda diretamente do Oriente, desfaremos o problema. Então, basta remover as quebras de linha? Ah, fica fácil com o `tr`. Ei, mas assim o arquivo todo ficará sendo apenas uma linha...

```
$ tr -d '\n' < pagina4.html
<html><body>Você conhece o Google?<p>Acesse a página do"buscador",
o <ahReF="HTTP://www.GOOGLE.com.br">Google</A>.</p></body></html>
```

Se tiver mais de um link vai dar problema, o ideal é que cada tag estivesse em uma linha, para ficar fácil pescar a linha desejada com o `grep`. Outro problema é que o "ahReF" ficou grudado, então não basta remover as quebras de linha, é melhor trocá-las por um espaço em branco.

```
$ tr '\n' ' ' < pagina4.html | awk 'gsub("<[^/]", "\n&")'
<html>
<body> Você conhece o Google?
<p>Acesse a página do "buscador", o
<a hReF="HTTP://www.GOOGLE.com.br" >Google</A>. </p> </body>
  </html>
```

Beleza! O tr trocou todas as quebras de linha por espaços em branco, deixando todo o texto em uma única (longa) linha. Em seguida, o `awk` colocou uma quebra de linha antes de cada tag de abertura `<[^/]` (sinal de menor seguido de qualquer coisa exceto uma barra, para não pegar as tags de fechamento). E aqui o texto está novamente no formato que nosso código conhece, então podemos usá-lo sem modificações:

```
$ tr '\n' ' ' < pagina4.html | awk 'gsub("<[^/]", "\n&")' |
> fgrep -i '>Google</a>' | tr A-Z a-z | tr -d \" |
> sed 's/.*href=// ; s/>.*//'
http://www.google.com.br
$
```

Tudo certo. Esta técnica de deixar cada tag em uma linha pode ser muito útil para primeiro arrumar o HTML, somente então começar a lidar com ele. Outros cenários onde ela vai salvar seu dia é ao receber um arquivo HTML com quebras de linha em um formato diferente do seu (Windows CR+LF, Linux LF) ou arquivos gerados por outros programas, que gostam de colocar a página inteira em uma única longa linha.

Concluindo esta introdução, gostaria de salientar que o importante até aqui não foram as ferramentas utilizadas, nem o exemplo em si, mas, sim, apresentar que tipo de problemas você vai enfrentar ao lidar com arquivos HTML. A flexibilidade na escrita das tags pelo usuário gera problemas para os programas que vão manipular o arquivo, é bom estar ciente disso. Arquivos XML tendem a ser mais fáceis por forçar que as tags estejam em minúsculas e que sempre se usem aspas, não sendo tão liberais quanto o HTML.

Há uma ferramenta chamada de tidy, que é especialista em arrumar e alinhar códigos HTML/XML, muito útil para corrigir códigos malfeitos. Há várias opções de linha de comando para controlar sua execução, vale a pena dar uma lida em sua man page. Caso o comando `tidy` não esteja instalado em seu sistema, baixe-o de http://tidy.sourceforge.net.

Extração de manchetes da Internet

O conhecimento recém-adquirido de lidar com marcações HTML pode ser ainda mais poderoso se combinado com uma coisinha sem importância dos dias de hoje: Internet. São milhares de páginas, cada uma com o seu próprio formato e disposição dos dados.

Usando o shell e suas ferramentas, é possível automatizar o processo de obtenção de informação, agilizando o acesso e economizando tempo. Em vez de abrir o navegador, acessar a página, ignorar todos os banners e texto inútil para finalmente encontrar a informação, um programa em shell pode fazer tudo isso automaticamente, mostrando na tela apenas o texto importante.

O navegador em modo texto lynx (http://lynx.isc.org) é um aliado poderoso. Ele pode baixar uma página HTML inteira com o código, ou renderizá-la, mostrando apenas seu conteúdo. Como em ambos os casos ele manda a

página para a saída padrão, o `lynx` é uma ótima ferramenta para usar com filtros.

Principais opções do lynx

Opção	Descrição
`-dump`	Renderiza uma página HTML e manda para STDOUT.
`-source`	Baixa o fonte HTML de uma página e manda para STDOUT.
`-nolist`	Usado com o `-dump`, para omitir a listagem de links.
`-width`	Define a largura máxima do texto (o padrão é 80).

Exemplos:

```
$ lynx -dump -width=70 arquivo.html > arquivo.txt
$ lynx -source http://google.com > google.html
```

Vamos brincar? A ideia é extrair as cinco manchetes mais recentes do site BR-Linux (http://br-linux.org), que publica diariamente várias notícias interessantes sobre Linux e software livre. As manchetes estão na página principal. Veremos três maneiras diferentes de realizar esta tarefa, cada uma com sua particularidade: extrair manchetes do texto, do código HTML e do feed XML.

Extraindo manchetes do texto

O que queremos é texto, então nada mais fácil que trabalhar diretamente com texto, sem tags e outras complicações para atrapalhar. Usaremos o lynx para baixar a página inicial do site e já renderizá-la, formatando seu texto e eliminando as tags HTML.

```
$ lynx -dump -nolist http://br-linux.org > brlinux.txt
```

Execute este comando agora em sua máquina e analise o arquivo brlinux.txt. Precisamos pescar as manchetes, mas para isso é preciso identificar um padrão que as diferencie do restante do texto. Olhe com atenção e tente encontrar este padrão. Vá lá, eu espero.

Caso as letras acentuadas não estejam aparecendo corretamente, é porque a codificação dos caracteres do seu sistema não está batendo com a codificação do site, que é UTF-8. Para resolver este problema, use o comando `iconv` para converter o arquivo para o formato do seu sistema, que geralmente é o ISO-8859-1 (ou latin1). Exemplo: `iconv -f utf-8 -t latin1 brlinux.txt > brlinux-latin1.txt`

Encontrou? O padrão das manchetes é que elas são colocadas bem no início da linha:

```
Instalando o modem Motorola SM56 "manualmente"
   Seg, 19/11/2007 - 10:30
     "Se você procurar por material sobre o Motorola SM56 para
     GNU/Linux, você encontrará muitos depoimentos de dor de
  cabeça e
     dificuldades para configuração. A empresa produziu um driver
  que
     funciona apenas para a versão 2.4 do kernel.
     ...
```

Parece fácil, não? Basta procurar por linhas que iniciam com uma letra maiúscula.

```
$ grep '^[A-Z]' brlinux.txt
BR-Linux.org BR-Linux.org
Navegação
User login
Destaques de hoje no BR-Linux
Instalando o modem Motorola SM56 "manualmente"
Vídeo: Novidades da próxima versão do Compiz Fusion
Fim do Kurumin? Ainda não. Alguém se habilita?
Opções de VoIP em plataformas livres
MPlayer promove inclusão digital de dispositivos com Windows
  Mobile
Vídeo-aulas de desenho de Mozart Couto - com GIMP
Gerenciamento de atualizações : uma solução simples e eficaz
I ECSL - I Encontro Cefetiano de Software Livre, em Sergipe
João Pessoa sedia o II Encontro de Software Livre da Paraíba
Permissões e propriedades de arquivos e diretórios no linux
BIOS, Barramentos e o Linux
Os primeiros testes com o "nosso" Classmate
A semana no BR-Linux: 11 anos de BR-Linux!
Notícias anteriores
Publicidade*
Anunciantes e patrocinadores
Efetividade.net
Rec6 Tecnologia
Meio Bit
```

```
RSS e mais
$
```

Ops! Vieram mais linhas além das manchetes. Estas linhas adicionais são os menus e outros componentes do site, que o `grep` acabou pegando também. É preciso removê-las. Há dois padrões que endereçam este texto indesejado:

- Da primeira linha até a linha que começa com "Destaques de hoje"
- Da linha "Notícias anteriores" até a última linha

Mais uma vez o `sed` é chamado para o resgate. Sabemos que é possível indicar faixas de texto para ele, podendo usar diretamente o número da linha, ou seu conteúdo. Então basta endereçar o bloco não desejado e usar o comando para apagá-lo:

```
$ grep '^[A-Z]' brlinux.txt | sed '1,/^Destaques de hoje/ d' |
  head -n 5
Instalando o modem Motorola SM56 "manualmente"
Vídeo: Novidades da próxima versão do Compiz Fusion
Fim do Kurumin? Ainda não. Alguém se habilita?
Opções de VoIP em plataformas livres
MPlayer promove inclusão digital de dispositivos com Windows
  Mobile
$
```

Traduzindo: Querido `sed`, para a faixa de linhas que vai desde a primeira (`1`) até aquela que começar (`^`) com "Destaques de hoje", aplique o comando delete (`d`).

Como só precisamos das cinco primeiras manchetes, um simples `head` toma conta do recado e não é preciso se preocupar em apagar o bloco não desejado do final. Mas claro, se você fizer questão, basta um outro comando `sed`: `/^Notícias anteriores/,$ d`.

E é só isso, está pronto! Agora é só colocar o código em um programa e executá-lo sempre que quiser saber das quentinhas.

brlinux-1.sh

```
#!/bin/bash
# brlinux-1.sh
# Mostra as 5 últimas manchetes do BR-Linux
# Versão 1 que procura no texto
```

```
#
# Aurelio, Agosto de 2006
URL="http://br-linux.org"
# O padrão são linhas que iniciam com maiúsculas.
# Até a linha "Destaques de hoje" é lixo.
#
lynx -dump -nolist "$URL" |
    grep '^[A-Z]' |
    sed '1,/^Destaques de hoje/ d' |
    head -n 5
```

Testando a execução do programa:

```
$ ./brlinux-1.sh
Instalando o modem Motorola SM56 "manualmente"
Vídeo: Novidades da próxima versão do Compiz Fusion
Fim do Kurumin? Ainda não. Alguém se habilita?
Opções de VoIP em plataformas livres
MPlayer promove inclusão digital de dispositivos com Windows
  Mobile
$
```

Viu como usando expressões regulares é possível fazer programinhas bem interessantes com pouquíssimo código? Não precisou de loop com `while`, não precisou de `if`, não precisou de algoritmo algum. O que fizemos foi filtrar um texto poluído deixando apenas as informações importantes. Foi feita uma bela faxina :)

Extraindo manchetes do código HTML

No exemplo anterior tivemos a sorte de encontrar um padrão simples no meio do texto, o que facilitou muito a tarefa. Mas nem sempre isso é possível, pois o arquivo pode não estar muito bem-formatado. A alternativa, nestes casos, é tentar identificar o padrão em meios às tags do código HTML da página. Vamos modificar o programa anterior para aprender esta técnica nova. Primeiro vamos baixar o código:

```
$ lynx -source http://br-linux.org > brlinux.html
```

Dessa vez o `lynx` foi usado com a opção `-source` para baixar o código-fonte da página. Execute este comando e agora analise o arquivo

`brlinux.html`, tentando encontrar um padrão para as manchetes em meio aos códigos HTML. Tenha paciência, o código é feio e intimidante, mas as manchetes estão lá. Lembre-se da dica do `iconv` do tópico anterior, caso tenha problema com os acentos.

```
      <h1><a style="color: black; padding-top:8px; "
  href="/linux/instalando-o-modem-motorola-sm56-manualmente"
  title="Instalando o modem Motorola SM56
  &amp;quot;manualmente&amp;quot;">Instalando o modem Motorola SM56
  &amp;quot;manualmente&amp;quot;</a></h1>
```

Conseguiu? Essa foi mais difícil. Veja no exemplo: a manchete está em uma longa linha, dentro de um link (`<a>`) que por sua vez está dentro de um título (`<h1>`). Primeiro precisamos pescar essas linhas com as manchetes para depois limpá-las, deixando apenas o texto.

```
$ grep '<h1><a style' brlinux.html
      <h1><a style="color: black; padding-top:8px; "
  href="/linux/instalando-o-modem-motorola-sm56-manualmente"
  title="Instalando o modem Motorola SM56
  &amp;quot;manualmente&amp;quot;">Instalando o modem Motorola SM56
  &amp;quot;manualmente&amp;quot;</a></h1>
      <h1><a style="color: black; padding-top:8px; "
  href="/linux/novidades-da-proxima-versao-do-compiz-fusion"
  title="Vídeo: Novidades da próxima versão do Compiz
  Fusion">Vídeo: Novidades da próxima versão do Compiz Fusion</a>
  </h1>
      <h1><a style="color: black; padding-top:8px; "
  href="/linux/fim-do-kurumin-ainda-nao-alguem-se-habilita"
  title="Fim do Kurumin? Ainda não. Alguém se habilita?">Fim do
  Kurumin? Ainda não. Alguém se habilita?</a></h1>
    ...
$
```

Funcionou, apareceram apenas as linhas desejadas. Fica difícil de enxergar por causa de todas essas tags do HTML, mas ali estão as manchetes (em negrito). Mas já que são justamente estas tags que estão atrapalhando, vamos apagá-las. Já sabemos como fazer isso, lembra? É aquela expressão regular pequenina lá do início do capítulo `<[^>]*>`, que vamos colocar em um `sed` bacana:

```
$ grep '<h1><a style' brlinux.html | sed 's/<[^>]*>//g'
      Instalando o modem Motorola SM56 &amp;quot;manualmente&amp;quot;
```

```
      Vídeo: Novidades da próxima versão do Compiz Fusion
      Fim do Kurumin? Ainda não. Alguém se habilita?
      Opções de VoIP em plataformas livres
      MPlayer promove inclusão digital de dispositivos com Windows
  Mobile
      Vídeo-aulas de desenho de Mozart Couto - com GIMP
      Gerenciamento de atualizações : uma solução simples e eficaz
      I ECSL - I Encontro Cefetiano de Software Livre, em Sergipe
      João Pessoa sedia o II Encontro de Software Livre da Paraíba
      Permissões e propriedades de arquivos e diretórios no linux
      BIOS, Barramentos e o Linux
      Os primeiros testes com o &quot;nosso&quot; Classmate
      A semana no BR-Linux: 11 anos de BR-Linux!
$
```

Esse comandinho de nada varreu todas as tags, limpando o texto. Lindo, não? O pior já passou, agora só faltam alguns ajustes pequenos para formatar melhor esta saída:

- Resgatar as aspas do texto original, que estão codificadas como `&quot;` - Veja exemplo na primeira notícia.
- Apagar aqueles espaços em branco do início da linha.
- Limitar o resultado para apenas cinco manchetes.

```
$ grep '<h1><a style' brlinux.html |
> sed 's/<[^>]*>//g ; s/&quot;/"/g ; s/^  *//' | head -n 5
Instalando o modem Motorola SM56 "manualmente"
Vídeo: Novidades da próxima versão do Compiz Fusion
Fim do Kurumin? Ainda não. Alguém se habilita?
Opções de VoIP em plataformas livres
MPlayer promove inclusão digital de dispositivos com Windows
  Mobile
$
```

Agora, sim! Com uma segunda substituição dentro do `sed`, restauramos as aspas. Uma terceira substituição usa uma expressão regular para apagar qualquer quantidade de espaços em branco no início da linha e finalmente vem o `head` para limitar o número de manchetes.

Em vez do `head`, poderia ter sido colocado mais um comando `5q` dentro do `sed`, que o efeito seria o mesmo. Mas com o `head` fica mais claro e fácil de alterar. Nunca

se esqueça da manutenção!

brlinux-2.sh

```
#!/bin/bash
# brlinux-2.sh
# Mostra as 5 últimas manchetes do BR-Linux
# Versão 2 que procura no código HTML
#
# Aurelio, Agosto de 2006
URL="http://br-linux.org"
# O padrão são linhas com "<h1><a style".
# O sed remove as tags HTML, restaura as aspas e
# apaga os espaços do início.
# O head limita o número de manchetes em 5.
#
lynx -source "$URL" |
    grep '<h1><a style' |
    sed '
        s/<[^>]*>//g
        s/&quot;/"/g
        s/^  *//' |
    head -n 5
```

Não ficou tão complicado, basta entender bem o que cada pedacinho desses faz. A sistemática é sempre a mesma: baixa o arquivo, identifica e pesca o padrão, apaga o lixo, formata o texto e mostra na tela. Testando:

```
$ ./brlinux-2.sh
Instalando o modem Motorola SM56 "manualmente"
Vídeo: Novidades da próxima versão do Compiz Fusion
Fim do Kurumin? Ainda não. Alguém se habilita?
Opções de VoIP em plataformas livres
MPlayer promove inclusão digital de dispositivos com Windows
  Mobile
$
```

Extraindo manchetes do Feed XML

Além das páginas HTML, vários sites de notícias também disponibilizam as manchetes em um formato especial chamado *feed*, que é um arquivo

XML contendo apenas as notícias e nada mais. Este arquivo geralmente é menor que a página HTML, sendo mais rápido de ser baixado. E por não possuir formatação nem outras informações, tende a ser bem mais fácil de pescar as manchetes.

Para descobrir o endereço do *feed* de um site, caso ele possua, veja o seu código HTML e procure por algum cabeçalho ou link que mencione as palavras RSS ou Atom. Dentro do argumento `href` está o endereço do XML que iremos utilizar. No caso do nosso exemplo BR-Linux, esta é a linha:

```
<a href=http://br-linux.org/linux/node/feed>RSS</a>
```

Alguns navegadores informam-no quando um site possui *feeds*, por meio de botões, imagens ou mensagens. Basta clicar neste aviso para acessar o arquivo XML.

Já temos o endereço, então vamos começar tudo de novo, primeiro com a análise do arquivo para encontrar o padrão das manchetes.

```
$ lynx -source http://br-linux.org/linux/node/feed > brlinux.xml
```

Vá lá, mais uma vez. Analise o arquivo XML e tente encontrar o padrão das manchetes. Não se preocupe, esse é bem fácil!

```
<item>
    <title>Instalando o modem Motorola SM56
&amp;quot;manualmente&amp;quot;</title>
    <link>http://br-linux.org/linux/instalando-o-modem-motorola-
sm56-manualmente</link>
    <description>&amp;lt;blockquote class=usertext&amp;gt;&amp;lt;p&amp;gt;
    &amp;amp;ldquo;Se você procurar por material sobre o Motorola SM56
para GNU/Linux, você encontrará muitos depoimentos de dor de
cabeça e dificuldades para configuração. A empresa produziu um
driver que funciona apenas para a versão 2.4 do kernel.
&amp;lt;/p&amp;gt;
  ...
    </description>
     <comments>http://br-linux.org/linux/instalando-o-modem-
motorola-sm56-manualmente#comments</comments>
    <category domain="http://br-
linux.org/linux/taxonomy/term/8">Distribuições</category>
    <pubDate>Mon, 19 Nov 2007 10:30:00 -0200</pubDate>
    <dc:creator>brain</dc:creator>
    <guid isPermaLink="false">10837 at http://br-
```

```
  linux.org/linux</guid>
</item>
```

Covardia, não tem nem graça! Todas as manchetes já estão lá, separadinhas dentro de tags `<title>`, esperando para serem pescadas. Essa é uma grande vantagem do XML sobre o HTML na obtenção de informações: o **conteúdo** está marcado, e não a sua formatação.

```
$ grep '<title>' brlinux.xml
<title>BR-Linux.org - Linux levado a sério desde 1996</title>
<title>Instalando o modem Motorola SM56 &amp;quot;manualmente&amp;quot;
  </title>
<title>Vídeo: Novidades da próxima versão do Compiz Fusion</title>
<title>Fim do Kurumin? Ainda não. Alguém se habilita?</title>
<title>Opções de VoIP em plataformas livres</title>
<title>MPlayer promove inclusão digital de dispositivos com
  Windows Mobile</title>
<title>Vídeo-aulas de desenho de Mozart Couto - com GIMP</title>
<title>Gerenciamento de atualizações : uma solução simples e
  eficaz</title>
<title>I ECSL - I Encontro Cefetiano de Software Livre, em
  Sergipe</title>
<title>João Pessoa sedia o II Encontro de Software Livre da
  Paraíba</title>
<title>Permissões e propriedades de arquivos e diretórios no
  linux</title>
<title>BIOS, Barramentos e o Linux</title>
<title>Os primeiros testes com o &amp;quot;nosso&amp;quot;
  Classmate</title>
<title>A semana no BR-Linux: 11 anos de BR-Linux!</title>
<title>Mais sobre o Ubuntu Mobile and Embedded</title>
$
```

Agora basta repetir o mesmo tratamento que já fizemos no programa anterior: apagar as tags e os espaços, restaurar as aspas e limitar a saída para cinco linhas. Note que a primeira linha é o título do site, então também será preciso apagá-la:

```
#!/bin/bash
# brlinux-3.sh
# Mostra as 5 últimas manchetes do BR-Linux
# Versão 3 que procura no Feed XML
```

```
#
# Aurelio, Agosto de 2006
URL="http://br-linux.org/linux/node/feed"
# O padrão são linhas com "<title>".
# O sed remove as tags HTML, restaura as aspas,
# apaga os espaços do início e remove a primera linha.
# O head limita o número de manchetes em 5.
#
lynx -source "$URL" |
    grep '<title>' |
    sed '
        s/<[^>]*>//g
        s/&quot;/"/g
        s/^  *//
        1d' |
    head -n 5
```

Testando a execução do programa:

```
$ ./brlinux-3.sh
Instalando o modem Motorola SM56 "manualmente"
Vídeo: Novidades da próxima versão do Compiz Fusion
Fim do Kurumin? Ainda não. Alguém se habilita?
Opções de VoIP em plataformas livres
MPlayer promove inclusão digital de dispositivos com Windows
  Mobile
$
```

Feeds RSS/Atom – Uma solução genérica

Sabe qual é a melhor parte de extrair os dados direto do Feed XML em vez das páginas HTML do site? Consistência e manutenção zero.

- As páginas mudam frequentemente, pois o autor muda de gerenciador, resolve colocar um design novo, troca as tags, muda os conteúdos de lugar... e você vai precisar atualizar o seu programa sempre que isso acontecer, para que ele continue funcionando no formato novo. É uma eterna corrida de gato e rato.
- O Feed XML é um formato padrão, que não muda. O `<title>` vai estar sempre lá, independente das mudanças no formato do site.

Caso você ainda não tenha visto a luz, vou dar a lanterna: sendo o XML um padrão, seu código vai funcionar para **qualquer** site que use feeds, não somente para o BR-Linux!

Muitos sites já usam feeds e a tendência é que a grande maioria adote o formato. Você pode acompanhar diretamente da linha de comando as novidades de seus blogs preferidos do WordPress/Blogger, sites de notícia nacionais e internacionais, listas de discussão, ferramentas comunitárias, entre outros. Basta descobrir o endereço do feed e fazer a extração de todos os `<title>` que encontrar.

Se já temos um programa que extrai as notícias do feed do BR-Linux, então teoricamente ele funcionará para qualquer feed, correto? Sim! Vamos fazer algumas modificações no `brlinux-3.sh` para que ele fique mais poderoso:

- O endereço do feed será passado como um argumento da linha de comando, possibilitando a obtenção das manchetes de qualquer feed.
- Com uma otimização na expressão regular que remove as tags HTML, já removeremos os espaços em branco também, em um único passo: "`*<[^>]*>`".
- Por falar em brancos, alguns feeds vêm alinhados com TABs em vez de espaços, então é prudente apagá-los com um `tr`.
- Podemos retirar o comando `head` do final e sempre mostrar todas as manchetes do feed, pois cada site mostra um número diferente de notícias.
- Com um código compacto de poucos comandos, esse programa pode muito bem virar uma função, que colocada no `~/.bashrc` transforma-se em um comando do sistema.

Função feed()

```
# Função feed: Extrai as manchetes mais recentes de um Feed
# Passe o endereço do feed como argumento
# Exemplo: feed http://br-linux.org/feed/
#
feed() {
    lynx -source "$1" | grep '<title>' | tr -d \\t |
```

```
        sed 's/ *<[^>]*>//g; s/&quot;/"/g; 1d'
}
```

Exemplos de execução:

```
$ feed http://br-linux.org/feed/
Instalando o modem Motorola SM56 "manualmente"
Vídeo: Novidades da próxima versão do Compiz Fusion
Fim do Kurumin? Ainda não. Alguém se habilita?
Opções de VoIP em plataformas livres
MPlayer promove inclusão digital de dispositivos com Windows
  Mobile
Vídeo-aulas de desenho de Mozart Couto - com GIMP
Gerenciamento de atualizações : uma solução simples e eficaz
I ECSL - I Encontro Cefetiano de Software Livre, em Sergipe
João Pessoa sedia o II Encontro de Software Livre da Paraíba
Permissões e propriedades de arquivos e diretórios no linux
BIOS, Barramentos e o Linux
Os primeiros testes com o "nosso" Classmate
A semana no BR-Linux: 11 anos de BR-Linux!
$ feed http://aurelio.wordpress.com/feed
Convença-me a te dar um livro
Show do No Use For A Name (NUFAN) em Curitiba
1.000 piazinhos vendidos
Brinque de CSS
Garoto-propaganda FNAC
Fiz trinta
Ida ao banco em Matinhos
Adivinhe o número de moedas
Minha palestra na PyConBrasil
12:00 piscando no micro-ondas
$
```

Não é bonito ver como em shell é possível fazer programinhas realmente úteis com poucas linhas de código? Você imaginava que com apenas uma linha conseguiria acessar um site, extrair as últimas notícias e mostrá-las em seu terminal? Isso é shell!

Mais uma vez cabe lembrar que, caso você enfrente problemas com a codificação do XML ser diferente da utilizada em seu sistema, use o comando `iconv` para transformar textos em UTF-8 para ISO-8859-1, ou vice-versa.

Após a publicação deste livro, o site BR-Linux poderá mudar seu formato, fazendo com que os programas demonstrados neste capítulo deixem de funcionar. Caso isso aconteça, consulte o site do livro em www.shellscript.com.br para obter versões atualizadas.

Capítulo 9

Arquivos de configuração

Você já usa opções de linha de comando em seu programa. Isso o torna amigável para usuários avançados que utilizam o terminal. Mas usuários que não têm tanta intimidade com a máquina ficarão intimidados. Aprenda a utilizar arquivos de configuração, permitindo que seus usuários apenas editem um arquivo de texto normal para modificar o comportamento de seu programa. Além de ficar mais amigável, gerará menos chamados de suporte para você.

Um arquivo de configuração é um arquivo que contém somente texto, que especifica como será o funcionamento do programa que o utiliza. Dentro dele pode ter:

- Informações sobre o usuário.
- Definições de limites de máximo e mínimo.
- Definições de parâmetros necessários para a execução do programa.
- Estados de chaves que ligam e desligam funcionalidades.
- Localização de arquivos externos.
- Definições de aparência do programa.
- Personalização de funcionamento.

O arquivo de configuração existe para facilitar a modificação de comportamento do programa, sem que precise recompilá-lo ou editar seu código-fonte. Quanto mais opções puderem ser configuráveis, maior a flexibilidade do usuário para alterar o programa conforme suas necessidades. Praticamente todo programa de médio e grande portes usa arquivos de configuração. É um canal de comunicação simples e eficiente entre o usuário e o programa.

Duas funcionalidades são necessárias a um programa para que ele possa utilizar arquivos de configuração: saber ler e gravar o arquivo no formato de configuração. Ler para obter e aplicar as preferências do usuário e gravá-lo para guardar possíveis mudanças de configuração feitas durante a execução do programa. O nome do programa, ou o trecho de código que cuida destas tarefas, é parser.

Sendo essa uma tarefa rotineira para a maioria dos programas, criou-se um formato padrão para o arquivo de configuração e cada linguagem já tem um módulo ou rotina pronta para ler e gravar nele, correto? Infelizmente, não. Em vez de se criar e seguir um padrão internacional e termos um parser genérico, cada autor reinventa a roda e cria seu próprio formato superespecial, 30% melhor que todos os outros. O resultado é caótico, uma fauna infindável de formatos de arquivos de configuração.

Assim, ao se deparar com a tarefa de ter que extrair ou gravar

informações de um arquivo de configuração de algum programa já existente, prepare-se. Primeiro será necessário conhecer em detalhes todas as regras do arquivo, para depois começar a codificar o parser. Pensa que acabou? Além do formato interno, a extensão utilizada no nome do arquivo também é algo que não segue um padrão. Temos `*.cfg`, `*.conf`, `*.cnf`, `*rc`, ...

Tour pelos formatos já existentes

Para que se tenha uma ideia do quão variada e exótica é a implementação de arquivos de configuração, aqui vai uma pequena amostra dos arquivos encontrados em um sistema Linux.

Nos formatos seguintes, a palavra "branco" refere-se a espaço em branco ou TAB.

Palavra-chave, brancos, valor

Arquivo: `httpd.conf`

Programa: Apache (Servidor HTTP)

Detalhes: Usa o caractere # para comentários, espaços em branco separam a palavra-chave de seu valor, que pode estar ou não entre aspas. Preferência por palavras-chave iniciadas por maiúsculas, SempreGrudadas.

```
#
# Timeout: The number of seconds before receives and sends time
  out.
#
Timeout 300

#
# ServerAdmin: Your address, where problems with the server should
  be
# e-mailed.  This address appears on some server-generated pages,
  such
# as error documents.
#
ServerAdmin root@localhost

#
```

```
# DocumentRoot: The directory out of which you will serve your
# documents. By default, all requests are taken from this
  directory,
# but symbolic links and aliases may be used to point to other
# locations.
#
DocumentRoot "/var/www/default"
```

Palavra-chave, brancos, valor opcional

Arquivo: `fvwmrc`

Programa: Fvwm (Gerenciador de janelas)

Detalhes: Parecido com o anterior, mas permite palavras-chave sozinhas, sem valores.

```
# Fonts
WindowFont -adobe-helvetica-medium-r-*-*-16-*-*-*-*-*-*-*
Font       -misc-fixed-medium-r-*-*-14-*-*-*-*-*-*-*
# Focus
AutoRaise 5000
SloppyFocus
# MWM Emulation
MWMBorders
MWMDecorHints
MWMFunctionHints
MWMHintOverride
# Miscellaneous Stuff
OpaqueMove     0
EdgeScroll     0 0
EdgeResistance 0 0
```

Palavra-chave, igual, valor opcional

Arquivo: `lilo.conf`

Programa: LILO (Gerenciador de inicialização)

Detalhes: Possui seções sequenciais (a ordem importa) iniciadas pela palavra `other`. Os espaços ao redor do sinal de igual são opcionais. Algumas palavras-chave não precisam de valores. Palavras-chave em minúsculas.

```
boot = /dev/hda
map = /boot/map
timeout = 500
prompt
  vga = normal
  read-only
other = /dev/hda1
  label = win
other = /dev/hda2
  label = linux
```

Palavra-chave, igual, valor

Arquivo: `my.cnf`

Programa: MySQL (Banco de dados)

Detalhes: Possui seções iniciadas por marcadores entre [colchetes] que identificam componentes do banco.

```
[mysql.server]
datadir=/var/lib/mysql
socket=/var/lib/mysql/mysql.sock
[mysqladmin]
socket=/var/lib/mysql/mysql.sock
[mysqld]
datadir=/var/lib/mysql
socket=/var/lib/mysql/mysql.sock
[mysql]
socket=/var/lib/mysql/mysql.sock
```

Palavra-chave, dois-pontos, valor

Arquivo: `lynx.cfg`

Programa: Lynx (Navegador em modo texto)

Detalhes: Formatação HTML embutida (`.h1`, `.h2`) para gerar documentação. Não pode haver brancos antes ou depois do dois-pontos. Preferência por palavras-chave em maiúsculas, separadas pelo sublinhado.

```
.h1 Interaction
```

```
# These settings control interaction of the user with lynx.
.h2 SCROLLBAR
# If SCROLLBAR is set TRUE, Lynx will show scrollbar on
  windows.  With
# mouse enabled, the scrollbar strip outside the bar is clickable,
  and
# scrolls the window by pages.  The appearance of the scrollbar
  can be
# changed from LYNX_LSS file:  define attributes scroll.bar,
# scroll.back (for the bar, and for the strip along which the
# scrollbar moves).
SCROLLBAR:FALSE
.h2 SCROLLBAR_ARROW
# If SCROLLBAR_ARROW is set TRUE, Lynx's scrollbar will have
  arrows at
# the ends.  With mouse enabled, the arrows are clickable, and
  scroll
# the window by 2 lines.  The appearance of the scrollbar arrows
  can
# be changed from LYNX_LSS file:  define attributes scroll.arrow,
# scroll.noarrow (for enabled-arrows, and disabled arrows).  An
  arrow
# is "disabled" if the bar is at this end of the strip.
SCROLLBAR_ARROW:TRUE
```

Palavra-chave, arroba opcional

Arquivo: `lpd.conf`

Programa: lpd (Servidor de impressão)

Detalhes: É preciso colocar um `@` no final da palavra-chave para desligá-la.

```
# Purpose: open the printer for reading and writing
#   default rw@ (FLAG off)
# Purpose: save job when an error
#   default save_on_error@ (FLAG off)
# Purpose: save job when done
#   default save_when_done@ (FLAG off)
# Purpose: send block of data, rather than individual files
#   default send_block_format@ (FLAG off)
# Purpose: send data files first, then control file
```

```
#   default send_data_first@ (FLAG off)
```

Componente, igual, valor numérico

Arquivo: `sysctl.conf`

Programa: sysctl (Configura parâmetros do kernel)

Detalhes: Palavra-chave tipo programação orientada a objeto, no formato módulo.componente. Utiliza os valores 0 (zero) e 1 para desligar e ligar uma opção. Comentários podem iniciar por # ou ponto e vírgula. Os brancos ao redor do sinal de igual são opcionais.

```
# If active, logs 'impossible' source addresses
net.ipv4.conf.all.log_martians = 1
# If active, rewrites outgoing IP packets if address changes
net.ipv4.ip_dynaddr = 1
# If active, ignores ICMP with broadcast destination
net.ipv4.icmp_echo_ignore_broadcasts = 1
# If active, accepts IP packets with source route info
net.ipv4.conf.all.accept_source_route = 0
# If active, ignores all ICMP requests
net.ipv4.icmp_echo_ignore_all = 0
```

Comando, brancos, palavra-chave, brancos, valor

Arquivo: `inputrc`

Programa: Biblioteca Readline (Comportamento da linha de comando)

Detalhes: A linha de configuração é um comando (`set`). As palavras-chave são separadas internamente pelo hífen e possuem valores padrão, não se pode colocar qualquer valor. Utiliza `on` e `off` para ligar e desligar funções.

```
set bell-style visible
set meta-flag on
set convert-meta off
set output-meta on
set editing-mode vi
set completion-ignore-case on
```

Comando, brancos, palavra-chave, brancos, igual, valor

Arquivo: `vimrc`

Programa: Vim (Editor de textos)

Detalhes: Utiliza as aspas como caractere de comentário, podendo aparecer também no meio da linha. Cada linha de configuração é um comando. Não são permitidos brancos ao redor do sinal de igual. Algumas palavras-chave possuem valores, outras não.

```
set aw              "AutoWrite: gravacao automatica a cada
  alteracao
set ai              "AutoIndent: identacao automatica
set ts=4            "TabStop: numero de caracteres de avanco do TAB
set report=0        "reporta acoes com linhas
set shm=filmnrwxt   "SHortMessages: encurta as mensagem do rodape
set et              "ExpandTab: troca TABs por espacos
retab               "converter os TABs ja existentes
```

Código Lisp

Arquivo: `mesa.conf`

Programa: Mesa (Biblioteca de gráficos 3D)

Detalhes: A configuração é feita em código em Lisp!

```
;; Default profile - should normally be an empty list of
;; configurations.
;;
(config-mesa mesa3.1beta1 ())
;; Really is an empty config.
;;
(config-mesa empty ())
;; Turn off some compliance for the sake of speed.
;;
(config-mesa quake2
(
  ;; Quake2 likes this extension, but it really hurts performance
 if
  ;; you don't also disable software fallbacks, below.  (And do
  ;; something else to stop the eye-space calculations too...)
  ;;
  (disable-extension GL_EXT_point_parameters)))
```

Ufa! Com essa pequena amostra já foi possível perceber o quão criativos são os programadores na hora de inventar seu próprio padrão de configuração.

Qual vai ser o seu formato?

Se sua necessidade é ler um arquivo de configuração já existente, não há escolha: seu parser deve reconhecer aquele formato. Mas se você tem um programa e quer fazer um arquivo de configuração para ele, qual formato utilizar? Você vai escolher um desses já vistos ou vai inventar um novo?

Calma, antes de começar a ter ideias vanguardistas, vamos analisar o que vimos até aqui para tentar encontrar os pontos básicos do assunto. Um arquivo de configuração é formado por quatro componentes:

Componente	Descrição
Palavra-chave	Define qual característica do programa está em foco.
Valor	Define qual será o comportamento dessa característica.
Separador	Caractere que separa a palavra-chave do seu valor.
Comentário	Caractere que faz uma linha ser ignorada.

Dos exemplos apresentados, podemos constatar que:

- **Palavra-chave:** É formada apenas de letras e números, tendo como opcionais o hífen e o sublinhado para separar as palavras que formam o termo. Exemplos: `use-color`, `WindowFont`, `EXEC_PATH`.
- **Valor:** Pode ser numérico, indicando uma quantidade. Também pode ser do tipo liga/desliga (0/1, on/off, true/false) ou ainda um texto qualquer, colocado ou não entre aspas. Exemplos: `123`, `OFF`, `"~/mail"`.
- **Separador:** É necessário para que a palavra-chave e o valor nãofiquemgrudados. Geralmente usa-se o sinal de igual, podendo ou não ter espaços ou TABs ao redor. Alguns omitem o sinal e usam o próprio branco como separador, facilitando a leitura. Exemplos: *`chave=valor`*, *`chave valor`*.
- **Comentário:** Útil para desativar temporariamente uma opção ou colocar textos explicativos. É quase uma unanimidade usar o caractere `#`. Poucos transgressores usam alternativas como aspas, ponto e vírgula

ou `//`. Além do caractere há diferenças na aplicação, alguns podendo ser colocados somente no início da linha, outros no meio.

Além destes quatro componentes básicos, alguns arquivos de configuração mais complexos são divididos em seções, pois as configurações são dependentes de contexto. O formato preferido é colocar colchetes ao redor. Exemplo: `[init]`.

O denominador comum

Como não existe um padrão estabelecido que se possa seguir, você é livre para escolher o formato que preferir. Mas em vez de criar algo totalmente novo, ignorando a experiência anterior de outros programadores e usuários, o melhor é encontrar um denominador comum. Um formato simples que tenha as características mais comuns e marcantes dos arquivos já existentes.

Este será o formato utilizado nos exemplos seguintes e no restante do aprendizado:

```
# comentário
chave valor
```

O `#` já é um clássico para comentários em arquivos de configuração, além de ser utilizado pelo próprio shell, então não há o que discutir. Usar brancos como separador é o mais intuitivo, não há um caractere especial para se lembrar, basta separar visualmente e pronto.

Para deixar o formato mais flexível e à prova de erros, podemos permitir brancos opcionais no início e no final da linha, que serão ignorados no processamento, bem como vários brancos entre a chave e o valor. Este é basicamente o formato do arquivo de configuração do Apache (`httpd.conf`), que já é amigo de longa data dos administradores de sistemas.

Especificação do formato

- As linhas de comentário iniciam por `#`, podendo ser precedido por brancos.
- A chave fica no início da linha, podendo ser precedida por brancos.

- O valor fica após a chave, separado por um ou mais brancos.
- A chave pode ser composta por letras e números somente.
- O valor pode ser qualquer texto (string) ou número, sem aspas.
- Por branco, entende-se: espaço em branco ou TAB.

Sintaxe:

```
<brancos opcionais> # comentário <brancos opcionais>
<brancos opcionais> chave <um ou mais brancos> valor <brancos
  opcionais>
```

Expressões regulares:

```
^ \s* # .* \s* $
^ \s* [A-Za-z0-9]+ \s+ .* \s* $
```

Agora que já temos um formato definido, podemos começar a programar, não?

Codificação do parser

Conforme já visto, o parser é o programa ou código que gerencia arquivos de configuração. Ele sabe como ler e extrair os dados do arquivo, respeitando seu formato. Se necessário, o parser também pode alterar ou reescrever o arquivo.

Vamos escrever um parser para o nosso formato récem-descrito. Começaremos com um exemplo simples, mas que ilustra a maioria dos aspectos envolvidos na escrita de um parser. O primeiro passo é escrever um arquivo de configuração pequeno para ser usado nos testes.

⌨ Arquivo de configuração de testes

```
#
## mensagem.conf - O arquivo de configuração do mensagem.sh
#
### Opções Liga/Desliga
#
# Use ON/OFF para ligar e desligar as opções
UsarCores ON
```

```
### Opções Numéricas
#
#   COR         | LETRA | FUNDO
#   ------------+-------+-------
#   preto       |  30   |  40
#   vermelho    |  31   |  41
#   verde       |  32   |  42
#   amarelo     |  33   |  43
#   azul        |  34   |  44
#   rosa        |  35   |  45
#   ciano       |  36   |  46
#   branco      |  37   |  47
#
CorFundo 44
CorLetra 37

### Opções com Strings
#
Mensagem    Modo texto é legal
```

Este arquivo define o comportamento do programa `mensagem.sh`, o qual serve para mostrar uma mensagem na tela. Supondo que o programa seja executado com tal configuração, este será o resultado:

```
$ ls
mensagem.conf       mensagem.sh
$ ./mensagem.sh
Modo texto é legal
$
```

O texto mostrado na tela tem letras brancas em fundo azul, conforme informado nas opções `CorLetra` e `CorFundo` no arquivo de configuração. Funciona assim: o script lê o arquivo de configuração, identifica as chaves e aplica os valores encontrados. Estes valores definem qual a mensagem a ser mostrada e em quais cores ela vai aparecer.

Antes de escrever o código do programa, cabe uma análise prévia do que ele precisará ter. Este é um esqueleto de nosso parser:

- Definir variáveis com os valores padrão de cada opção.
- Ler linha a linha (loop) o arquivo de configuração.

- Se encontrar linhas em branco ou comentários, ignorar.
- Processar as linhas com palavras-chave, definindo as variáveis do programa com os valores encontrados.

- Configurações carregadas, basta processá-las e mostrar a mensagem na tela.

É um programa simples, mas é bom ter essa visão geral para poder codificar cada pedaço de maneira independente, sem misturar os conceitos.

Parser passo a passo

Se ainda não estiver com seu editor de textos preferido aberto, faça isso agora! Digite o código à medida que vai lendo, para entender e fixar cada técnica. Se não puder usar um computador neste momento, não se preocupe, basta ler o código com bastante atenção.

Primeiro crie o arquivo de configuração `mensagem.conf` com o conteúdo já visto. Resuma os comentários se preferir, mas não os exclua! Lembre-se de que o parser também deve saber o que fazer com comentários. Em seguida, crie no mesmo diretório o arquivo `mensagem.sh`, que será nosso parser. Seu conteúdo começa agora:

```
#!/bin/bash
#
# mensagem.sh
# Mostra uma mensagem colorida na tela, lendo os
# dados de um arquivo de configuração externo.
#
# 2006-10-31 Fulano da Silva
```

Regra de ouro: o primeiro passo é o cabeçalho, sempre. Descreva o que o programa fará. Se você não tem uma visão geral do programa antes de começar a codificá-lo, vai codificar o quê?

```
CONFIG="mensagem.conf"            # Arquivo de configuração

# Configurações (serão lidas do $CONFIG)
USAR_CORES=0                      # config: UsarCores
COR_LETRA=                        # config: CorLetra
```

```
COR_FUNDO=                          # config: CorFundo
MENSAGEM='Mensagem padrão'          # config: Mensagem
```

Nada de novo até aqui, certo? A variável `CONFIG` traz o nome do arquivo de configuração e as outras variáveis guardarão as opções encontradas. Elas já foram inicializadas com valores padrão, então caso o arquivo de configuração esteja vazio, será mostrada a frase "Mensagem padrão", sem cores.

O próximo trecho é o loop para ler o arquivo de configuração linha a linha. Lembre-se de não cair na pegadinha do `cat arquivo | while` (explicado no tópico "Dicas preciosas"), então faremos do jeito certo, usando redirecionamento de entrada.

```
# Loop para ler linha a linha a configuração, guardando em $LINHA
while read LINHA; do
    # Comando vazio, que não faz nada
    :
done < "$CONFIG"
```

Note que a variável que guarda o arquivo de configuração foi colocada entre aspas: `"$CONFIG"`. Isso evita problemas caso o nome do arquivo de configuração (ou algum de seus diretórios) possuam espaços em branco no nome. Acostume-se a sempre colocar aspas. Sempre.

Deixemos o miolo desse loop para ser preenchido depois e vamos seguir adiante. Só lembre-se de que é neste loop que toda a configuração será lida e guardada nas variáveis. O próximo passo já é o final do programa, onde só resta checar o valor das variáveis e mostrar a mensagem na tela.

```
# Configurações lidas, mostre a mensagem
if [ $USAR_CORES -eq 1 ]; then
    # Mostrar mensagem colorida
    # Exemplo: \033[40;32mOlá\033[m
    echo -e "\033[$COR_FUNDO;${COR_LETRA}m$MENSAGEM\033[m"
else
    # Não usar cores
    echo "$MENSAGEM"
fi
```

E chegamos ao fim do programa. Se a chave das cores estiver ligada (valor 1), será usado o echo vitaminado, com os caracteres de controle e as

variáveis `COR_FUNDO` e `COR_LETRA`, fazendo a mensagem aparecer colorida na tela. Caso contrário, mostra o texto sem cores.

A variável `${COR_LETRA}` foi colocada entre chaves para evitar confusão com a letra "m" seguinte, que faz parte do caractere de controle `ESC[m`. Sem as chaves o shell tentaria expandir a variável `$COR_LETRAm`, que não existe.

Perceba que no arquivo de configuração a palavra-chave `UsarCores` só aceita os valores ON ou OFF, mas aqui no programa a variável `$USAR_CORES` tem os valores 0 e 1. Essa tradução de valores é feita dentro do loop (logo voltaremos a ele) e é uma das vantagens de se usar um arquivo de configuração externo. Você pode deixar o formato flexível, permitindo que o usuário escreva ON/OFF, ligado/desligado, true/false e no parser é que tais valores são normalizados e convertidos em números.

Experimente rodar o programa assim como está. O que vai acontecer? No início, as variáveis vão ser definidas com seus valores padrão. Depois, o loop vai ler todo o arquivo de configuração, mas como ainda não há códigos dentro do while, nada será processado. Terminado o loop será mostrada a mensagem na tela, usando as configurações padrão:

```
$ ./mensagem.sh
Mensagem padrão
$
```

Tudo certo, não há erros de sintaxe e tudo funcionou como deveria. Este é o comportamento do programa quando o arquivo de configuração estiver vazio.

E por que deixar o miolo do loop por último? Esta é uma técnica muito utilizada em programação, você nunca senta e escreve o programa todo de cabo a rabo, sem fazer teste algum. O certo é escrever um pouco, salvar e executar. Caso apareça algum erro, ficará fácil identificá-lo, pois somente poucas linhas foram alteradas. Se não houver erros, edita mais um pouco, salva e testa novamente. E assim vai, em pequenos passos, sempre no controle da situação. Leia este parágrafo novamente.

Conseguiu sair do loop infinito do parágrafo anterior? :) A ênfase é porque essa técnica de escrever e testar em ciclos curtos é muito importante, acostume-se com ela. Deixe que isso se torne a sua prática normal de programação e evite dores de cabeça.

Voltando ao loop, agora é a hora de fazer nosso programa entender-se com o arquivo de configuração.

```
# Loop para ler linha a linha a configuração, guardando em $LINHA
while read LINHA; do

    # DICA:
    # Basta referenciar o $LINHA sem aspas para que todos
    # os brancos do início e fim da linha sejam removidos,
    # e os espaços e TABs entre a chave e o valor sejam
    # convertidos para apenas um espaço normal.
    #
    # Descomente as linhas seguintes para testar
    echo Com aspas: "$LINHA"
    echo Sem aspas: $LINHA

    # Ignorando as linhas de comentário
    [ "$(echo $LINHA | cut -c1)" = '#' ] && continue

    # Ignorando as linhas em branco
    [ "$LINHA" ] || continue

    # Quem sobrou?
    echo +++ $LINHA

done < "$CONFIG"
```

Como a dica nos comentários descreve, a expansão de variáveis do shell sem as aspas remove todos os brancos indesejados, facilitando muito o trabalho do parser. Não é preciso preocupação se o usuário colocou espaço em branco ou TAB, ou se foram vários espaços. O shell limpa tudo, deixando apenas um espaço entre a chave e o valor, assim como na linha de comando:

```
$ echo     um dois  três   quatro    cinco     seis
um dois três quatro cinco seis
$
```

Em seguida temos o código que ignora os comentários. É feito o teste se o primeiro caractere (`cut -c1`) da linha é o `#`. Se for, o comando `continue` é chamado e o loop passa para a próxima linha. Depois vem o teste das linhas em branco, que requer atenção. É testado o conteúdo da variável, entre aspas para não ter problema quando ela for vazia. Se a variável **não**

possuir conteúdo (`||`) , o continue é chamado.

O comando `[ "$LINHA" ] || continue` pode ser lido assim: o negócio é o seguinte, ou a `$LINHA` tem conteúdo, ou nada feito, passe para a próxima. Isso me lembra a frase "skate or die", tão famosa nos anos 80, que tem sua versão nerd no "skate || die".

Estes dois testes são como um firewall (barreira) que só deixa passar as linhas desejadas, que são, em nosso caso, as linhas com alguma configuração. Linhas em branco e com comentários não passam pela barreira. Caso no futuro seja necessário algum outro teste, basta adicionar aqui e o firewall ficará mais exigente.

O próximo comando é um `echo` que está mostrando quais foram as linhas que passaram pelo firewall. É colocada uma marcação "+++" no início da linha para ficar mais fácil a sua visualização em meio às outras mensagens. Opa, já foram feitas muitas alterações, hora de testar novamente:

```
$ ./mensagem.sh | tail
Com aspas:
Sem aspas:
Com aspas: ### Opções com Strings
Sem aspas: ### Opções com Strings
Com aspas: #
Sem aspas: #
Com aspas: Mensagem     Modo texto é legal
Sem aspas: Mensagem Modo texto é legal
+++ Mensagem Modo texto é legal
Mensagem padrão
$
```

Estas são as últimas linhas (`tail`) do resultado da execução. O programa continua mostrando a mensagem padrão no final, já que ainda não interpretamos as configurações. Pelas mensagens de com e sem aspas, é possível constatar o efeito do shell remover os brancos. E parece que nosso firewall está funcionando, pois destas linhas a única que passou (e ganhou o prefixo +++) foi a configuração Mensagem. Conferindo:

```
$ ./mensagem.sh | fgrep +++
+++ UsarCores ON
+++ CorFundo 44
```

```
+++ CorLetra 37
+++ Mensagem Modo texto é legal
$
```

Certo, até aqui está tudo bem, somente as linhas corretas estão chegando no final do loop. As linhas com os echos informativos (debug) já podem ser comentadas ou apagadas.

Agora começa a diversão. É hora de ler as chaves e seus valores nestas linhas que passaram pelo firewall. Lembrando que apenas um espaço os separa, em outras linguagens poderia ser feito um split(' ', 1). Mas o shell nos dá outras alternativas mais interessantes, acompanhe:

```
# Guardando cada palavra da linha em $1, $2, $3, ...
# "Suzy é metaleira" fica  $1=Suzy  $2=é  $3=metaleira
set - $LINHA
# Extraindo os dados
# Primeiro vem a chave, o resto é o valor
chave=$1
shift
valor=$*

# Conferindo se está tudo certo
echo "+++ $chave --> $valor"
```

O comando `set -` atribui às variáveis posicionais (`$1`, `$2`, ...) todas as palavras que lhe forem passadas. Como a chave é sempre a primeira palavra da linha em nosso arquivo de configuração, com este comando ela foi guardada em `$1`. O restante (seu valor) ficou em `$2`. Caso o valor possua mais de uma palavra, `$3`, `$4` e seguintes são usados.

O que é feito a seguir é guardar a chave na variável `$chave`, depois usar o comando `shift` para remover o `$1`. O que sobrar é o valor. Na dúvida, se ele é composto por uma ou mais palavras, basta usar o `$*` para pegar todas de uma vez. Execute um exemplo simples na linha de comando para conferir a técnica:

```
$ set - um dois três quatro
$ echo $1
um
$ echo $*
um dois três quatro
```

```
$ shift
$ echo $1
dois
$ echo $*
dois três quatro
$
```

Voltando ao código, no final temos outro `echo` informativo com o prefixo +++ para mostrar cada chave encontrada e seu valor. Executando o programa, percebe-se que estamos no caminho certo, com chaves e valores separados corretamente:

```
$ ./mensagem.sh
+++ UsarCores --> ON
+++ CorFundo --> 44
+++ CorLetra --> 37
+++ Mensagem --> Modo texto é legal
Mensagem padrão
$
```

Para finalizar o loop do parser, só resta aplicar a configuração encontrada. Os dados já estão armazenados em `$chave` e `$valor`, basta processá-los. Leia o código e tente interpretá-lo, não está difícil:

```
    # Processando as configurações encontradas
    case "$chave" in

    UsarCores)
        [ "$valor" = 'ON' ] && USAR_CORES=1
        ;;
    CorFundo)
        COR_FUNDO=$valor
        ;;
    CorLetra)
        COR_LETRA=$valor
        ;;
    Mensagem)
        MENSAGEM=$valor
        ;;
    *)
        echo "Erro no arquivo de configuração"
```

```
        echo "Opção desconhecida '$chave'"
        exit 1
        ;;
    esac
```

Tranquilo. Para cada chave, o valor é guardado na variável correspondente do parser. Note que `$USAR_CORES` só vai ser ligada quando a chave `UsarCores` possuir o valor `ON`. Para qualquer outro valor ela continua em seu estado padrão, desligada. O último bloco do comando `case` será executado caso uma chave com nome desconhecido seja encontrada.

Ao encontrar uma configuração inválida, você pode simplesmente ignorá-la ou mostrar um erro na tela, forçando que o usuário arrume as configurações. Escolha o que for melhor levando em conta o tipo de seu programa e o perfil de seus usuários.

E pronto, está feito o parser. Cruze os dedos, chegou a hora do teste final:

```
$ ./mensagem.sh
Modo texto é legal
$
```

Funciona! A mensagem mostrada é aquela indicada no arquivo de configuração, assim como as cores, fundo azul e letra branca. Mude os valores no `mensagem.conf` e execute o programa novamente para conferir que ele realmente está lendo as configurações. Bacana, não? Vejamos como ficou o programa completo:

mensagem.sh

```
1       #!/bin/bash
2       #
3       # mensagem.sh
4       # Mostra uma mensagem colorida na tela, lendo os
5       # dados de um arquivo de configuração externo.
6       #
7       # 2006-10-31 Fulano da Silva
8
9       CONFIG="mensagem.conf"            # Arquivo de
  configuração
10
```

```
11      # Configurações (serão lidas do $CONFIG)
12      USAR_CORES=0                  # config: UsarCores
13      COR_LETRA=                    # config: CorLetra
14      COR_FUNDO=                    # config: CorFundo
15      MENSAGEM='Mensagem padrão'    # config: Mensagem
16
17      # Loop para ler linha a linha a configuração, guardando
 em $LINHA
18      while read LINHA; do
19
20          # DICA:
21          # Basta referenciar o $LINHA sem aspas para que todos
22          # os brancos do início e fim da linha sejam
 removidos,
23          # e os espaços e TABs entre a chave e o valor sejam
24          # convertidos para apenas um espaço normal.
25          #
26          # Descomente as linhas seguintes para testar
27          # echo Com aspas: "$LINHA"
28          # echo Sem aspas: $LINHA
29
30          # Ignorando as linhas de comentário
31          [ "$(echo $LINHA | cut -c1)" = '#' ] && continue
32
33          # Ignorando as linhas em branco
34          [ "$LINHA" ] || continue
35
36          # Guardando cada palavra da linha em $1, $2, $3, ...
37          # "Suzy é metaleira"
 fica  $1=Suzy  $2=é  $3=metaleira
38          set - $LINHA
39
40          # Extraindo os dados
41          # Primeiro vem a chave, o resto é o valor
42          chave=$1
43          shift
44          valor=$*
45
46          # Conferindo se está tudo certo
47          #echo "+++ $chave --> $valor"
```

```
48
49          # Processando as configurações encontradas
50          case "$chave" in
51
52          UsarCores)
53              [ "$valor" = 'ON' ] && USAR_CORES=1
54              ;;
55          CorFundo)
56              COR_FUNDO=$valor
57              ;;
58          CorLetra)
59              COR_LETRA=$valor
60              ;;
61          Mensagem)
62              MENSAGEM=$valor
63              ;;
64          *)
65              echo "Erro no arquivo de configuração"
66              echo "Opção desconhecida '$chave'"
67              exit 1
68              ;;
69          esac
70
71      done < "$CONFIG"
72
73      # Configurações lidas, mostre a mensagem
74
75      if [ $USAR_CORES -eq 1 ]; then
76          # Mostrar mensagem colorida
77          # Exemplo: \033[40;32mOlá\033[m
78          echo -e
 "\033[$COR_FUNDO;${COR_LETRA}m$MENSAGEM\033[m"
79      else
80          # Não usar cores
81          echo "$MENSAGEM"
82      fi
```

Melhorias no parser

Nessa primeira versão funcional do parser temos mais de 80 linhas, codificadas com calma e fazendo cada parte de uma vez. Programando desta maneira é possível fazer códigos realmente complexos sem muito estresse. Quanto menos linhas forem alteradas entre cada teste, mais controle sobre possíveis problemas você terá.

Mas como toda versão inicial tem suas limitações, é possível fazer algumas melhorias para tornar o parser mais robusto e flexível. Tenha em mente que sempre que utilizar um arquivo de configuração, você está dando ao usuário o poder de alterar o comportamento de seu programa. De brinde, o usuário ainda ganha o poder de **quebrar** o programa. Ah, e como eles gostam de fazer isso :)

```
USARCORES on
CorFundo 99
CorLetra Timão
Mensagem
```

Que tal esta configuração? Cada linha revela uma falha diferente no parser. Você consegue identificá-las?

1. Chave em maiúsculas e valor em minúsculas.
2. Valor numérico fora da faixa permitida (30-37, 40-47).
3. Valor é um texto quando deveria ser numérico.
4. Está faltando o valor.

Como esse nosso parser é furado! Nenhuma destas condições está prevista no código. Assim, somente editando o arquivo de configuração e colocando valores não esperados o usuário descobre a fragilidade do programa.

Bem-vindo à vida real da programação! Quanto mais flexibilidade, mais brechas podem aparecer. Ao dar a facilidade do usuário poder configurar seu programa, é preciso prever o inesperado. Tire a poeira daquela sua bola de cristal trazida do Paraguai...

Felizmente com pequenas alterações é possível corrigir todos estes furos. Mas basta chegar outro usuário mais criativo e novos furos aparecerão. E assim é a rotina do programador: arrumar os problemas (bugs) que aparecem à medida que mais pessoas utilizem seus programas e se sobrar

tempo, implementar alguma funcionalidade nova ;)

O primeiro problema é um clássico das mais remotas eras*: maiúsculas e minúsculas. Sempre que for possível e cabível, ao comparar valores de texto faça de maneira que funcione independente da caixa da letra. Uma prática comum é primeiro converter todas as letras para minúsculas, depois fazer a verificação.

* No princípio criou Deus o $CEU e a $TERRA. Ambos foram declarados com valor inicial vazio. No terceiro dia Deus definiu um valor fazendo terra="vegetais". Salvou, executou e não funcionou. E viu Deus que isso não era bom. Deus, pois, usou o `set -x` e viu que o problema era a diferença entre maiúsculas e minúsculas. E disse Deus: Sejam respeitadas as diferenças. E assim foi. (Gn 1:1-11)

Em nosso parser, devemos mudar a linha 42 para guardar o nome da chave sempre em minúsculas:

```
chave=$(echo $1 | tr A-Z a-z)
```

E dentro do case, trocar todos os identificadores para minúsculas:

```
case "$chave" in
usarcores)
    ...
corfundo)
    ...
corletra)
    ...
mensagem)
    ...
```

Resolvido o problema com as chaves. Internamente elas serão sempre minúsculas, independente de como tenham sido digitadas no arquivo de configuração. O usuário pode colocar `usarCores`, `USARCORES` ou até `UsArCoReS`, todos funcionarão. De maneira similar pode-se aceitar ON/on/On/oN como valor para ligar a chave na linha 53:

```
usarcores)
    [ "$(echo $valor | tr A-Z a-z)" = 'on' ] && USAR_CORES=1
    ;;
```

Furo número um tapado! O furo número dois vai ser resolvido do jeito mais fácil: será ignorado. A faixa válida para os números que indicam as cores é bem restrita com apenas 16 possibilidades (do 30 ao 37, do 40 ao

47) e o usuário digitou 99. Porém, antes de sair fazendo códigos para verificar o número informado, que tal testar o que acontece quando usamos um números desses?

```
$ echo -e '\033[99;99mTexto\033[m'
Texto
$ echo -e '\033[9999999;9999999mTexto\033[m'
Texto
$
```

Não há problema, o texto é simplesmente mostrado sem cores. Então não é preciso alterar o programa. Gostou dessa? Eu também.

No terceiro furo era para digitar um número e o usuário colocou um texto no lugar. Ah, esses usuários criativos... Uma das possibilidades é verificar se o valor é um número, e caso não seja, ignorá-lo. Uma outra alternativa que dispensa a verificação é simplesmente remover todos os caracteres que não sejam numéricos. Assim, se o usuário colocou algo como "cor44", ainda é possível aproveitar o número.

```
$ echo cor44 | tr -d -c 0-9
44
$
```

No quarto furo a chave Mensagem está vazia e, ao rodar o programa, é mostrada uma linha em branco na tela. Isso pode ser considerado um problema ou um comportamento esperado, é uma questão conceitual. Uma mensagem vazia significa que o usuário quer que apareça uma linha em branco? Ou essa configuração deve ser considerada inválida e a mensagem padrão deve ser mostrada? Não há certo ou errado, cabe ao programador decidir o qual o comportamento mais natural. A famosa lei da menor surpresa.

Em nosso caso, vamos modificar o programa para mostrar a mensagem padrão caso a mensagem do usuário esteja vazia. Como a variável interna `$MENSAGEM` já é definida com a mensagem padrão no início do programa, basta usar a lógica: só gravar a mensagem do usuário caso ela não seja vazia (linha 62):

```
[ "$valor" ] && MENSAGEM=$valor
```

	Traduzindo: Se houver valor, grave-o na variável `MENSAGEM`.

Fechado o último furo, não há mais nada a alterar. Por enquanto... Veja como ficou o código do parser, com as alterações em destaque:

mensagem.sh (melhorado)

```
1       #!/bin/bash
2       #
3       # mensagem.sh
4       # Mostra uma mensagem colorida na tela, lendo os
5       # dados de um arquivo de configuração externo.
6       #
7       # 2006-10-31 Fulano da Silva
8
9       CONFIG="mensagem.conf"            # Arquivo de
 configuração
10
11       # Configurações (serão lidas do $CONFIG)
12       USAR_CORES=0                       # config: UsarCores
13       COR_LETRA=                         # config: CorLetra
14       COR_FUNDO=                         # config: CorFundo
15       MENSAGEM='Mensagem padrão'         # config: Mensagem
16
17       # Loop para ler linha a linha a configuração, guardando
 em $LINHA
18       while read LINHA; do
19
20           # DICA:
21           # Basta referenciar o $LINHA sem aspas para que todos
22           # os brancos do início e fim da linha sejam
 removidos,
23           # e os espaços e TABs entre a chave e o valor sejam
24           # convertidos para apenas um espaço normal.
25           #
26           # Descomente as linhas seguintes para testar
27           #echo Com aspas: "$LINHA"
28           #echo Sem aspas: $LINHA
29
30           # Ignorando as linhas de comentário
31           [ "$(echo $LINHA | cut -c1)" = '#' ] && continue
```

```
32
33          # Ignorando as linhas em branco
34          [ "$LINHA" ] || continue
35
36          # Guardando cada palavra da linha em $1, $2, $3, ...
37          # "Suzy é metaleira"
 fica  $1=Suzy  $2=é  $3=metaleira
38          set - $LINHA
39
40          # Extraindo os dados
41          # Primeiro vem a chave, o resto é o valor
42          chave=$(echo $1 | tr A-Z a-z)
43          shift
44          valor=$*
45
46          # Conferindo se está tudo certo
47          #echo "+++ $chave --> $valor"
48
49          # Processando as configurações encontradas
50          case "$chave" in
51
52          usarcores)
53              [ "$(echo $valor | tr A-Z a-z)" = 'on' ] &&
 USAR_CORES=1
54              ;;
55          corfundo)
56              COR_FUNDO=$(echo $valor | tr -d -c 0-9) # só
 números
57              ;;
58          corletra)
59              COR_LETRA=$(echo $valor | tr -d -c 0-9) # só
 números
60              ;;
61          mensagem)
62              [ "$valor" ] && MENSAGEM=$valor
63              ;;
64          *)
65              echo "Erro no arquivo de configuração"
66              echo "Opção desconhecida '$chave'"
67              exit 1
```

```
68                ;;
69            esac
70
71        done < "$CONFIG"
72
73        # Configurações lidas, mostre a mensagem
74
75        if [ $USAR_CORES -eq 1 ]; then
76            # Mostrar mensagem colorida
77            # Exemplo: \033[40;32mOlá\033[m
78            echo -e
  "\033[$COR_FUNDO;${COR_LETRA}m$MENSAGEM\033[m"
79        else
80            # Não usar cores
81            echo "$MENSAGEM"
82        fi
```

Com as linhas modificadas em negrito fica fácil de entender as mudanças. Sabe o comando `diff`? É ele quem mostra esse tipo de informação. Veja a comparação:

```
$ diff -u mensagem.sh mensagem-melhorado.sh
--- mensagem.sh 2006-08-31 21:27:21.000000000 -0300
+++ mensagem-melhorado.sh   2006-08-31 23:37:08.000000000 -0300
@@ -39,7 +39,7 @@
     # Extraindo os dados
     # Primeiro vem a chave, o resto é o valor
-    chave=$1
+    chave=$(echo $1 | tr A-Z a-z)
     shift
     valor=$*
@@ -49,17 +49,17 @@
     # Processando as configurações encontradas
     case "$chave" in

-    UsarCores)
-        [ "$valor" = 'ON' ] && USAR_CORES=1
+    usarcores)
+        [ "$(echo $valor | tr A-Z a-z)" = 'on' ] && USAR_CORES=1
         ;;
```

```
-    CorFundo)
-        COR_FUNDO=$valor
+    corfundo)
+        COR_FUNDO=$(echo $valor | tr -d -c 0-9) # só números
        ;;
-    CorLetra)
-        COR_LETRA=$valor
+    corletra)
+        COR_LETRA=$(echo $valor | tr -d -c 0-9) # só números
        ;;
-    Mensagem)
-        MENSAGEM=$valor
+    mensagem)
+        [ "$valor" ] && MENSAGEM=$valor
        ;;
    *)
        echo "Erro no arquivo de configuração"
$
```

As linhas iniciadas com o sinal de menos foram trocadas pelas linhas iniciadas pelo sinal de mais. As outras linhas são iguais em ambos, só foram listadas para revelar o contexto. Acostume-se a salvar o arquivo com nomes diferentes a cada rodada de mudanças. Depois, com o `diff`, você pode acompanhar com precisão todo o histórico de alterações e eventualmente identificar falhas que ocorreram no processo.

Se você usa o editor de textos Vim, é possível ver esta saída do comando `diff` em cores, facilitando muito a leitura. Basta redirecionar a saída para o editor e fazê-lo ler os dados da entrada padrão (STDIN), usando o hífen como nome de arquivo, assim: `diff -u arquivo1 arquivo2 | vi -`

Evolução para um parser genérico

Tudo bem, o `mensagem.sh` funciona. Agora imagine que você queira usar um arquivo de configuração com este mesmo formato em algum outro programa. Ele também precisará de um parser para ler a configuração. Certo? Certo. Então é só copiar e colar o loop do mensagem.sh nele e adaptar o que precisar, certo?

Este é um daqueles momentos mágicos em que se diferenciam os

profissionais dos amadores, os programadores dos scripteiros. Você vai lá e copia o código do parser no programa novo. Faz as adaptações necessárias e deixa funcionando. Como você gostou dessa facilidade de ter arquivos de configuração, copia este código para vários outros programas. Ao usar um deles, você descobre um problema no parser e o corrige. Você vai lembrar de alterar todos os outros programas que também utilizam esse código? Dificilmente.

Neste caso, o melhor é escrever um programa especialista, cuja única função seja cuidar de arquivos de configuração. Então cada programa que precisar de configurações vai utilizá-lo. Vamos chamá-lo de `parser.sh`. Ele será um programa autônomo, com suas próprias versões e desenvolvimento. Assim as tarefas de gerenciamento de configuração ficam centralizadas e cada melhoria que for implementada será automaticamente aproveitada por todos os outros programas que o utilizam.

Características de um parser genérico

O parser precisa ser genérico, funcionando com qualquer combinação de chaves e valores, pois cada programa terá chaves diferentes para serem configuradas. Ele também precisará acessar os arquivos onde quer que eles estejam, não necessariamente no mesmo diretório. E, por fim, ele precisará se comunicar com o programa que o chamar, recebendo o nome do arquivo a ser lido e depois entregando os resultados.

Exemplo de requisições:

- Parser, dê-me o valor da chave `SomenteLeitura` do arquivo `permissoes.conf`
- Parser, dê-me todas as configurações que encontrar no `global.conf`

Para que este esquema funcione, há algumas alternativas a considerar:

1. O parser pode ser uma biblioteca, um conjunto de funções que devem ser incluídas na shell atual e conforme chamadas, retornam os valores requisitados.

```
$ CONFIG=mensagem.conf
$ source parser.sh
```

```
$ echo $(PegaValor Mensagem)
```

2. O parser pode ser um programa, que deve ser incluído na shell atual e definirá variáveis de acordo com as configurações encontradas.

```
$ CONFIG=mensagem.conf
$ source parser.sh
$ echo $MENSAGEM
```

3. O parser pode ser um programa, que recebe um arquivo como parâmetro e retorna na saída padrão toda a configuração no formato *chave=valor* do shell, pronto para ser usado com o comando `eval`.

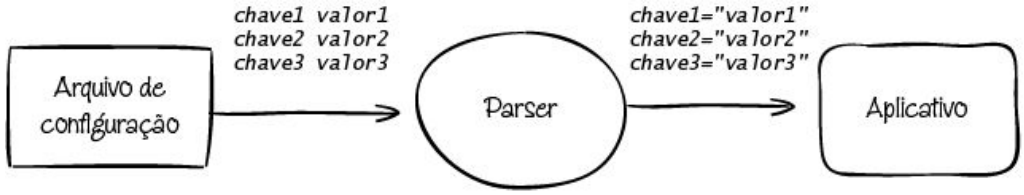
```
$ eval $(./parser.sh mensagem.conf)
$ echo $MENSAGEM
```

A primeira maneira é a mais profissional e mais parecida com a programação em linguagens tradicionais. Mas por ora vamos encará-la como um ideal a atingir, um refinamento futuro. A segunda maneira é intermediária e a terceira é a mais simples, o parser conversor. Simples é bom, vamos implementá-la.

Como curiosidade, sabia que é possível fazer um parser que funcione nas três maneiras sugeridas, utilizando o mesmo código? Pois é, e você achava que shell era só para fazer scripts...

Parser do tipo conversor

Este parser é chamado do tipo conversor, pois é exatamente o que ele faz. É um programa autônomo que funciona via opções de linha de comando, transformando o texto da configuração em variáveis do shell. A comunicação entre o parser e o programa que o utilizará dá-se da seguinte maneira:



Funcionamento do parser de configuração tipo conversor

Ou seja, o parser lê as configurações no formato *chave valor* e as converte para o formato de definição de variáveis do shell *chave="valor"*,

mandando o resultado para a saída padrão. Tomando como exemplo o arquivo de configuração `mensagem.conf` já visto, este seria o resultado:

```
$ ./parser.sh mensagem.conf
USARCORES="ON"
CORFUNDO="44"
CORLETRA="37"
MENSAGEM="Modo texto é legal"
$
```

Note que os nomes dos campos foram convertidos para maiúsculas, forçando uma padronização. O usuário tem a liberdade de escrever os nomes das chaves com maiúsculas ou minúsculas, tanto faz, mas o programa sempre acessará a chave em maiúsculas. Os valores são sempre colocados entre aspas, sejam eles numéricos, liga/desliga ou texto.

Lembre-se de que em shell não há tipagem explícita de dados, então 5, '5' e "5" são iguais.

Estas são as especificações, já podemos começar a escrever o parser. Não é preciso ser vidente para adivinhar o que vem primeiro:

```
#!/bin/bash
# parser.sh
# Lê arquivos de configuração e converte os dados para
# variáveis do shell na saída padrão.
#
# 2006-10-31 Fulano da Silva
#
```

O caminho/nome do arquivo de configuração será recebido pela linha de comando. Verificações nunca são demais; então, antes de começar a procurar pelas chaves e valores, é bom conferir se o arquivo indicado realmente existe e se está legível. Nunca se esqueça deste detalhe da legibilidade. O fato do arquivo existir não significa que o parser conseguirá lê-lo. Podem haver problemas com as permissões do usuário, por exemplo, caso ele tente ler um arquivo de configuração de outro usuário ou do administrador do sistema.

```
# O arquivo de configuração é indicado na linha de comando
CONFIG=$1
# O arquivo deve existir e ser legível
```

```
if [ -z "$CONFIG" ]; then
    echo Uso: parser arquivo.conf
    exit 1
elif [ ! -r "$CONFIG" ]; then
    echo Erro: Não consigo ler o arquivo $CONFIG
    exit 1
fi
```

Se o arquivo de configuração passar pela verificação, está tudo pronto para a extração e conversão dos dados. Vai ser muito parecido com o que já foi feito no exemplo anterior, é um loop nas linhas do arquivo, ignorando comentários e linhas vazias.

```
# Loop para ler linha a linha a configuração, guardando em $LINHA
# Dica: Use $LINHA sem "aspas" para remover os brancos
while read LINHA; do
    # Ignorando as linhas de comentário
    [ "$(echo $LINHA | cut -c1)" = '#' ] && continue
    # Ignorando as linhas em branco
    [ "$LINHA" ] || continue
    # Guardando cada palavra da linha em $1, $2, $3, ...
    set - $LINHA

    # Extraindo os dados (chaves sempre maiúsculas)
    chave=$(echo $1 | tr a-z A-Z)
    shift
    valor=$*

    # Mostrando chave="valor" na saída padrão
    echo "CONF_$chave=\"$valor\""
done < "$CONFIG"
```

As diferenças para o código anterior estão em destaque. Agora, em vez de deixar a chave em minúsculas, foram convertidas para maiúsculas. Também ganharam um prefixo `CONF_` para se diferenciarem das outras variáveis já existentes nos programas. O bom e velho `echo` mostra o resultado na saída, no formato de *`variável`*`="`*`valor`*`"`. Não é mais necessário o `case`, pois nenhum processamento é feito nos dados. Este parser simplesmente lê, converte e mostra, fria e indiscriminadamente. Testando:

```
$ ./parser.sh mensagem.conf
CONF_USARCORES="ON"
CONF_CORFUNDO="44"
CONF_CORLETRA="37"
CONF_MENSAGEM="Modo texto é legal"
$
```

Se quiser que o parser retorne uma única configuração específica (digamos, somente a `CONF_MENSAGEM`), o `grep` é seu amigo.

Integrando os programas

Agora que o parser está genérico e funcional, podemos adaptar o `mensagem.sh` do exemplo anterior para utilizá-lo. As configurações são mandadas para a saída padrão já no formato de variáveis do shell, então um simples `eval` dará conta o recado.

mensagem.sh (usando parser externo)

```
1       #!/bin/bash
2       #
3       # mensagem.sh
4       # Mostra uma mensagem colorida na tela, lendo os
5       # dados de um arquivo de configuração externo.
6       #
7       # 2006-10-31 Fulano da Silva
8
9       CONFIG="mensagem.conf"    # Arquivo de configuração
10
11       # Configurações padrão
12       USAR_CORES=0                # config: UsarCores
13       COR_LETRA=                  # config: CorLetra
14       COR_FUNDO=                  # config: CorFundo
15       MENSAGEM='Mensagem padrão'  # config: Mensagem
16
17       # Carregando a configuração do arquivo externo
18       eval $(./parser.sh $CONFIG)
19
20       # Processando os valores
21       [ "$(echo $CONF_USARCORES | tr A-Z a-z)" = 'on' ] &&
 USAR_CORES=1
```

```
22      COR_FUNDO=$(echo $CONF_CORFUNDO | tr -d -c 0-9) # só
 números
23      COR_LETRA=$(echo $CONF_CORLETRA | tr -d -c 0-9) # só
 números
24      [ "$CONF_MENSAGEM" ] && MENSAGEM=$CONF_MENSAGEM
25
26      # Configurações lidas, mostre a mensagem
27
28      if [ $USAR_CORES -eq 1 ]; then
29          # Mostrar mensagem colorida
30          # Exemplo: \033[40;32mOlá\033[m
31          echo -e
 "\033[$COR_FUNDO;${COR_LETRA}m$MENSAGEM\033[m"
32      else
33          # Não usar cores
34          echo "$MENSAGEM"
35      fi
```

Na linha 18, o parser externo lê o arquivo de configuração e manda de volta os dados no formato de variáveis do shell, que, por sua vez, são incluídas no ambiente atual pelo comando `eval`. A seguir, as mesmas checagens de valor que antes estavam dentro do `case` e do `while` agora ocupam menos espaço. A parte final continua a mesma. Conferindo:

```
$ ./mensagem.sh
Modo texto é legal
$
```

É isso. Se houver melhorias a se fazer no parser, todos os programas que o utilizam serão beneficiados. Para que qualquer outro programa também utilize arquivos de configuração, basta adicionar uma única linha, a do `eval`. Uma implementação modular e simples, como deve ser.

Considerações de segurança

Mas voltando ao início deste tópico, lembre-se que esta é apenas uma das maneiras de se fazer um parser. É a mais simples, mas sofre de um problema sério: segurança. E se o usuário colocar a seguinte linha no arquivo de configuração:

```
Mensagem    $(cat /etc/passwd)
```

E aí? Pense por momento: este comando será executado? Todo o conteúdo do arquivo de usuários e senhas será mostrado na tela somente com a edição de uma linha do arquivo de configuração? Sim.

Fica a dica: evite o `eval`. Até soa bem falar isso: evite o `eval`. Sempre que ficar tentado a usar o `eval`, lembre-se de que o conteúdo da variável será executado como um comando normal do shell. A grande maioria dos seus usuários não sabe disso, mas basta chegar um moleque curioso para que o seu pesadelo comece. O exemplo anterior é bonzinho até, imagine se fossem as seguintes linhas no arquivo de configuração:

```
UsarCores $(cat /etc/passwd)
CorFundo  $HOME $PWD $UID
CorLetra  `rm *`
Mensagem  $(rm -rf /)
```

As variáveis seriam expandidas e os comandos, executados. Se alguém for executar o programa como usuário administrador (root), pode dizer adeus ao seu sistema, pois a última linha se encarregará de apagá-lo.

NÃO TESTE AS DUAS ÚLTIMAS LINHAS. O comando `rm` vai apagar seus arquivos, sem perguntar antes e sem undelete nem lixeira. Use `rm -f /tmp/*` se quiser verificar se ele realmente apaga algo.

Um remendo aqui seria remover os caracteres perigosos (`$` e a crase) antes de executar o `eval`, para evitar a execução de subshell. Mas o melhor mesmo é esquecer de vez do `eval` e partir para a implementação do parser mais robusto.

Parser mais robusto e seguro

Mencionado páginas atrás, o parser ideal tem um funcionamento mais tradicional e usa uma função para extrair o valor de uma chave:

```
$ CONFIG=mensagem.conf
$ source parser.sh
$ echo $(PegaValor Mensagem)
Modo texto é legal
$
```

Ou seja, o parser apenas definirá a função `PegaValor`, que será usada para obter um por um os valores das chaves, à medida que seu programa

precisar deles. A implementação do parser fica mais limpa e segura, não utilizando o `eval`. Em seu programa, bastar incluir a biblioteca e chamar as funções, nada de complicado.

A função pesquisará a chave no arquivo indicado pela variável de ambiente `$CONFIG`, mas também pode funcionar como um filtro, recebendo os dados de configuração pela entrada padrão (`STDIN`). Cabe ao programador decidir o que é mais conveniente.

Outro detalhe do parser é que ele também precisa gravar dados no arquivo de configuração, e não somente lê-los. Então, também é preciso uma função `GravaValor`. Esta função deve apagar o dado atual da chave e inserir um novo, lembrando que esta chave pode existir ou não no arquivo. Ou pode até estar em várias linhas, caso o usuário a tenha duplicado.

Por que o usuário duplicaria uma linha de configuração? Porque ele é usuário, ora :)

Enfim, o assunto arquivos de configuração não termina aqui, e a implementação do parser pode render várias horas de programação divertida (ou não). Outras melhorias que poderiam ser avaliadas:

- Se não existir o arquivo de configuração o que deve ser feito? Criar um vazio, criar um com valores padrão, mandar um aviso ou um erro para o usuário?
- A função que grava valores de volta no arquivo de configuração vai fazer isso editando somente a linha desejada ou vai reescrever o arquivo todo de uma vez? Ou vai apagar a linha atual e anexar a configuração nova no final? Os comentários originais serão preservados?
- O que fazer ao encontrar uma chave inválida? Ignorar silenciosamente ou avisar o usuário? É melhor tolerar o erro ou forçar as regras?
- Se o parser precisar mostrar mensagens (informativas ou de erro), como ele fará? Mandará para a saída padrão ou a de erro? Ou talvez gravará em um arquivo de log?
- Que tal fazer o parser já validar o valor? Por exemplo, você pode dizer para ele que quer o valor da chave `MAX_SIZE` e que este valor deve ser

um número. E se não for, o que acontece? O parser retornará um valor padrão (zero) ou mostrará um erro?

- Se o usuário tentou abusar de seu arquivo de configuração tentando "hackeá-lo", isso deve ser ignorado (simplesmente apagar os caracteres perigosos) ou alguma atitude deve ser tomada?

Deste ponto em diante, é com você. Avalie até aonde o seu parser deve evoluir e mãos à massa. Ou melhor, dedos ao teclado!

Para finalizar aguçando a sua curiosidade, uma função curta, porém poderosa. Ela retorna o valor de uma chave de um arquivo de configuração qualquer. Destrinchar a sopa de letrinhas fica como exercício para uma noite fria e chuvosa.

```
$ config() { tr -s '\t '  ' ' < "$2" | sed 's/^ //' | grep -i "^$1
  " | cut -d' ' -f2-; }
$ config mensagem mensagem.conf
Modo texto é legal
$
```

Capítulo 10

Banco de dados com arquivos texto

Guardar e acessar dados é uma necessidade básica para vários tipos de programas. Usar um banco de dados tradicional para um programa de pequeno e médio portes geralmente é um exagero. A solução é usar os próprios recursos do sistema para simular as funcionalidades básicas de um banco. Aprenda a utilizar arquivos de texto para o armazenamento de dados, eliminando a necessidade de um banco relacional.

O **banco de dados** é um lugar específico para se guardarem dados. O banco também manipula estes dados, apagando registros, incluindo novos e alterando registros existentes. O tipo mais comum é o banco de dados relacional. Geralmente, bancos de dados são sistemas complexos, que exigem instalação, configuração e manutenção. São feitos para gerenciar volumes imensos de informação em pouquíssimo tempo e têm facilidades como desfazimento, becape (backup) e otimizações.

Os **arquivos texto** são arquivos comuns do sistema, que guardam apenas texto, sem formatação ou imagens, como, por exemplo, arquivos de configuração e mensagens simples de e-mail. Um arquivo texto pode ser movido de um lugar para outro, transmitido pela Internet, guardado em CD e editado em qualquer editor de textos, mesmo nos mais simples como Bloco de Notas ou nano.

A ideia é juntar esses dois conceitos, implementando um banco de dados, utilizando apenas arquivos texto e ter um produto novo: um banco de dados simples, porém de fácil manipulação e extremamente portável. Isto é o "Banco de dados com arquivos texto", ou abreviadamente Banco Textual.

Vantagens:

- **Acesso fácil ao banco:** Sendo apenas um arquivo texto, pode-se usar qualquer editor para fazer manutenção no banco e alterar dados.
- **Portabilidade:** O banco pode ser facilmente transmitido via Internet, e-mail, guardado em pendrive e será o mesmo independente do sistema operacional. Pode-se inclusive copiar e colar com o mouse o seu conteúdo, pois é apenas texto.
- **Compactável:** Sendo somente texto, caso o banco fique com muitos dados, é possível compactá-lo para que seu tamanho fique reduzido. Embora tenha-se em mente que para bancos muito grandes os arquivos texto não são aconselháveis, pois o acesso fica lento.
- **Simplicidade:** O grande trunfo desse banco é ser simples. Sendo assim, é fácil compreender sua estrutura, editá-lo e escrever ferramentas para manipulá-lo.

Desvantagens:

- **Performance:** É rápido ler e manipular arquivos normais. Mas caso seu banco cresça muito (milhares de linhas), a velocidade de acesso vai ser prejudicada. Em outras palavras: quanto mais dados, mais lento.
- **Relacionamentos:** O banco em texto não tem relacionamentos entre os dados, então características de bancos de dados normais como chaves estrangeiras (*foreign keys*) e triggers não estão disponíveis. Até é possível de fazer, mas a complexidade de manter o código torna mais viável a adoção de um banco relacional em vez de textual.
- **Fragilidade:** Sendo apenas um arquivo texto, o banco está sujeito a acidentes de percurso que podem ocorrer com arquivos, como apagamento acidental e edição descuidada, apagando conteúdo sem perceber. E por ser possível o acesso direto ao banco, via editor de texto, são maiores as chances de se corromperem os dados em um erro de digitação.

Em resumo, a simplicidade é a maior vantagem do banco de dados textual, mas, por outro lado, é sua maior limitação. Antes de se decidir por usar o banco em texto, cabe uma análise prévia do caso e da situação específica, projeções de crescimento do banco e previsão da necessidade de operações avançadas.

De qualquer forma, mesmo que posteriormente se necessite migrar do banco texto para um relacional, o processo não é doloroso, pois texto é fácil de manipular. Já objetos e relacionamentos...

Quando utilizar bancos textuais

Antes de analisar os detalhes do conceito, é bom conhecer quais são os usos comuns para esse tipo de banco. Um programa sempre precisa guardar informações de estado, configuração ou funcionalidades. Distribuir um banco de dados relacional juntamente com o seu programa é um exagero, então o banco simplificado brilha isolado. Alguns exemplos:

- Guardar dados de estado para fins históricos, como quais foram os últimos arquivos abertos, as últimas palavras pesquisadas etc.

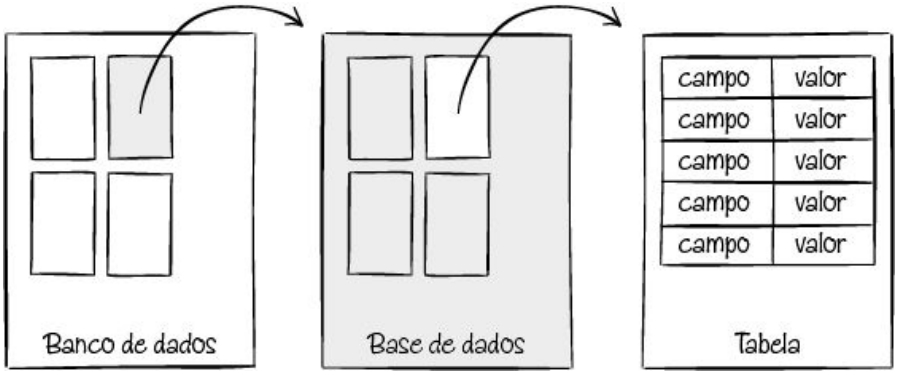
- Guardar dados temporários, utilizados apenas em tempo de execução, mas que precisam ser gravados em disco para garantir consistência.
- Guardar informações em um formato intermediário, quando estiver fazendo conversões.
- Guardar dados de configuração (Exemplo: `/etc/passwd`).

Ou, ainda, usar como base de dados para aplicativos completos como:

- Agenda de contatos pessoal.
- Catálogos de CDs ou arquivos MP3.
- Controle de estoque simples.
- Controle de CDs/fitas de becape.
- Controle de tarefas com estados (tipo TODO).
- Qualquer cadastro de dados simples ou históricos.

Definindo o formato do arquivo

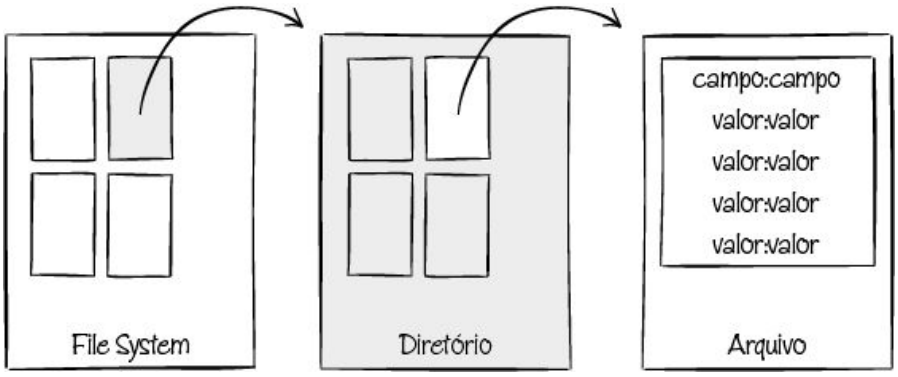
Um banco de dados relacional possui alguns componentes, e eles respeitam uma hierarquia interna. Um banco pode ter várias bases de dados. Cada uma dessas bases pode ter várias tabelas, que se relacionam entre si. Cada tabela pode ter vários campos e cada campo guarda um valor.



Componentes do banco relacional

O banco de dados textual usa a própria hierarquia já existente no sistema de arquivos. Há diretórios que contêm arquivos, que contêm

texto. Cada arquivo funciona como uma tabela de um banco de dados relacional.



Componentes do banco textual

Como diretórios e arquivos são entidades já existentes no sistema, basta arranjar os dados dentro do arquivo texto. Que padrão utilizar? Um registro por linha? Um campo por linha? Todos os registros em uma linha só? Como identificar os nomes dos campos?

A decisão de qual será o formato interno do arquivo texto e de como os dados ficarão arranjados, é crucial. A dificuldade em escrever os programas que acessarão o banco está diretamente relacionada ao projeto da estrutura do arquivo. Quanto mais complexa a estrutura, mais difícil será codificar o gerenciador desse banco. O ideal é que o formato seja o mais simples possível, com poucas regras, porém consistentes.

Um formato famoso de disposição de dados em arquivos texto é o CSV (Comma Separated Values), que significa Valores Separados por Vírgulas. É um formato simples, de poucas regras.

Especificação do formato CSV:

- Um registro por linha.
- A primeira linha contém os nomes dos campos.
- As linhas seguintes são os dados.
- Tanto os campos quanto os dados são separados entre si por vírgulas.
- Números são colocados diretamente.
- Textos são colocados “entre aspas”.

■ Textos podem conter vírgulas, pois estão protegidas pelas aspas.

Este formato é conhecido e largamente utilizado, sendo suportado por vários programas que manipulam dados, desde o Microsoft Excel até a Agenda de Endereços Online do Yahoo.

⌨ Arquivo CSV do Yahoo Endereços (resumido)

```
"Nome","Sobrenome","Apelido","E-mail","Fone","Celular"
"Aurelio","Marinho Jargas","aurelio","verde@aurelio.net","",""
"Maria","Cunha da Silva","maria","","","(41) 9999-1234"
"João","Almeida","joao","joao@email.com","3456-7890","9999-5678"
"","","zeca","zeca@coldmail.com","5432-9876",""
```

A primeira linha do exemplo indica os nomes dos campos, colocados entre aspas: Nome, Sobrenome, Apelido, E-mail, Fone e Celular. As linhas seguintes são os dados, um registro por linha, com os valores entre aspas e separados por vírgulas. Campos sem valor são definidos com aspas vazias "".

Apesar de simples, esse formato tem o complicador das aspas, que torna a extração dos dados mais complexa. Primeiro é preciso isolar os dados entre aspas (que podem conter vírgulas) para só depois dividir os registros em campos. Ficam as perguntas: como colocar aspas literais? E se não fechar as aspas? Para evitar estes detalhes, vamos fazer uma especificação mais simples ainda, um derivado do CSV.

Formato CSV simplificado

■ Um registro por linha.

■ A primeira linha contém os nomes dos campos.

■ As linhas seguintes são os dados.

■ Tanto os campos quanto os dados são separados entre si por vírgulas.

■ Vírgulas literais são "mascaradas" com um caractere exótico.

Os primeiros quatro itens são iguais aos do CSV, mas a grande diferença está no último. Para não precisar do esquema das aspas, simplesmente definimos a vírgula como único caractere especial, todos os outros são literais. Caso algum dado contenha uma vírgula literal, esta será

mascarada para um caractere exótico como "§" ou "£".

Formato CSV	Formato CSV simplificado
`"Nome","Endereço","Telefone"` `"Carlos","Rua das Palmeiras, 789","234-5678"`	`Nome,Endereço,Telefone` `Carlos,Rua das Palmeiras§ 789,234-5678`

As aspas não são mais necessárias e a vírgula foi trocada por um "§". Com isso simplificamos muito o formato do banco, porém perdemos a possibilidade de se colocar um "§" literal. Cada escolha sempre traz seus prós e contras, esse é o preço da simplificação. Por isso deve-se escolher um caractere realmente exótico, para que a sua falta não seja um problema em seu banco.

Mas ainda podemos melhorar um pouco esse formato. A vírgula é um caractere muito comum de se encontrar em dados, como no endereço já citado e em números com casas decimais e valores monetários. Usaremos então os dois-pontos ":" como o caractere separador de nosso banco textual. Apesar de comum em textos corridos, ele não aparece tanto em dados e seu uso já é conhecido como separador, como no arquivo de usuários do Unix, o `/etc/passwd`.

Formato CSV	Formato CSV simplificado
`"Nome","Endereço","Telefone"` `"Carlos","Rua das Palmeiras, 789","234-5678"`	`Nome:Endereço:Telefone` `Carlos:Rua das Palmeiras, 789:234-5678`

A chave primária

Ainda há um último melhoramento a ser feito. No formato CSV não existe o conceito de Chave Primária (Primary Key). Porém, isto é vital para um banco de dados, por mais simples que sejam: não podem haver informações repetidas. A chave primária serve para assegurar isso, não sendo possível cadastrar duas chaves de mesmo conteúdo no banco (`UNIQUE`).

Uma chave primária é um campo especial de uma tabela, que nunca aparece repetido. Ao tentar incluir um registro com uma chave que já exista, é mostrado um erro. Por exemplo, duas pessoas não podem ter o mesmo CPF, então esse número é único no banco. Não podem haver dois registros com o mesmo CPF. Então o campo que guarda o CPF será a chave única (primária) desta tabela.

A maneira mais limpa de introduzir o conceito de chave primária em

nossa especificação é considerar que o primeiro campo da linha será o identificador único. Ele pode ser um número serial ou um nome, tanto faz. A ordem dos campos dentro do arquivo não faz diferença para o aplicativo que utilizar o banco, então não há problema em considerarmos esta primeira posição algo privilegiado. Com isso fechamos os detalhes do formato.

Especificação final do formato CSV simplificado:

- Um registro por linha.
- A primeira linha contém os nomes dos campos.
- As linhas seguintes são os dados.
- O primeiro campo é a chave primária (única).
- Tanto os campos quanto os dados são separados entre si por dois-pontos ":".
- Dois-pontos literais são mascarados com o caractere exótico "§".

Voltando ao exemplo CSV do Yahoo, ele fica assim no formato simplificado:

```
Apelido:Nome:Sobrenome:E-mail:Fone:Celular
aurelio:Aurelio:Marinho Jargas:verde@aurelio.net::
maria:Maria:Cunha da Silva:::(41) 9999-1234
joao:João:Almeida:joao@email.com:3456-7890:9999-5678
zeca:::zeca@coldmail.com:5432-9876:
```

Note que como o campo Apelido foi colocado primeiro, ele passa a ser a chave primária destes dados. É isso. Temos uma estrutura, temos uma especificação. Agora só falta a parte boa: programar.

Gerenciador do banco

Um banco sozinho não tem muita serventia. De que adianta guardar dados em um formato especial se não se tem ferramentas para manipulá-los? Antes de sair programando uma agenda ou um catálogo de CDs, primeiro é preciso criar uma interface de comunicação com o banco textual, o componente intermediário que irá "falar" diretamente com o banco, o gerenciador de banco de dados.

Imagine que você hoje vai escrever a Agenda de Contatos usando um banco textual. Você senta, codifica e termina. Está tudo funcionando. Mas se passam dois meses e agora você precisa escrever o Catálogo de CDs. O aplicativo é totalmente diferente, mas o miolo do código que cuida da inserção, edição e remoção de dados no banco, é exatamente o mesmo!

Ter o mesmo código em dois ou mais programas não é um bom negócio. Cada programa segue sua evolução natural e rapidamente aquele trecho que era igual em ambos, ficará irreconhecível. As melhorias do miolo de um não serão aplicadas ao outro e logo logo cada um vai estar conversando com o banco de uma maneira diferente. É o mesmo problema visto no tópico de Arquivos de Configuração, que precisou de um gerenciador à parte.

Surge a necessidade de se ter um programa especialista, específico para se comunicar com o banco. Ele saberá como ler dados, incluir novos, apagar e alterar registros existentes. Será o mestre do banco, o único programa que terá acesso direto a ele. Os programas que usarão o banco para guardar dados são os programas aplicativos, que cuidarão da interface com o usuário e da manipulação dos dados extraídos do banco. Nessa categoria encaixam-se a Agenda e o Catálogo de CDs. Eles nunca acessarão o banco diretamente; em vez disso farão requisições ao gerenciador e este se encarregará de executá-las.

Comunicação com o gerenciador do banco de dados

Agora o shell entra na conversa

Nossa missão é escrever um gerenciador de banco de dados textual. Ao se pensar em qual linguagem implementar o gerenciador do banco textual, temos um candidato natural: falou em arquivos, falou em shell! Trabalhar com arquivos é básico no shell, é o tipo de tarefa para a qual ele foi feito. Sua função é interagir com o sistema de arquivos (filesystem), provendo facilidades para o usuário gerenciar seus arquivos e diretórios.

Além do shell, temos ao alcance uma vasta coleção de ferramentas especialistas em manipular texto, como `sed`, `cut` e `grep`. Elas fazem o trabalho pesado, transformando caracteres soltos em dados importantes. Com a dupla shell+ferramentas, tem-se um ambiente completo para desempenhar as tarefas de:

- Apagar, criar, alterar, copiar e mover arquivos.
- Apagar, incluir e alterar linhas de arquivos.

São exatamente estas as características necessárias para se gerenciar os arquivos de texto do nosso banco. Descrevendo de maneira mais genérica, estas são as funções que o gerenciador deve desempenhar:

- Mostrar registros.
- Apagar registros.
- Inserir registros.
- Modificar registros.

Codificação do gerenciador

Antes de pensar em codificar o gerenciador, precisamos criar um banco de testes. Para simplificar o processo, faremos um banco pequeno, com poucos campos e dados distintos entre si. Inspirado no arquivo de usuários do sistema Unix, temos:

⌨ Banco de testes – Usuários do sistema

```
login:nome completo:idade:sexo
junior:Alfredo Vieira Júnior:12:M
dani:Daniele Almeida Teles:19:F
luciana:Luciana Silveira:22:F
ana:Ana Paula de Souza:58:F
mario:Mário da Silva:35:M
ze:José Carlos Vaz:45:M
```

Na primeira linha estão identificamos os campos:

- Login (chave primária).
- Nome completo.

■ Idade.

■ Sexo.

Pelos nomes dos campos já sabemos qual o tipo do seu conteúdo. As linhas seguintes são os dados cadastrados nesse banco. São três homens e três mulheres, com idades variadas.

O gerenciador será uma biblioteca. Uma biblioteca não é um arquivo executável, mas, sim, um conjunto de funções especializadas usadas por outros programas. Seu programa inclui a biblioteca durante a execução e pode utilizar suas funções diretamente. Não entendeu nada? Não se preocupe, ver o exemplo funcionando vai clarear tudo.

Começaremos com uma biblioteca bem simples, que apenas cadastra e remove registros do banco, ainda sem se preocupar com checagens de erro ou dados inconsistentes. Posteriormente ela será melhorada.

bantex-1.sh

```
# bantex.sh - Gerenciador do Banco Textual
#
# Biblioteca de funções para gerenciar os dados do banco textual.
# Use o comando "source" para inclui-la em seu script.
#
# 2006-10-31 Fulano da Silva
#
#------------------[ configuração ]----------------
SEP=:           # defina aqui o separador, padrão é :
TEMP=temp.$$    # arquivo temporário
#------------------[ funções ]---------------------
# O arquivo texto com o banco já deve estar definido
[ "$BANCO" ] || {
    echo "Base de dados não informada. Use a variável BANCO."
    return 1
}
# Apaga o registro da chave $1 do banco
apaga_registro() {
    grep -i -v "^$1$SEP" "$BANCO" > "$TEMP"    # apaga a chave
    mv "$TEMP" "$BANCO"                        # regrava o banco
}
```

```
# Insere o registro $* no banco
insere_registro() {
    echo "$*" >> "$BANCO"        # grava o registro
}
```

Analise este código com atenção, acostume-se com ele. É o início de uma biblioteca que vai crescer e ficar mais complexa. Então, não perca o fio da meada.

Por enquanto, sem surpresas. Definição de variáveis globais no início, depois uma verificação básica se o banco foi definido na variável `$BANCO` e então seguem as duas funções para apagar e incluir registros no banco. Como cada registro é uma única linha, um `grep -v` o remove e um redirecionamento >> o adiciona no final do arquivo.

Para apagar, é passada a chave primária à função `apaga_registro`. Uma expressão regular é pesquisada, casando do início da linha até o primeiro separador. Supondo que a chave é joao, a expressão que o `grep` receberá será "`^joão:`". Caso a encontre, esta linha será omitida porque estamos usando a opção `-v`. E pronto, a linha foi apagada. Um arquivo temporário é usado para guardar o resultado e na linha seguinte o banco é regravado.

Para incluir um registro, a função `insere_registro` já recebe os dados prontos, com o separador entre os campos. Para guardar o registro novo no banco, basta adicionar a linha nova nele. Foi usado o redirecionamento `>>`.

Vamos testar? O aplicativo que usará esta biblioteca será a última parte que codificaremos, então faremos os testes na própria linha de comando. Mas como uma biblioteca não é um programa, não deve ser executada diretamente. O correto é usar o comando `source` para incluí-la na shell atual. Feito isso, poderemos usar as funções do banco.

```
$ source bantex-1.sh
Base de dados não informada. Use a variável BANCO.
$
```

O comando ponto "." faz o mesmo que o `source`. Então o comando anterior também poderia ser "`. bantex-1.sh`". Mas lembre-se que estamos aprendendo a programar do jeito profissional, pensando em manutenção futura. É muito melhor usar o `source`, pois além de ficar mais claro o que o comando faz (o que significa um ponto?), fica fácil dar um `grep` em vários arquivos para ver quais bibliotecas eles

utilizam.

Ops! É claro, antes de usar a biblioteca, precisamos especificar qual será o arquivo texto que ela manipulará. Nosso banco de testes foi gravado no arquivo `usuarios.txt`, que está no mesmo diretório da biblioteca. Conferindo:

```
$ ls
bantex-1.sh    usuarios.txt
$ cat usuarios.txt
login:nome completo:idade:sexo
junior:Alfredo Vieira Júnior:12:M
dani:Daniele Almeida Teles:19:F
luciana:Luciana Silveira:22:F
ana:Ana Paula de Souza:58:F
mario:Mário da Silva:35:M
ze:José Carlos Vaz:45:M
$
```

Beleza, vamos tentar de novo:

```
$ BANCO=usuarios.txt
$ source bantex-1.sh
$
```

Tudo certo, agora ele não reclamou. Lembre-se de que na filosofia Unix, se o comando não lhe disse nada, é porque está tudo bem. Então as funções já estão prontas para serem utilizadas. Será que funciona? Mário, sinto muito mas você vai ser ejetado:

```
$ apaga_registro mario
$ cat usuarios.txt
login:nome completo:idade:sexo
junior:Alfredo Vieira Júnior:12:M
dani:Daniele Almeida Teles:19:F
luciana:Luciana Silveira:22:F
ana:Ana Paula de Souza:58:F
ze:José Carlos Vaz:45:M
$
```

Funciona! Agora o teste da função de inclusão. Luigi, sua vez!

```
$ insere_registro luigi:Luigi da Silva:28:M
```

```
$ cat usuarios.txt
login:nome completo:idade:sexo
junior:Alfredo Vieira Júnior:12:M
dani:Daniele Almeida Teles:19:F
luciana:Luciana Silveira:22:F
ana:Ana Paula de Souza:58:F
ze:José Carlos Vaz:45:M
luigi:Luigi da Silva:28:M    <-----------------
$
```

Até aqui nossa biblioteca funciona como deveria. Está apagando os registros e incluindo os novos no final. Isso é o básico do básico do básico. E ainda assim está muito básico e fraquinho. E se eu tentar excluir uma chave que não existe? E se eu tentar incluir o Luigi de novo?

Excluir a chave que não existe não tem problema, pois o `grep` simplesmente não vai encontrá-la e nada será apagado. Mas o banco vai ser regravado sem necessidade. Já incluir a mesma chave duas ou mais vezes é uma falha grave, pois a chave primária deve ser única em todo o banco. Se já tem um luigi cadastrado, não pode haver outro. Então o primeiro melhoramento da biblioteca vai ser a habilidade de saber se uma determinada chave já existe ou não no banco.

```
# Verifica se a chave $1 está no banco
tem_chave() {
    grep -i -q "^$1$SEP" "$BANCO"
}
```

Usando a mesma expressão regular da função de exclusão, fazemos uma pesquisa silenciosa (`-q`) no banco, que é um teste básico que retorna SIM ou NÃO. Com este retorno é possível tomar decisões nas funções. Acompanhe as mudanças:

```
apaga_registro() {
    tem_chave "$1" || return                # se não tem, nem tente
    grep -i -v "^$1$SEP" "$BANCO" > "$TEMP"    # apaga a chave
    mv "$TEMP" "$BANCO"                        # regrava o banco
    echo "O registro '$1' foi apagado"
}
```

A função de remoção ganhou um teste já na primeira linha. Caso a

chave a ser removida não exista no banco, o processamento já para aqui. Isso evita regravar o arquivo do banco sem necessidade (`grep` + `mv`). Caso a chave exista e o registro seja apagado, uma mensagem informativa é mostrada na tela. Durante o desenvolvimento é aconselhável colocar estas mensagens para termos certeza do que acontece durante a execução do programa. A função de inserção sofreu mudanças mais profundas:

```
insere_registro() {
    local chave=$(echo "$1" | cut -d $SEP -f1)  # pega primeiro
  campo
    if tem_chave "$chave"; then
        echo "A chave '$chave' já está cadastrada no banco."
        return 1
    else                          # chave nova
        echo "$*" >> "$BANCO"       # grava o registro
        echo "Registro de '$chave' cadastrado com sucesso."
    fi
    return 0
}
```

É praticamente tudo novo. A linha que adiciona o registro no banco continua lá, mas agora há toda uma preparação antes de chegar nela. Primeiro a chave é extraída e armazenada em uma variável local da função. Então é feito o teste se esta chave já existe no banco. Se sim, uma mensagem informativa é mostrada e o processamento para, retornando código de erro (1). Se não existir a chave, o registro novo é incluído e uma mensagem é mostrada. O retorno é normal, código zero.

Foram várias mudanças, então o nome do arquivo da biblioteca mudará para `bantex-2.sh` e o cabeçalho registrará as mudanças. Note que isso não é necessário já que ainda estamos bem no início do desenvolvimento, mas é bom para praticar o procedimento de atualização.

```
# 2006-10-31 v1 Fulano da Silva: Versão inicial
# 2006-10-31 v2 Fulano da Silva:
#   - Adicionada função tem_chave()
#   - Inserção e exclusão agora checam antes a existência da chave
#   - Adicionadas mensagens informativas na inserção e exclusão
#
```

Após qualquer alteração na biblioteca, é necessário incluí-la novamente na shell para utilizar a versão nova do arquivo. Quer apostar quanto que você vai esquecer disso várias vezes?

```
$ source bantex-2.sh
$ tem_chave luigi && echo sim || echo não
sim
$ tem_chave LUIGI && echo sim || echo não
sim
$ tem_chave jonas && echo sim || echo não
não
$
```

Não é preciso redefinir a variável `$BANCO`, pois continuamos na mesma shell e seu valor não foi alterado. A função nova `tem_chave` já foi testada e está funcionando. Perceba que como foi utilizada a opção `-i` no `grep`, tanto faz informar a chave em maiúsculas ou minúsculas. Esta é uma decisão estrutural que você pode alterar caso queira ter ambas as chaves `luigi` e `LUIGI` em seu banco, cada uma sendo um registro diferente. Para preservar sua sanidade e suas madeixas, sugiro não fazer isso.

```
$ apaga_registro duende
$ insere_registro luigi:Luigi da Silva:28:M
A chave 'luigi' já está cadastrada no banco.
$
```

Continuando os testes, tentamos apagar um registro que não existe e tudo correu bem. O comando foi silenciosamente ignorado. Ao tentar incluir um registro novo com uma chave já existente, a mensagem informa que isso não será possível. Estamos progredindo!

Mas colocar checagens é chato, o legal é criar novidades. Sabe o que mais eu gostaria de ter nessa biblioteca? Uma função que me mostrasse quais são os nomes dos campos do banco e outra que mostrasse todos os dados de um registro qualquer. Os nomes dos campos é fácil, estão na primeira linha do banco. Já os dados requer um loop para mostrar um por linha e ficar bacana.

```
# Mostra os nomes dos campos do banco, um por linha
campos() {
    head -n 1 "$BANCO" | tr $SEP \\n
```

```
}
# Mostra os dados do registro da chave $1
mostra_registro() {
    local dados=$(grep -i "^$1$SEP" "$BANCO")
    local i=0
    [ "$dados" ] || return                    # não achei
    campos | while read campo; do             # para cada campo...
        i=$((i+1))                            # índice do campo
        echo -n "$campo: "                    # nome do campo
        echo "$dados" | cut -d $SEP -f $i     # conteúdo do campo
    done
}
```

Antes de dissecar as funções novas, veja como elas se comportam:

```
$ campos
login
nome completo
idade
sexo
$ mostra_registro luigi
login: luigi
nome completo: Luigi da Silva
idade: 28
sexo: M
$
```

A primeira não tem segredo, é a primeira linha do banco, com quebras em cada separador, ficando um campo por linha. Útil para montar formulários na sua aplicação ou inspecionar quais campos determinado banco possui.

A segunda já é mais elaborada. Primeiro os dados do registro são obtidos com a mesma expressão regular já usada nas funções anteriores. Se não fosse o `$1` que varia, essa expressão poderia estar em uma variável, não é mesmo? A clássica variável `$i` será o contador para o loop. Falando nele, é um `while` que lê um campo a cada passo, aproveitando a função `campos()` recém-criada. O nome do campo é mostrado. Note o uso da opção `-n` do `echo` para que a quebra de linha não seja mostrada no final. Assim o próximo `echo` vai ser mostrado nesta mesma linha.

Os dados de cada campo são extraídos com um `cut` que pega somente o campo correto da variável `$dados`, usando o contador `$i` para identificar o índice. Para ajudar a clarear este trecho, no primeiro passo o comando será: `echo "$dados" | cut -d : -f 1`. E assim o loop segue, mostrando um *campo*:*valor* por linha. Note que o número de campos não é fixo no código, então essa função funcionará tanto para bancos com dois campos quanto para bancos de 10 ou mais campos. Vejamos como está a biblioteca nesta terceira versão:

bantex-3.sh

```
# bantex.sh - Gerenciador do Banco Textual
#
# Biblioteca de funções para gerenciar os dados do banco textual.
# Use o comando "source" para inclui-la em seu script.
#
# 2006-10-31 v1 Fulano da Silva: Versão inicial
# 2006-10-31 v2 Fulano da Silva:
#   - Adicionada função tem_chave()
#   - Inserção e exclusão agora checam antes a existência da chave
#   - Adicionadas mensagens informativas na inserção e exclusão
# 2006-10-31 v3 Fulano da Silva:
#   - Adicionadas funções campos() e mostra_registro()
#
#---------------[ configuração ]--------------
SEP=:           # defina aqui o separador, padrão é :
TEMP=temp.$$    # arquivo temporário
#------------[ funções ]--------------
# O arquivo texto com o banco já deve estar definido
[ "$BANCO" ] || {
    echo "Base de dados não informada. Use a variável BANCO."
    return 1
}
# Verifica se a chave $1 está no banco
tem_chave() {
    grep -i -q "^$1$SEP" "$BANCO"
}
# Apaga o registro da chave $1 do banco
apaga_registro() {
```

```
    tem_chave "$1" || return                  # se não tem, nem
  tente
    grep -i -v "^$1$SEP" "$BANCO" > "$TEMP"   # apaga a chave
    mv "$TEMP" "$BANCO"                       # regrava o banco
    echo "O registro '$1' foi apagado"
}
# Insere o registro $* no banco
insere_registro() {
    local chave=$(echo "$1" | cut -d $SEP -f1)  # pega primeiro
  campo
    if tem_chave "$chave"; then
        echo "A chave '$chave' já está cadastrada no banco."
        return 1
    else                          # chave nova
        echo "$*" >> "$BANCO"     # grava o registro
        echo "Registro de '$chave' cadastrado com sucesso."
    fi
    return 0
}
# Mostra os nomes dos campos do banco, um por linha
campos() {
    head -n 1 "$BANCO" | tr $SEP \\n
}
# Mostra os dados do registro da chave $1
mostra_registro() {
    local dados=$(grep -i "^$1$SEP" "$BANCO")
    local i=0
    [ "$dados" ] || return            # não achei
    campos | while read campo; do     # para cada campo...
        i=$((i+1))                    # índice do campo
        echo -n "$campo: "                 # nome do campo
        echo "$dados" | cut -d $SEP -f $i  # conteúdo do campo
    done
}
```

Estamos quase lá. Mais algumas perguntinhas: e se o arquivo do banco não existir? Ou se ele existir mas não tiver permissão de leitura e/ou escrita? E se eu quiser somente o campo X do registro Y? E o tal do caractere exótico, como fica? De volta ao editor de textos...

```
# O arquivo deve poder ser lido e gravado
[ -r "$BANCO" -a -w "$BANCO" ] || {
   echo "Base travada, confira as permissões de leitura e
  escrita."
   return 1
}
```

Tá, primeira e segunda perguntas respondidas rapidinho, sem problema. Para obter somente um campo específico do registro, precisamos de uma função nova. Essa função receberá a chave do registro a ser obtido e o número do campo a ser extraído. Poderia ser o nome do campo em vez do número, é uma decisão estrutural. Usaremos o número para ficar mais simples a implementação.

```
# Mostra o valor do campo número $1 do registro de chave $2
  (opcional)
pega_campo() {
    local chave=${2:-.*}
    grep -i "^$chave$SEP" "$BANCO" | cut -d $SEP -f $1
}
```

Novamente a mesma expressão regular para obter o registro da chave passada pelo `$2` e o `cut` encarrega-se de extrair somente o campo desejado, informado pelo `$1`. Uma esperteza torna esta função ainda mais poderosa. Caso não seja informado o nome da chave, todas as chaves serão pesquisadas. Veja como funciona: a primeira linha da função guarda o nome da chave, obtendo o valor de `$2`. Mas caso ele não seja informado, é guardada a expressão .* em seu lugar, que é o curinga que casa qualquer coisa. A sintaxe `${`*`variável`*`:-`*`texto`*`}` é expansão de variáveis do shell, você sabe. Teste:

```
$ pega_campo 2
nome completo
Alfredo Vieira Júnior
Daniele Almeida Teles
Luciana Silveira
Ana Paula de Souza
José Carlos Vaz
Luigi da Silva
$ pega_campo 2 luigi
Luigi da Silva
```

```
$ pega_campo 3 luigi
28
$
```

Legal, hein? E ainda tem um bônus. Se você pegar o primeiro campo, ganha de brinde a lista de todas as chaves primárias de seu banco. Essa listagem é útil para compor menus ou fazer pesquisas em sua base. Mas e o caractere exótico, quando ele vai entrar na brincadeira?

Lembre-se de que todos os dois-pontos literais ":" devem ser mascarados para o caractere "§". Mas aqui encontramos um problema estrutural de nossa biblioteca. Na inserção de um registro novo, a `insere_registro()` já recebe a linha pronta, com os separadores já colocados. Com isso, o mascaramento deve ser feito no próprio aplicativo, quando o ideal seria que ele fosse feito internamente pela biblioteca. Fica assim por enquanto. O que a biblioteca pode fazer para ajudar é prover uma função que faça a conversão.

```
MASCARA=§     # caractere exótico para mascarar o separador
...
# Esconde/revela o caractere separador quando ele for literal
mascara()    { tr $SEP $MASCARA ; }   # exemplo: tr : §
desmascara() { tr $MASCARA $SEP ; }   # exemplo: tr § :
...
pega_campo() {
    local chave=${2:-.*}
    grep -i "^$chave$SEP" "$BANCO" | cut -d $SEP -f $1 |
  desmascara
}
...
mostra_registro() {
    ...
      echo "$dados" | cut -d $SEP -f $i | desmascara  # conteúdo
  do campo
    ...
}
```

Várias mudanças no código. Primeiro a variável global `$MASCARA` que guarda o caractere exótico. As duas funções novas `mascara()` e `desmascara()` fazem os dois caminhos de proteger o caractere separador

para inserir dados no banco e restaurá-lo na hora de pegar estes dados. As funções que mostram dados foram modificadas para fazer uso dessa funcionalidade. Veja:

```
$ insere_registro esquisita:Um§Dois§Três:77:F
Registro de 'esquisita' cadastrado com sucesso.
$ grep esquisita usuarios.txt
esquisita:Um§Dois§Três:77:F
$ pega_campo 2 esquisita
Um:Dois:Três
$ mostra_registro esquisita
login: esquisita
nome completo: Um:Dois:Três
idade: 77
sexo: F
$
```

Na inserção, é preciso usar o caractere exótico "na mão". Bug da biblioteca. Para mudar isso é preciso mudar a maneira que a função `insere_registro` recebe os dados. Poderia receber um campo por linha, por exemplo. Mas isso fica de exercício de casa. No comando seguinte vemos com o `grep` que os dados foram incluídos mascarados no banco, conforme desejado. E para terminar, as funções que extraem dados estão desmascarando o exótico, então sua aplicação não precisa se preocupar com isso.

Ufa!

Com isso terminamos a versão inicial de nosso gerenciador de banco textual. Ele possui as funcionalidades básicas que um aplicativo irá precisar para manipular um banco. Ao aplicativo basta definir a variável `$BANCO`, incluir o gerenciador com o comando `source` e usar as funções conforme necessário. Use a imaginação e crie um programa simples que possa manipular algum tipo de dados. Cadastros, listagem, estoques e organizadores são bons exemplos.

Várias melhorias podem ser feitas, como checagens adicionais, funções novas para facilitar o uso, funções de estatísticas (número de campos e registros, espaço ocupado), função de ordenamento (`sort`), função de edição de um registro, função de log (quem fez o que quando?), função...

E por aí vai, crie ferramentas conforme vá precisando delas. Cuide para não exagerar. Não sobrecarregue a biblioteca, deixe-a limpa com o que é realmente útil para seus programas.

Bantex, o gerenciador do banco textual

```
1       # bantex.sh - Gerenciador do Banco Textual
2       #
3       # Biblioteca de funções para gerenciar os dados do banco
  textual.
4       # Use o comando "source" para inclui-la em seu script.
5       #
6       # 2006-10-31 v1 Fulano da Silva: Versão inicial
7       # 2006-10-31 v2 Fulano da Silva:
8       #   - Adicionada função tem_chave()
9       #   - Inserção e exclusão agora checam antes a existência
  da chave
10      #   - Adicionadas mensagens informativas na inserção e
  exclusão
11      # 2006-10-31 v3 Fulano da Silva:
12      #   - Adicionadas funções campos() e mostra_registro()
13      # 2006-10-31 v4 Fulano da Silva:
14      #   - Adicionada verificação de permissões do arquivo do
  banco
15      #   - Adicionada função pega_campo()
16      #   - Adicionadas funções mascara() e desmascara() e
  $MASCARA
17      #
18
19      #-------------[ configuração ]--------------
20
21      SEP=:           # defina aqui o separador, padrão é :
22      TEMP=temp.$$    # arquivo temporário
23      MASCARA=§       # caractere exótico para mascarar o
  separador
24
25
26      #-------------[ funções ]-----------------
27
28      # O arquivo texto com o banco já deve estar definido
```

```
29      [ "$BANCO" ] || {
30          echo "Base de dados não informada. Use a variável
 BANCO."
31          return 1
32      }
33
34      # O arquivo deve poder ser lido e gravado
35      [ -r "$BANCO" -a -w "$BANCO" ] || {
36          echo "Base travada, confira as permissões de leitura
 e escrita."
37          return 1
38      }
39
40      # Esconde/revela o caractere separador quando ele for
 literal
41      mascara()    { tr $SEP $MASCARA ; }   # exemplo: tr : §
42      desmascara() { tr $MASCARA $SEP ; }   # exemplo: tr § :
43
44      # Verifica se a chave $1 está no banco
45      tem_chave() {
46          grep -i -q "^$1$SEP" "$BANCO"
47      }
48
49      # Apaga o registro da chave $1 do banco
50      apaga_registro() {
51          tem_chave "$1" || return                # se não
 tem, nem tente
52          grep -i -v "^$1$SEP" "$BANCO" > "$TEMP"  # apaga a
 chave
53          mv "$TEMP" "$BANCO"                     # regrava o
 banco
54          echo "O registro '$1' foi apagado"
55      }
56
57      # Insere o registro $* no banco
58      insere_registro() {
59          local chave=$(echo "$1" | cut -d $SEP -f1)  # pega
 primeiro campo
60
61          if tem_chave "$chave"; then
```

```
62                echo "A chave '$chave' já está cadastrada no
 banco."
63                return 1
64            else                                # chave nova
65                echo "$*" >> "$BANCO"         # grava o registro
66                echo "Registro de '$chave' cadastrado com
 sucesso."
67            fi
68            return 0
69        }
70
71        # Mostra o valor do campo número $1 do registro de chave
 $2 (opcional)
72        pega_campo() {
73            local chave=${2:-.*}
74            grep -i "^$chave$SEP" "$BANCO" | cut -d $SEP -f $1 |
 desmascara
75        }
76
77        # Mostra os nomes dos campos do banco, um por linha
78        campos() {
79            head -n 1 "$BANCO" | tr $SEP \\n
80        }
81
82        # Mostra os dados do registro da chave $1
83        mostra_registro() {
84            local dados=$(grep -i "^$1$SEP" "$BANCO")
85            local i=0
86            [ "$dados" ] || return           # não achei
87            campos | while read campo; do    # para cada campo...
88                i=$((i+1))                   # índice do campo
89                echo -n "$campo: "           # nome do campo
90                echo "$dados" | cut -d $SEP -f $i |
 desmascara    # conteúdo do campo
91            done
92        }
```

Zuser, o aplicativo que usa o banco

Achou que ia ficar na mão sem um exemplo de aplicativo, não é?

Dijeininhum! Vamos fazer um programinha bem simples para mostrar essa integração com o gerenciador e aguçar suas ideias para implementar seus próprios programas.

O programa zuser gerencia a base de usuários do sistema fictício Z. Coincidentemente, o sistema Z usa os mesmos dados que já temos cadastrados em nosso `usuarios.txt` do exemplo anterior. Veja só que beleza! Este programa é interativo e tem três comandos: lista, adiciona e remove. Como você já está embalado pela codificação passo a passo do gerenciador, podemos pisar no acelerador e já analisar um código maduro:

zuser

```
1       #!/bin/bash
2       #
3       # zuser
4       # Lista, adiciona e remove usuários do sistema Z
5       #
6       # Requisitos: bantex.sh
7       #
8       # Uso: zuser [ lista | adiciona | remove ]
9       #
10       # 2006-10-31 Fulano da Silva
11
12
13       # Se não passar nenhum argumento, mostra a mensagem de
 ajuda
14       [ "$1" ] || {
15           echo
16           echo "Uso: zuser [ lista | adiciona | remove ]"
17           echo
18           echo "   lista    - Lista todos os usuários do
 sistema"
19           echo "   adiciona - Adiciona um usuário novo no
 sistema"
20           echo "   remove   - Remove um usuário do sistema"
21           echo
22           exit 0
23       }
```

```
24
25      # Localização do arquivo do banco de dados
26      BANCO=usuarios.txt
27
28      # Inclui o gerenciador do banco
29      source bantex.sh || {
30          echo "Ops, ocorreu algum erro no gerenciador do
 banco"
31          exit 1
32      }
33
34      # Lida com os comandos recebidos
35      case "$1" in
36
37          lista)
38              # Lista dos logins (apaga a primeira linha)
39              pega_campo 1 | sed 1d
40          ;;
41
42          adiciona)
43              echo -n "Digite o login do usuário novo: "
44              read login
45
46              # Temos algo?
47              [ "$login" ] || {
48                  echo "O login não pode ser vazio, tente
 novamente."
49                  exit 1
50              }
51
52              # Primeiro confere se já não existe esse usuário
53              tem_chave "$login" && {
54                  echo "O usuário '$login' já foi cadastrado."
55                  exit 1
56              }
57
58              # Ok, é um usuário novo, prossigamos
59              echo -n "Digite o nome completo: "
60              read nome
```

```
61                echo -n "Digite a idade: "
62                read idade
63                echo -n "É do sexo masculino ou feminino? [MF] "
64                read sexo
65                echo
66
67                # Dados obtidos, hora de mascarar eventuais dois-
 pontos
68                nome=$(echo $nome | mascara)
69
70                # Tudo pronto, basta inserir
71                insere_registro "$login:$nome:$idade:$sexo"
72                echo
73            ;;
74
75            remove)
76                # Primeiro mostra a lista de usuários, depois
 pergunta
77                echo "Lista dos usuários do sistema Z:"
78                pega_campo 1 | sed 1d | tr \\n ' '
79                echo
80                echo
81                echo -n "Qual usuário você quer remover? "
82                read login
83                echo
84
85                # Vamos apagar ou puxar a orelha?
86                if tem_chave "$login"; then
87                    apaga_registro "$login"
88                else
89                    echo "Não, não, esse usuário não está
 aqui..."
90                fi
91                echo
92            ;;
93
94            *)
95                # Qualquer outra opção é erro
96                echo "Opção inválida $1"
97                exit 1
```

```
98            ;;
99        esac
```

Viu como o código fica simples e legível quando se tem um gerenciador que toma conta dos detalhes chatos? Pelos comentários e mensagens é possível entender tudo o que o programa faz, sem nenhuma surpresa. Nestas poucas linhas temos um aplicativo completo, com:

- Código limpo e comentado.
- Tela de ajuda.
- Checagens de erro em pontos importantes.
- Interatividade na inclusão e exclusão dos dados.
- Opções de linha de comando.
- Detecção de opções inválidas.
- Informação em caso de problemas com chaves existentes ou não.

São os pequenos detalhes que fazem a diferença entre um programa e um script, percebe? Já está enxergando a luz ou ainda precisa de mais exemplos? Bem, precisando ou não, eles virão... Voltando ao tópico, veja como fica a execução de nosso programa:

```
$ ./zuser
Uso: zuser lista|adiciona|remove
   lista    - Lista todos os usuários do sistema
   adiciona - Adiciona um usuário novo no sistema
   remove   - Remove um usuário do sistema
$ ./zuser lista
junior
dani
luciana
ana
ze
luigi
esquisita
$ ./zuser adiciona
Digite o login do usuário novo: guybrush
Digite o nome completo: Guybrush Threepwood
Digite a idade: 18
```

```
É do sexo masculino ou feminino? [MF] M
Registro de 'guybrush' cadastrado com sucesso.
$ ./zuser remove
Lista dos usuários do sistema Z:
junior dani luciana ana ze luigi esquisita guybrush
Qual usuário você quer remover? esquisita
O registro 'esquisita' foi apagado
$ ./zuser lista
junior
dani
luciana
ana
ze
luigi
guybrush
$
```

Só para finalizar, um detalhezinho. Sabe o Sr. Miyagi que manda o Daniel San ficar pintando cercas e encerando o chão para somente depois ensinar como dar socos e chutes? Ou aquele pai que tem um carro zero na garagem mas ensina o filho a dirigir em um fusca ou jipe? Pois é, pode parecer injusto, mas aprender o mais difícil primeiro, compensa a longo prazo.

Lembra-se do tal caractere exótico, das funções `mascara` e `desmascara`, da `$MASCARA` e da `$SEP` em todos os `cut`? Pois é, você aprendeu do jeito difícil. Se usar um TAB como separador dos campos do banco, nada disso é necessário ;)

Capítulo 11

Interfaces amigáveis com o Dialog

Seu programa não precisa ficar restrito à interface de linha de comando. Usuários em geral estão acostumados às interfaces mais coloridas e interativas, com suas janelas, botões e menus. Aprenda a dar esta roupagem mais amigável ao seu programa, fazendo-o funcionar como um aplicativo gráfico. No processo você vai descobrir que, além de facilitar a vida do usuário, você ainda simplifica seus códigos, eliminando verificações que tornam-se desnecessárias devido à natureza restrita da interface.

Este capítulo vai ser diferente. Começaremos já codificando e somente no final conheceremos os detalhes do conceito. Aproveitando que o código do aplicativo `Zuser` que acabamos de fazer no capítulo anterior ainda está fresco na memória, vamos transformá-lo em um aplicativo bem amigável com o usuário, com janelas, menus e botões.

Apresentação rápida do Dialog

O dialog é um programa de console (modo texto) que desenha janelas na tela, com menus, caixas de texto e botões. Feito especialmente para ser usado com o shell, é excelente para criar interfaces amigáveis para o usuário, fazendo com que ele escolha itens de menu em vez de digitar opções na linha de comando.

Por exemplo, o `Zuser` aceitava três opções na linha de comando: lista, adiciona e remove. A sua versão "gráfica" usando dialog apresentará ao usuário um menu com estas mesmas três opções. Assim fica bem claro quais são as funções do programa e evita-se o problema do usuário digitar uma opção errada.

Há vários tipos de janela que o dialog pode fazer, entre elas: menu, entrada de texto, entrada de senhas, calendário, escolha de arquivo e a tradicional janelinha com os botões SIM e NÃO. Iremos conhecendo cada uma à medida que formos precisando delas.

Para terminar esta apresentação, conheça a cara do dialog:

```
$ dialog --msgbox 'Olá, sou o dialog' 5 40
```

Zuserd, o Zuser com interface amigável

Vamos aprender a usar o dialog na prática, modificando o aplicativo `Zuser` do capítulo anterior para que ele use janelas em vez de opções de linha de comando. Se você não se lembra direito do `Zuser`, volte algumas páginas e

refresque sua memória.

Chamaremos esse aplicativo novo de `Zuserd`, com a última letra indicando que esta é a versão que utiliza o dialog como interface. Começaremos com algumas mudanças no cabeçalho e código inicial, acompanhe:

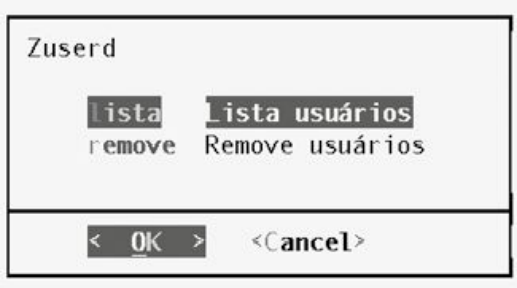
zuserd (início)

```
1       #!/bin/bash
2       #
3       # zuserd
4       # Lista, adiciona e remove usuários do sistema Z
5       #
6       # Requisitos: bantex.sh, dialog
7       #
8       # 2006-10-31 Fulano da Silva
9
10       # Localização do arquivo do banco de dados
11       BANCO=usuarios.txt
12
13       # Inclui o gerenciador do banco
14       source bantex.sh || {
15           echo "Ops, ocorreu algum erro no gerenciador do
 banco"
16           exit 1
17       }
18
```

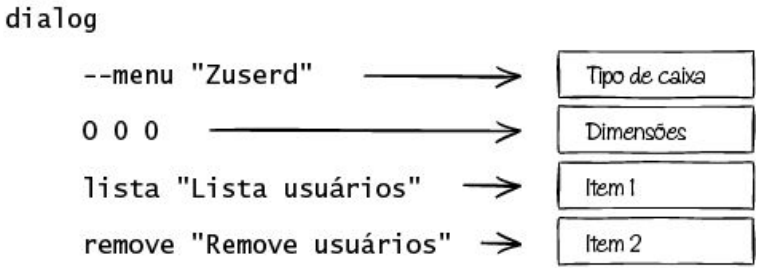
No cabeçalho foi mudado o nome do programa, adicionado o dialog como requisito e foi retirada a linha 8, que demonstrava o uso do programa. Como não teremos mais opções de linha de comando, não é necessário indicar seu uso. Por falar nisso, todas aquelas linhas que mostravam o texto de ajuda, caso nenhum argumento fosse passado, também foram varridas. Começamos simplificando as coisas. Que bom se sempre fosse assim, não?

O próximo trecho de código já é o `case` que lida com a opção escolhida pelo usuário. Então antes disso precisamos mostrar o menu na tela, para que ele possa escolher algo. Vamos começar a brincar de dialog:

```
$ dialog --menu "Zuserd" 0 0 0 lista "Lista usuários" remove
  "Remove usuários"
```



Vamos analisar este comando. A primeira opção `--menu` indica qual o tipo de janela a ser desenhado, passando o texto `Zused` como descrição. A seguir vêm as três dimensões da janela: altura, largura e número de itens visíveis no menu. Usando o zero, essas dimensões são calculadas automaticamente pelo programa. Depois vêm os itens do menu, em duplas. Primeiro uma palavra-chave para identificar o item, seguida de sua descrição. Foram usados apenas dois pares neste exemplo, com as palavras-chave `lista` e `remove`. Os botões são colocados automaticamente.



Fácil, não? Se precisar de mais itens, basta colocá-los no final da linha de comando. Vamos ver como fica o código do menu dentro de nosso programa:

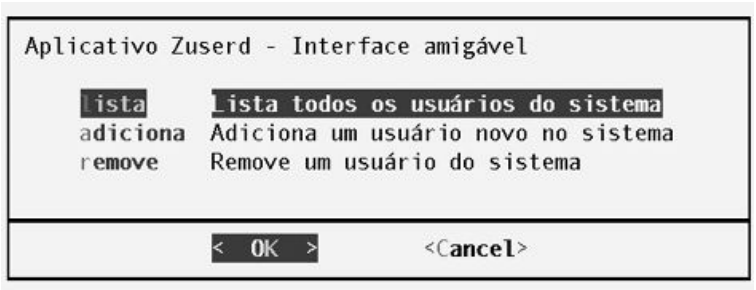
```
19      acao=$( dialog --stdout \
20          --menu "Aplicativo Zuser - Interface amigável" \
21          0 0 0 \
22          lista "Lista todos os usuários do sistema" \
23          adiciona "Adiciona um usuário novo no sistema" \
24          remove "Remove um usuário do sistema")
25
```

Há vários detalhes para se atentar neste trecho:

■ O comando `dialog` foi quebrado em várias linhas para ficar mais fácil

enxergar cada um de seus componentes. Note o escape "\" no final de cada linha. **Atenção:** Não podem haver espaços em branco após a barra.

- Foram colocadas no menu as três opções que o Zuser antigo aceitava pela linha de comando.
- A primeira opção `--stdout` faz o item de menu escolhido pelo usuário ser mandado para a saída padrão, que vai para dentro da variável `$acao`. Então ao final deste comando a variável vai conter "lista", "adiciona" ou "remove".



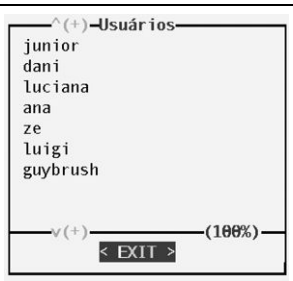
Neste ponto do código, já saberemos que tipo de ação o usuário vai querer executar. É hora de vir o case para decidir a rota de execução. Vamos aproveitar e já modificar o código que trata do primeiro item do menu, a listagem dos usuários:

```
26      # Lida com os comandos recebidos
27      case "$acao" in
28
29          lista)
30              # Lista dos logins (apaga a primeira linha)
31              temp=$(mktemp -t lixo)
32              pega_campo 1 | sed 1d > "$temp"
33              dialog --title "Usuários" --textbox "$temp" 13 30
34              rm $temp
35          ;;
36
```

O comando para obter a lista de todos os usuários do sistema continua o mesmo (`pega_campo | sed`), só que agora sua saída é guardada em um arquivo temporário. Ele é necessário para o uso com a janela `--textbox`, que serve para mostrar o conteúdo de um arquivo, usando rolagem caso o

texto não caiba inteiro na tela. Se quiser testar essa janela na linha de comando, faça: `dialog --textbox /etc/passwd 0 0`.

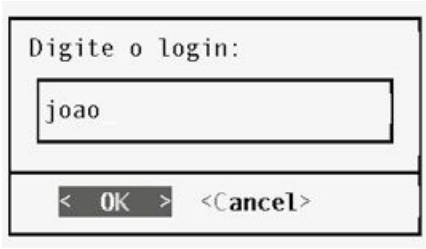
Voltando ao aplicativo, perceba que foi usada a opção `--title`. Seu uso é facultativo e ela pode ser empregada em todas as janelas. Sua função é escrever um texto no topo, no meio da borda superior. Outra novidade é a definição manual das dimensões em 13 linhas e 30 colunas, pois usando o dimensionamento automático a janela neste caso fica feia, muito estreita. Veja a diferença:

<table>
<tr>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td>Dimensão automática: 0 x 0</td>
<td>Dimensão manual: 13 x 30</td>
</tr>
</table>

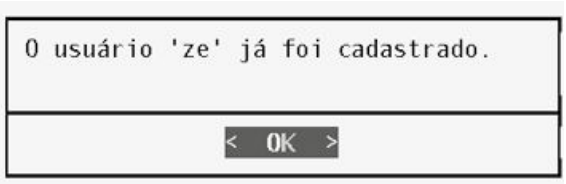
Beleza, a primeira das três funções do programa já está pronta. Vamos passar ao trecho que adiciona um usuário novo no banco. Veja as alterações feitas no segundo item do comando `case`:

```
37          adiciona)
38              login=$(dialog --stdout --inputbox "Digite o
 login:" 0 0)
39              [ "$login" ] || exit 1
40
41              # Primeiro confere se já não existe esse usuário
42              tem_chave "$login" && {
43                  msg="O usuário '$login' já foi cadastrado."
44                  dialog --msgbox "$msg" 6 40
45                  exit 1
46              }
47
```

Há dois tipos novos de janelas para conhecermos. O primeiro é o `--inputbox`, que, como o nome sugere, serve para obter um texto digitado pelo usuário. Na versão anterior do aplicativo (Zuser) era usado um `read` neste trecho. O funcionamento é o mesmo, o que o usuário digitar vai ser guardado na variável `$login`.



Em seguida é feito um teste básico para saber se o usuário digitou algo realmente ou deixou a caixa vazia. Neste caso, o programa acaba aqui. O próximo teste verifica se o usuário digitado já existe no banco. Lembre-se de que não é permitido ter dois usuários com o mesmo login. A linha que antes era um `echo` foi trocada por uma janela do tipo `--msgbox`. Ela é bem simples, serve para mostrar um aviso na tela e possui somente o botão OK.



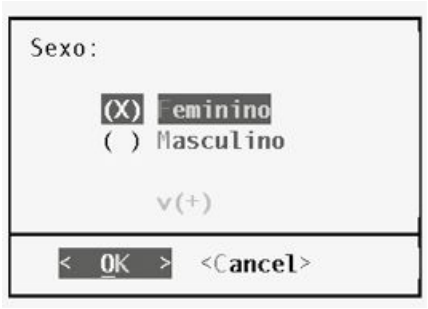
Perceba que as mudanças deste trecho de código para o original do Zuser foram mínimas. Bastou trocar um `read` por uma janela `--inputbox` e trocar um `echo` por uma `--msgbox`. Usar `dialog` é simples assim, pode parecer complicado no início, mas é apenas uma maneira diferente de interagir com o usuário. A lógica de seu programa continua a mesma, inalterada.

Neste ponto, temos um login novo para cadastrar no banco de dados. Agora é preciso obter os outros dados deste usuário, um por vez:

```
48              # Ok, é um usuário novo, prossigamos
49              nome=$(dialog --stdout --inputbox "Nome
 completo:" 0 0)
50              idade=$(dialog --stdout --inputbox "Idade:" 0 0)
51              sexo=$(dialog --stdout --radiolist "Sexo:" 0 0 3
 \
52                  Feminino "" on Masculino "" off)
53              sexo=$(echo $sexo | cut -c1)
54
```

O nome e a idade são obtidos da mesma maneira que o login, usando a

caixa de texto da janela `--inputbox`. Já o sexo aceita apenas dois valores no banco: M e F. Poderia ter sido usada mais uma caixa de texto, mas para evitar que o usuário digite letras inválidas e, ao mesmo tempo, tornar a interface mais amigável, foi usada uma `--radiolist`. Essa janela mostra uma lista de opções na tela, e o usuário escolhe uma.

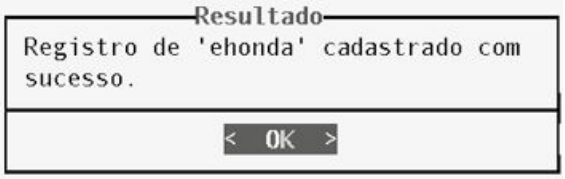


O comando usa o mesmo formato da janela `--menu`, colocando-se os pares de itens no final da linha. Neste caso as descrições foram deixadas vazias e apenas as palavras-chave Feminino e Masculino foram usadas. No banco apenas a letra inicial é usada, então a última linha serve para extraí-la com o `cut`.

O `cut` poderia estar diretamente no final da linha anterior, ficando `sexo=$(dialog | cut)`. Foram usados comandos separados para facilitar o entendimento.

Nesse ponto do programa temos todos os dados do usuário novo já guardados em variáveis, então agora basta adicioná-lo no banco de dados. A única mudança aqui é novamente trocar o antigo `echo` por uma `--msgbox`:

```
55                # Dados obtidos, hora de mascarar eventuais dois-
  pontos
56                nome=$(echo $nome | mascara)
57
58                # Tudo pronto, basta inserir
59                msg=$(insere_registro
  "$login:$nome:$idade:$sexo")
60                dialog --title "Resultado" --msgbox "$msg" 6 40
61            ;;
62
```



Calma, só mais um item e terminamos... Se quiser aproveitar para tomar uma água, ir ao banheiro ou respirar um ar fresco, esta é a hora. A remoção de usuário vai envolver um pouco de mágica, então é bom que sua concentração esteja renovada. Vai lá, eu espero.

Opa, já de volta? Então vamos.

```
63            remove)
64                # Obtém a lista de usuários
65                usuarios=$(pega_campo 1,2 | sed 1d)
66                usuarios=$(echo "$usuarios" | sed 's/:/
  "/;s/$/"/')
67
68                login=$(eval dialog --stdout \
69                    --menu \"Escolha o usuário a remover\" \
70                    0 0 0 $usuarios)
71
72                msg=$(apaga_registro "$login")
73                dialog --title "Resultado" --msgbox "$msg" 6 40
74            ;;
75        esac
```

A ideia aqui é compor um menu com todos os usuários já cadastrados, mostrando seu login e nome completo. Ao escolher um item do menu, esse usuário será removido.

O problema é que a lista dos usuários é dinâmica, deve ser obtida quando o programa estiver rodando. Assim o comando dialog também deve ser dinâmico, para que os pares de itens sejam colocados no final da linha de comando. Bastaria guardar os dados em uma variável e usá-la no dialog, isso funciona perfeitamente quando os itens não possuem espaços em branco. Mas como o nome completo tem várias palavras...

```
$ itens_normal="1 um 2 dois"
$ itens_espaco="1 um 2 'dois espaçado'"
$ dialog --menu "Teste" 0 0 0 $itens_normal   # funciona
$ dialog --menu "Teste" 0 0 0 $itens_espaco   # não funciona
Error: Expected 2 arguments, found only 1.
Use --help to list options.
$ dialog --menu "Teste" 0 0 0 "$itens_espaco"   # não funciona
Error: Expected at least 6 tokens for --menu, have 5.
Use --help to list options.
$
```

Com ou sem aspas, a expansão de variáveis e o dialog não conversam bem neste caso. Ele espera receber cada item do menu como uma dupla de argumentos, e o espaço em branco acaba confundindo tudo. A solução aqui é fazer um pré-processamento, colocando aspas ao redor das palavras espaçadas, e usar o `eval` para que o dialog enxergue essa string como uma linha de comando normal. Acompanhe:

```
$ itens="1 um 2 'dois espaçado'"
$ echo dialog --menu "Teste" 0 0 0 $itens
dialog --menu Teste 0 0 0 1 um 2 'dois espaçado'
$ eval dialog --menu "Teste" 0 0 0 $itens
```

Agora funciona. O `echo` mostra como fica a linha de comando com a variável expandida. Lá estão as aspas que garantem o número correto de argumentos passados para o dialog. Usando o `eval` essa linha é então executada.

Já vimos que o uso do `eval` é perigoso. Todas as recomendações de segurança devem ser levadas em conta neste caso também. Sempre que usar o eval pense na origem dos dados que ele vai executar e certifique-se de que eles não terão subshells ou variáveis que possam ser expandidas. Na dúvida, remova os caracteres perigosos antes do `eval`.

Voltando ao código, a lista de usuários é composta, já com as aspas, nas linhas 65 e 66. Está em dois passos para facilitar o entendimento. Primeiro obtém a lista, depois coloca as aspas. Para ilustrar melhor esse processo, basta executá-lo diretamente na linha de comando:

```
$ pega_campo 1,2 | sed 1d
junior:Alfredo Vieira Júnior
dani:Daniele Almeida Teles
luciana:Luciana Silveira
ana:Ana Paula de Souza
ze:José Carlos Vaz
luigi:Luigi da Silva
guybrush:Guybrush Threepwood
ehonda:Edmond Honda
```

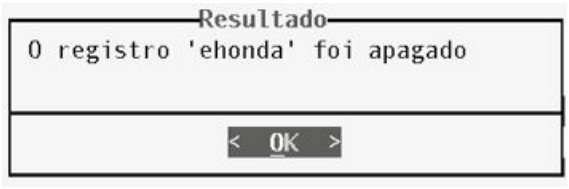
```
$ pega_campo 1,2 | sed 1d | sed 's/:/ "/;s/$/"/'
junior "Alfredo Vieira Júnior"
dani "Daniele Almeida Teles"
luciana "Luciana Silveira"
ana "Ana Paula de Souza"
ze "José Carlos Vaz"
luigi "Luigi da Silva"
guybrush "Guybrush Threepwood"
ehonda "Edmond Honda"
```

Usamos a boa e velha `pega_campo` para obter o login e o nome completo de todos os usuários cadastrados no banco. Lembre-se de que na primeira linha estão os nomes dos campos, por isso ela é apagada no primeiro `sed`. Já o segundo `sed` pega esta listagem e converte para o formato dos pares de palavra-chave/descrição do menu do dialog, usando aspas para proteger os nomes com espaços. Tranquilo, certo?

Voltando de volta novamente mais uma vez ao código, da linha 68 até a 70 é composto o comando que mostra a janela de menu na tela, usando a técnica do `eval` que acabamos de conhecer. Tá, eu vou facilitar a sua vida mais uma vez mostrando o comando inteiro, mas não se acostume ;)

```
dialog --stdout --menu "Escolha o usuário a remover" 0 0 0 junior "Alfredo Vieira Júnior" dani "Daniele Almeida Teles" luciana "Luciana Silveira" ana "Ana Paula de Souza" ze "José Carlos Vaz" luigi "Luigi da Silva" guybrush "Guybrush Threepwood" ehonda "Edmond Honda"
```

Executado este trecho, o menu é mostrado e o usuário escolhe quem ele quer remover. O código seguinte das linhas 72 e 73 encarrega-se de retirar o cadastro do banco e mostrar a mensagem informativa no final. Por falar em final, fechamos o `case` e acabou o `Zuserd`. Ufa!



Percebeu que não é mais necessário o item "*" dentro do `case`? Como não é mais o usuário quem digita a ação, não é preciso checar se há alguma ação inválida. Essa é uma das vantagens do uso do dialog: como o usuário vê apenas menus e opções predefinidas, várias checagens podem ser abandonadas, não é preciso prever o que o usuário criativo pode fazer para quebrar seu programa. O programa roda dentro de um ambiente restrito onde é o programador quem define o que é possível fazer.

Bem, já faz algumas páginas que começamos a codificar este aplicativo. É bem interessante ver a saída do comando `diff` para entender quais foram todas as mudanças necessárias para fazer um programa normal (com opções de linha de comando e `read`) tornar-se um aplicativo amigável para o usuário, com telas que guiam o seu caminho (`dialog`):

Zuser versus Zuserd (diff)

```
--- zuser    2007-11-13 20:03:48.000000000 -0200
+++ zuserd    2007-11-13 22:53:06.000000000 -0200
@@ -1,99 +1,75 @@
#!/bin/bash
#
```

```
-# zuser
+# zuserd
# Lista, adiciona e remove usuários do sistema Z
#
-# Requisitos: bantex.sh
-#
-# Uso: zuser [ lista | adiciona | remove ]
+# Requisitos: bantex.sh, dialog
#
# 2006-10-31 Fulano da Silva
-
-# Se não passar nenhum argumento, mostra a mensagem de ajuda
-[ "$1" ] || {
-    echo
-    echo "Uso: zuser [ lista | adiciona | remove ]"
-    echo
-    echo "   lista    - Lista todos os usuários do sistema"
-    echo "   adiciona - Adiciona um usuário novo no sistema"
-    echo "   remove   - Remove um usuário do sistema"
-    echo
-    exit 0
-}
-
# Localização do arquivo do banco de dados
BANCO=usuarios.txt
# Inclui o gerenciador do banco
source bantex.sh || {
    echo "Ops, ocorreu algum erro no gerenciador do banco"
    exit 1
}
+acao=$( dialog --stdout \
+    --menu "Aplicativo Zuserd - Interface amigável" \
+    0 0 0 \
+    lista "Lista todos os usuários do sistema" \
+    adiciona "Adiciona um usuário novo no sistema" \
+    remove "Remove um usuário do sistema")
+
# Lida com os comandos recebidos
-case "$1" in
```

```
+case "$acao" in
    lista)
        # Lista dos logins (apaga a primeira linha)
-        pega_campo 1 | sed 1d
+        temp=$(mktemp -t lixo)
+        pega_campo 1 | sed 1d > "$temp"
+        dialog --title "Usuários" --textbox "$temp" 13 30
+        rm $temp
    ;;
    adiciona)
-        echo -n "Digite o login do usuário novo: "
-        read login
-
-        # Temos algo?
-        [ "$login" ] || {
-            echo "O login não pode ser vazio, tente novamente."
-            exit 1
-        }
+        login=$(dialog --stdout --inputbox "Digite o login:" 0 0)
+        [ "$login" ] || exit 1
        # Primeiro confere se já não existe esse usuário
        tem_chave "$login" && {
-            echo "O usuário '$login' já foi cadastrado."
+            msg="O usuário '$login' já foi cadastrado."
+            dialog --msgbox "$msg" 6 40
            exit 1
        }

        # Ok, é um usuário novo, prossigamos
-        echo -n "Digite o nome completo: "
-        read nome
-        echo -n "Digite a idade: "
-        read idade
-        echo -n "É do sexo masculino ou feminino? [MF] "
-        read sexo
-        echo
+        nome=$(dialog --stdout --inputbox "Nome completo:" 0 0)
+        idade=$(dialog --stdout --inputbox "Idade:" 0 0)
+        sexo=$(dialog --stdout --radiolist "Sexo:" 0 0 3 \
```

```
+            Feminino "" on Masculino "" off)
+        sexo=$(echo $sexo | cut -c1)

        # Dados obtidos, hora de mascarar eventuais dois-pontos
        nome=$(echo $nome | mascara)

        # Tudo pronto, basta inserir
-        insere_registro "$login:$nome:$idade:$sexo"
-        echo
+        msg=$(insere_registro "$login:$nome:$idade:$sexo")
+        dialog --title "Resultado" --msgbox "$msg" 6 40
    ;;
    remove)
-        # Primeiro mostra a lista de usuários, depois pergunta
-        echo "Lista dos usuários do sistema Z:"
-        pega_campo 1 | sed 1d | tr \\n ' '
-        echo
-        echo
-        echo -n "Qual usuário você quer remover? "
-        read login
-        echo
+        # Obtém a lista de usuários
+        usuarios=$(pega_campo 1,2 | sed 1d)
+        usuarios=$(echo "$usuarios" | sed 's/:/ "/;s/$/"/')
+
+        login=$(eval dialog --stdout \
+            --menu \"Escolha o usuário a remover\" \
+            0 0 0 $usuarios)

-        # Vamos apagar ou puxar a orelha?
-        if tem_chave "$login"; then
-            apaga_registro "$login"
-        else
-            echo "Não, não, esse usuário não está aqui..."
-        fi
-        echo
-    ;;
-
-    *)
```

```
-        # Qualquer outra opção é erro
-        echo "Opção inválida $1"
-        exit 1
+        msg=$(apaga_registro "$login")
+        dialog --title "Resultado" --msgbox "$msg" 6 40
    ;;
esac
```

Cá entre nós, não é tão difícil nem trabalhoso. Agora você não tem mais desculpa para não fazer um programa amigável, pois sabe que não é tanto trabalho a mais. Na verdade, o programa até ficou menor! As 99 linhas do zuserd tradicional tornaram-se 75 linhas no zuserd com dialog. Para o seu deleite visual, aqui está o código completo do programa amigável:

⌨ Zuserd

```
1       #!/bin/bash
2       #
3       # zuserd
4       # Lista, adiciona e remove usuários do sistema Z
5       #
6       # Requisitos: bantex.sh, dialog
7       #
8       # 2006-10-31 Fulano da Silva
9
10       # Localização do arquivo do banco de dados
11       BANCO=usuarios.txt
12
13       # Inclui o gerenciador do banco
14       source bantex.sh || {
15           echo "Ops, ocorreu algum erro no gerenciador do
 banco"
16           exit 1
17       }
18
19       acao=$( dialog --stdout \
20           --menu "Aplicativo Zuserd - Interface amigável" \
21           0 0 0 \
22           lista "Lista todos os usuários do sistema" \
23           adiciona "Adiciona um usuário novo no sistema" \
```

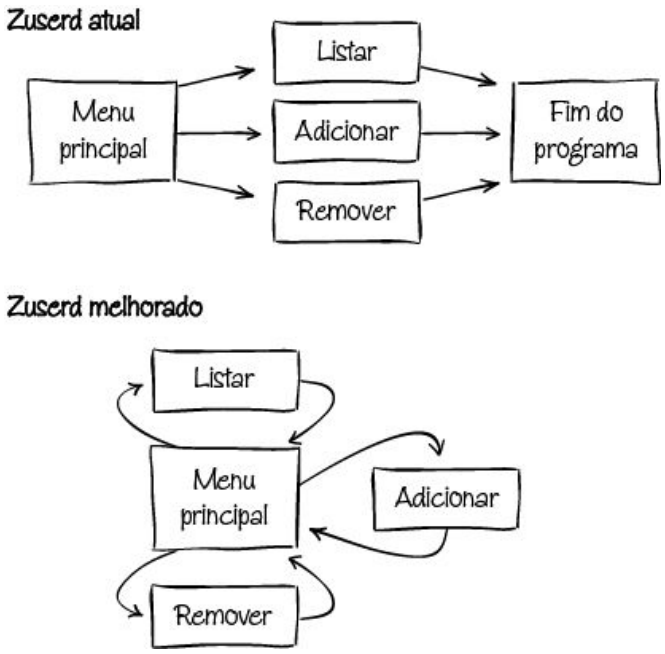
```
24            remove "Remove um usuário do sistema")
25
26        # Lida com os comandos recebidos
27        case "$acao" in
28
29            lista)
30                # Lista dos logins (apaga a primeira linha)
31                temp=$(mktemp -t lixo)
32                pega_campo 1 | sed 1d > "$temp"
33                dialog --title "Usuários" --textbox "$temp" 13 30
34                rm $temp
35            ;;
36
37            adiciona)
38                login=$(dialog --stdout --inputbox "Digite o
 login:" 0 0)
39                [ "$login" ] || exit 1
40
41                # Primeiro confere se já não existe esse usuário
42                tem_chave "$login" && {
43                    msg="O usuário '$login' já foi cadastrado."
44                    dialog --msgbox "$msg" 6 40
45                    exit 1
46                }
47
48                # Ok, é um usuário novo, prossigamos
49                nome=$(dialog --stdout --inputbox "Nome
 completo:" 0 0)
50                idade=$(dialog --stdout --inputbox "Idade:" 0 0)
51                sexo=$(dialog --stdout --radiolist "Sexo:" 0 0 3
 \
52                    Feminino "" on Masculino "" off)
53                sexo=$(echo $sexo | cut -c1)
54
55                # Dados obtidos, hora de mascarar eventuais dois-
 pontos
56                nome=$(echo $nome | mascara)
57
58                # Tudo pronto, basta inserir
59                msg=$(insere_registro
```

```
  "$login:$nome:$idade:$sexo")
60                dialog --title "Resultado" --msgbox "$msg" 6 40
61            ;;
62
63            remove)
64                # Obtém a lista de usuários
65                usuarios=$(pega_campo 1,2 | sed 1d)
66                usuarios=$(echo "$usuarios" | sed 's/:/
  "/;s/$/"/')
67
68                login=$(eval dialog --stdout \
69                    --menu \"Escolha o usuário a remover\" \
70                    0 0 0 $usuarios)
71
72                msg=$(apaga_registro "$login")
73                dialog --title "Resultado" --msgbox "$msg" 6 40
74            ;;
75        esac
```

Zuserd melhorado

Lembra de que um programa nunca tem versão final? Pois é, acabamos de terminar a versão inicial do Zuserd, mas já há melhoramentos importantes a se fazer. As mudanças são poucas, mas tornam a experiência do usuário mais agradável.

O aplicativo como está hoje, mostra o menu, faz o que tem que fazer e é finalizado. Se o usuário quiser executar outra ação terá que rodar o programa novamente. Quando ele ainda era um aplicativo de linha de comando, este era seu comportamento esperado. Mas agora que temos um menu principal, é ele quem deve comandar o fluxo de operações. Isso significa que, após completar cada ação, o programa deve voltar ao menu. Caso o usuário decida sair do menu, somente então o programa será finalizado.



Fluxo do Zuserd melhorado

Mas como sair do menu se ainda não estamos tratando do botão Cancelar? Este é o segundo melhoramento: tratar corretamente os botões Cancelar de todas as janelas. E ainda tem a tecla Esc, que também pode ser usada caso o usuário desista de uma operação.

De volta ao editor de textos, vamos fazer o menu ser o centro das atenções. Cada ação está registrada dentro do case e nosso objetivo é fazer com que a execução volte para o menu logo em seguida. Pense comigo: mostra menu, faz ação, mostra menu, faz ação, ... Isso me parece um loop. Concorda? Pois é, então basta fazer o aplicativo todo ser um loop:

```
...
# Localização do arquivo do banco de dados
BANCO=usuarios.txt

# Inclui o gerenciador do banco
source bantex.sh || { ... }

while :
do
    ... # mostra menu
    case "$acao" in
        lista) ...
```

```
        adiciona) ...
        remove) ...
    esac
done
```

Com apenas três linhas a mais no Zuserd, fizemos dele um aplicativo de execução constante. O "`while :`" indica um loop infinito, pois é uma condição sempre verdadeira. Dentro deste loop, primeiro é mostrado o menu e em seguida executada a ação. Este ciclo repete-se indefinidamente. Espere. Indefinidamente? Mas uma hora o usuário vai querer sair do programa. Precisamos de um `break` em algum lugar...

Missão: Estando no menu principal, se o usuário apertar o botão Cancelar ou a tecla Esc, o programa deve ser finalizado.

Mas como saber que estes eventos aconteceram? Eu ainda não contei, mas o dialog usa o código de retorno para indicar qual foi o botão ou tecla apertados pelo usuário. Até agora como apenas tratamos do botão OK, isso não era importante. Agora é hora de aprender.

Você já sabe que o código de retorno de um programa é sempre guardado na variável especial `$?`. Geralmente este código é zero quando o programa termina sem erros e qualquer outro numeral indica um tipo de erro. No dialog é diferente, cada número indica qual o botão ou tecla apertados para sair da janela. Veja um exemplo rápido:

```
$ dialog --yesno "sim ou não?" 0 0 ; echo Retorno: $?
```

Ao apertar o botão Yes, o retorno mostrado é zero. Se for o botão No, o retorno é um. No caso da tela de menu em que os botões são OK e Cancel, os códigos também são 0 e 1, respectivamente. Se a tecla Esc for apertada, o retorno é 255. Há também o botão Help que ainda não vimos, seu código é 2. É fácil tratar de todos eles de uma vez, com um único comando `case`:

```
case $? in
      0) echo O usuário apertou o botão OK / Yes ;;
      1) echo O usuário apertou o botão Cancel / No ;;
      2) echo O usuário apertou o botão Help ;;
    255) echo O usuário apertou a tecla Esc ;;
      *) echo Retorno desconhecido ;;
```

```
esac
```

Mas neste momento não precisamos ser tão detalhistas. O que nos interessa são apenas duas condições:

- O usuário continuou (botões OK ou Yes).
- O usuário desistiu (botões Cancel ou No, tecla Esc).

Para isso basta um único teste: caso o código de retorno seja diferente de zero, o usuário desistiu e precisamos alterar o fluxo normal do programa. Em outras palavras:

```
[ $? -ne 0 ] && mutley_faca_alguma_coisaaa
```

Voltando à missão da vez, é preciso sair do programa caso o usuário desista quando estiver no menu. Agora que já sabemos como lidar com botões e teclas, fica muito fácil. Basta testar o código de retorno logo após a aparição do menu, e caso seja uma desistência (Cancel ou Esc), finalizamos o programa:

```
acao=$( dialog --stdout \
    --title " Aplicativo Zuserd - Interface amigável " \
    --menu "" \
    0 0 0 \
    lista "Lista todos os usuários do sistema" \
    adiciona "Adiciona um usuário novo no sistema" \
    remove "Remove um usuário do sistema")
[ $? -ne 0 ] && exit
```

Ah, se sempre fosse assim... Só adicionar mais uma linha e o problema está resolvido. Nesse caso, felizmente, é apenas isso mesmo.

O que temos até aqui, então? Vamos recapitular. Nosso aplicativo é iniciado, mostra o menu principal e, dependendo da escolha do usuário, executa uma das três ações disponíveis e volta ao menu. Caso o usuário desista no menu, o programa acaba. Está quase lá, só o que está faltando agora é fazer com que o programa volte ao menu, caso o usuário desista no meio de uma das operações.

- **Operação listar:** É apenas uma única janela com um botão, qualquer que seja a ação do usuário o programa deve voltar ao menu e isto já está sendo feito. Nada a alterar.

- **Operação adicionar:** Há várias telas pedindo informações do usuário novo e em todas há um botão Cancelar que deve ser respeitado (além da tecla Esc, claro). Em qualquer dessas telas, a desistência deve fazer o programa voltar ao menu e os dados já informados serão descartados, evitando a criação do usuário novo.
- **Operação remover:** Há uma tela inicial que mostra os usuários do sistema e somente depois de escolhido ele é removido. Uma desistência nesta tela deve voltar ao menu principal e nenhum usuário será removido.

Eu poderia escrever mais um parágrafo aqui, reforçando as informações acima, mas isto não será necessário porque você já entendeu o que precisa ser feito, não é? Então sem mais delongas, o código:

```
43          adiciona)
44              login=$(dialog --stdout --inputbox "Digite o
 login:" 0 0)
45              [ $? -ne 0 ] && continue
46              [ "$login" ] || continue
47
48              # Primeiro confere se já não existe esse usuário
49              tem_chave "$login" && {
50                  msg="O usuário '$login' já foi cadastrado."
51                  dialog --msgbox "$msg" 6 40
52                  continue
53              }
54
55              # Ok, é um usuário novo, prossigamos
56              nome=$(dialog --stdout --inputbox "Nome
 completo:" 0 0)
57              [ $? -ne 0 ] && continue
58              idade=$(dialog --stdout --inputbox "Idade:" 0 0)
59              [ $? -ne 0 ] && continue
60              sexo=$(dialog --stdout --radiolist "Sexo:" 0 0 3
 \
61                  Feminino "" on Masculino "" off)
62              [ $? -ne 0 ] && continue
63              sexo=$(echo $sexo | cut -c1)
64
```

```
65                # Dados obtidos, hora de mascarar eventuais dois-
 pontos
66                nome=$(echo $nome | mascara)
67
68                # Tudo pronto, basta inserir
69                msg=$(insere_registro
 "$login:$nome:$idade:$sexo")
70                dialog --title "Resultado" --msgbox "$msg" 6 40
71            ;;
72
73            remove)
74                # Obtém a lista de usuários
75                usuarios=$(pega_campo 1,2 | sed 1d)
76                usuarios=$(echo "$usuarios" | sed 's/:/
 "/;s/$/"/')
77
78                login=$(eval dialog --stdout \
79                    --menu \"Escolha o usuário a remover\" \
80                    0 0 0 $usuarios)
81                [ $? -ne 0 ] && continue
82
83                msg=$(apaga_registro "$login")
84                dialog --title "Resultado" --msgbox "$msg" 6 40
85            ;;
```

As mudanças foram poucas, mas significativas. Todos os testes de desistência executam um `continue`, caso necessário. Assim o loop volta para o início imediatamente, mostrando novamente o menu e interrompendo qualquer ação que estivesse em andamento.

Note que as checagens de login vazio e login já existente na operação de adicionar foram trocadas de `exit 1` para `continue`. Ou seja, se antes saíamos do programa em qualquer erro, agora voltamos ao menu principal e o usuário pode tentar novamente, dessa vez sem repetir o mesmo erro. Ou pelo menos, assim esperamos :)

Zuserd melhorado

```
1       #!/bin/bash
2       #
3       # zuserd
```

```
4       # Lista, adiciona e remove usuários do sistema Z
5       #
6       # Requisitos: bantex.sh, dialog
7       #
8       # 2006-10-31 v1 Fulano da Silva: Versão inicial
9       # 2006-10-31 v2 Fulano da Silva:
10       #   - Tratamento do ESC e dos botões Cancelar
11       #   - Aplicação agora roda em loop
12
13       # Localização do arquivo do banco de dados
14       BANCO=usuarios.txt
15
16       # Inclui o gerenciador do banco
17       source bantex.sh || {
18           echo "Ops, ocorreu algum erro no gerenciador do
 banco"
19           exit 1
20       }
21
22       while :
23       do
24           acao=$( dialog --stdout \
25               --title " Aplicativo Zuserd - Interface amigável
 " \
26               --menu "" \
27               0 0 0 \
28               lista "Lista todos os usuários do sistema" \
29               adiciona "Adiciona um usuário novo no sistema" \
30               remove "Remove um usuário do sistema")
31           [ $? -ne 0 ] && exit
32
33           # Lida com os comandos recebidos
34           case "$acao" in
35
36           lista)
37               # Lista dos logins (apaga a primeira linha)
38               temp=$(mktemp -t lixo)
39               pega_campo 1 | sed 1d > "$temp"
40               dialog --title "Usuários" --textbox "$temp" 13 30
```

```
41                rm $temp
42            ;;
43
44            adiciona)
45                login=$(dialog --stdout --inputbox "Digite o
 login:" 0 0)
46                [ $? -ne 0 ] && continue
47                [ "$login" ] || continue
48
49                # Primeiro confere se já não existe esse usuário
50                tem_chave "$login" && {
51                    msg="O usuário '$login' já foi cadastrado."
52                    dialog --msgbox "$msg" 6 40
53                    continue
54                }
55
56                # Ok, é um usuário novo, prossigamos
57                nome=$(dialog --stdout --inputbox "Nome
 completo:" 0 0)
58                [ $? -ne 0 ] && continue
59
60                idade=$(dialog --stdout --inputbox "Idade:" 0 0)
61                [ $? -ne 0 ] && continue
62
63                sexo=$(dialog --stdout --radiolist "Sexo:" 0 0 3
 \
64                    Feminino "" on Masculino "" off)
65                [ $? -ne 0 ] && continue
66                sexo=$(echo $sexo | cut -c1)
67
68                # Dados obtidos, hora de mascarar eventuais dois-
 pontos
69                nome=$(echo $nome | mascara)
70
71                # Tudo pronto, basta inserir
72                msg=$(insere_registro
 "$login:$nome:$idade:$sexo")
73                dialog --title "Resultado" --msgbox "$msg" 6 40
74            ;;
75
```

```
76            remove)
77                # Obtém a lista de usuários
78                usuarios=$(pega_campo 1,2 | sed 1d)
79                usuarios=$(echo "$usuarios" | sed 's/:/
 "/;s/$/"/')
80
81                login=$(eval dialog --stdout \
82                    --menu \"Escolha o usuário a remover\" \
83                    0 0 0 $usuarios)
84                [ $? -ne 0 ] && continue
85
86                msg=$(apaga_registro "$login")
87                dialog --title "Resultado" --msgbox "$msg" 6 40
88            ;;
89            esac
90        done
```

Pense no usuário

Você consegue perceber como pequenas mudanças podem fazer uma grande diferença para o usuário do programa? A primeira versão do Zuserd funcionava, mas o usuário devia rodá-la novamente a cada operação ou em caso de erro. Você já imaginou se o seu leitor de e-mail se fechasse após mostrar uma mensagem? A segunda versão trouxe poucas mudanças se formos mesurar linhas de código, mas é **outro** programa se formos mesurar a usabilidade.

O programa agora comporta-se como o usuário espera que ele deveria se comportar, a famosa lei da menor surpresa. Ele faz o que tem que fazer, de uma maneira que ajuda o usuário em vez de fazê-lo ter que se adaptar ou aprender algo novo.

Esta é uma barreira que a maioria dos programadores tem uma dificuldade enorme em atravessar. Geralmente se codifica pensando na facilidade de manutenção do código (o centro é o programador) quando deveria se pensar na facilidade de uso (o centro é o usuário). Sim, geralmente é mais trabalhoso e envolve mais código fazer-se algo que fique claro e simples para o usuário. Mas compensa cada bit.

Se o seu programa for desengonçado, feio e difícil de usar, seus usuários passarão esta impressão para frente e seus depoimentos não o ajudarão a aumentar sua base de usuários. Mas se o programa tiver uma interface amigável, ajudar o usuário a usá-lo com o menor esforço possível, sem ter que ler o manual ou procurar na Internet, ele terá uma boa experiência e passará isso adiante, recomendando seu programa a outras pessoas.

Seu programa deve ser uma ferramenta que ajuda, e não uma fonte adicional de problemas para o usuário. Invista mais tempo no polimento, pense onde poderia melhorar, apare as arestas. Se possível, convide familiares, amigos ou outras pessoas não técnicas para tentar usar seu programa. Não dê instruções, veja como eles se comportam, se conseguem descobrir sozinhos. Quanto menos você precisar falar, mais amigável será seu programa.

Se você acha que seus usuários são ignorantes e fazem perguntas óbvias, isso é um sinal de que seu programa não é amigável. Lembre-se que os usuários não possuem o mesmo domínio que você, nem sobre o programa nem sobre a tarefa que ele desempenha. Tente usar seu programa imaginando-se um leigo completo. Bloqueie na mente todo seu conhecimento prévio e tente antecipar dúvidas e enganos que o usuário terá ao tentar usá-lo. Quando você conseguir ir fundo nessa abstração, fará programas mais amigáveis e verá que isso diminuirá seus custos de manutenção:

menos dúvidas = menos suporte = menos tempo perdido

Tudo bem, estou indo muito mais longe do que nosso Zuserd foi, mas se você quer que seus programas tenham sucesso, tente seguir estes conselhos. Afinal, você é um programador e não um scripteiro, certo? :)

Domine o Dialog

O dialog é relativamente simples de usar, mas como ele age um pouco diferente dos outros programas do sistema, pode assustar e parecer confuso em uma primeira tentativa. Aqui vai um resumo de suas características:

- Funciona tanto no modo gráfico (X11) quanto no console.
- A linha de comando é longa, cheia de opções.
- A janela pode ter dimensões fixas (definidas pelo usuário) ou automáticas (calculadas pelo dialog).
- O texto da janela pode ser ajustado automaticamente ou quebras de linha podem ser forçadas usando-se o `\n`.
- Códigos de retorno são usados para identificar o uso da tecla Esc e dos botões Sim/Não, Ok/Cancel e Help.
- A saída de erro (`STDERR`) é usada para mostrar textos digitados e itens escolhidos pelo usuário. Para em seu lugar usar a saída padrão (`STDOUT`), é preciso especificar a opção `--stdout`.
- O texto dos botões e as cores das janelas são configuráveis.
- O dialog simplesmente mostra uma janela, aguarda a ação do usuário e sai. Todo o fluxo e navegação entre janelas deve ser feito manualmente pelo programador.
- A janela de progresso (gauge) recebe a atualização do estado atual via `STDIN`. Exemplo: `(echo 50; sleep 2; echo 100) | dialog --gauge 'abc' 8 40 0`

Exemplos de todas as janelas

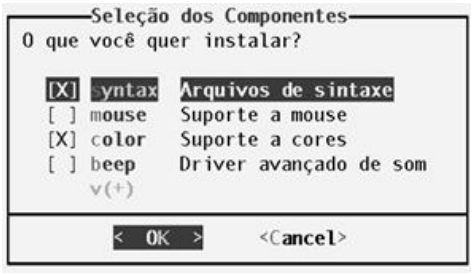
Nome	Desenha uma janela onde o usuário...
calendar	Vê um calendário e escolhe uma data.
checklist	Vê uma lista de opções e escolhe várias.
fselect	Digita ou escolhe um arquivo.
gauge	Vê uma barra de progresso (porcentagem).
infobox	Vê uma mensagem, sem botões.
inputbox	Digita um texto qualquer.
menu	Vê um menu e escolhe um item.
msgbox	Vê uma mensagem e aperta o botão OK.
passwordbox	Digita uma senha.
radiolist	Vê uma lista de opções e escolhe uma.

tailbox	Vê a saída do comando `tail -f`.
tailboxbg	Vê a saída do comando `tail -f` (em segundo plano).
textbox	Vê o conteúdo de um arquivo.
timebox	Escolhe um horário.
yesno	Vê uma pergunta e aperta o botão YES ou o NO.

Calendar

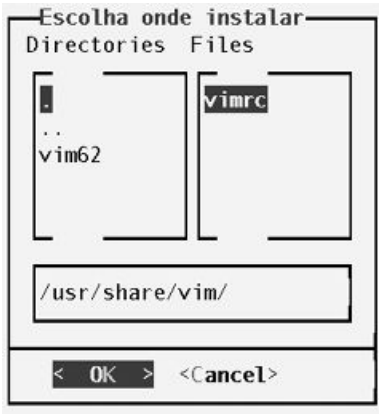

```
dialog                          \
   --title 'Escolha a data'     \
   --calendar ''                \
   0 0                          \
   31 12 1999
```

Checklist



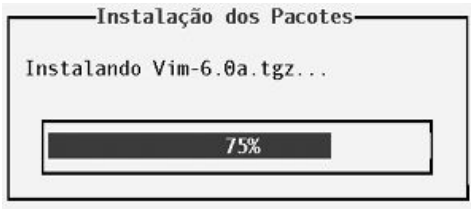

```
dialog                                          \
   --title 'Seleção dos Componentes'            \
   --checklist 'O que você quer instalar?'      \
   0 0 0                                        \
   syntax  'Arquivos de sintaxe'      on        \
   mouse   'Suporte a mouse'          off       \
   color   'Suporte a cores'          on        \
   beep    'Driver avançado de som'   off
```

Fselect



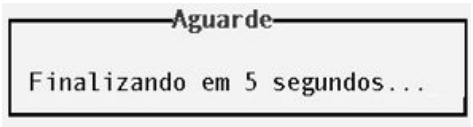

```
dialog                                  \
   --title 'Escolha onde instalar'  \
   --fselect /usr/share/vim/        \
   0 0
```

Gauge




```
( echo 40 ; sleep 1
  echo 75 ; sleep 1
  echo 100; sleep 1 ) |
dialog                                      \
   --title 'Instalação dos Pacotes'         \
   --gauge '\nInstalando Vim-6.0a.tgz...'  \
   8 40 60
```

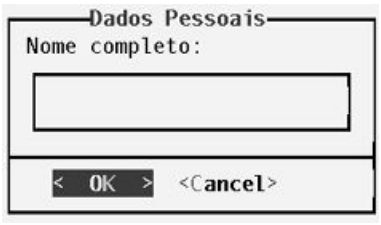
Infobox




```
dialog                                      \
   --title 'Aguarde'                        \
   --sleep 5                                \
```

```
   --infobox '\nFinalizando em 5 segundos...'  \
   0 0
```

Inputbox

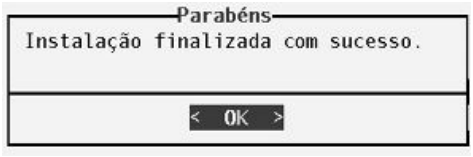



```
dialog                          \
   --title 'Dados Pessoais'     \
   --inputbox 'Nome completo:'  \
   0 0
```

Menu

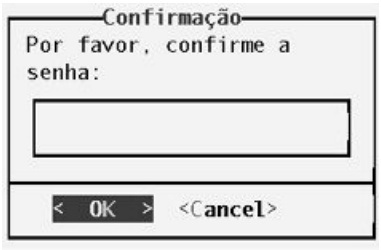

```
dialog                                      \
   --title 'Perfil'                         \
   --menu 'Escolha o perfil da instalação:'  \
   0 0 0                                    \
   mínima       'Instala o mínimo'          \
   completa     'Instala tudo'              \
   customizada  'Você escolhe'
```

Msgbox



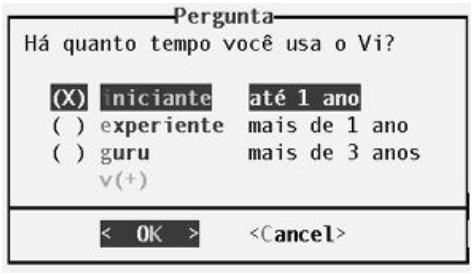

```
dialog                                          \
   --title 'Parabéns'                           \
   --msgbox 'Instalação finalizada com sucesso.'  \
   6 40
```

Passwordbox



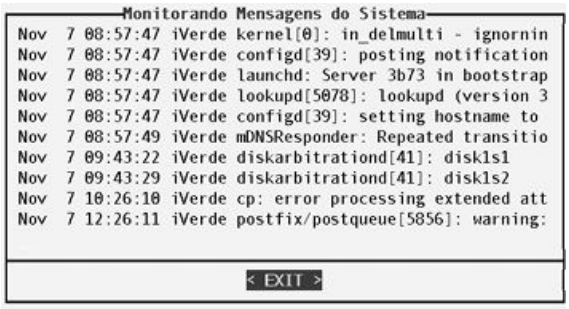

```
dialog                                              \
   --title 'Confirmação'                            \
   --passwordbox 'Por favor, confirme a senha:'     \
   0 0
```

Radiolist




```
dialog                                              \
   --title 'Pergunta'                               \
   --radiolist 'Há quanto tempo você usa o Vi?'     \
   0 0 0                                            \
   iniciante  'até 1 ano'      on                   \
   experiente 'mais de 1 ano'  off                  \
   guru       'mais de 3 anos' off
```

Tailbox, Tailboxbg



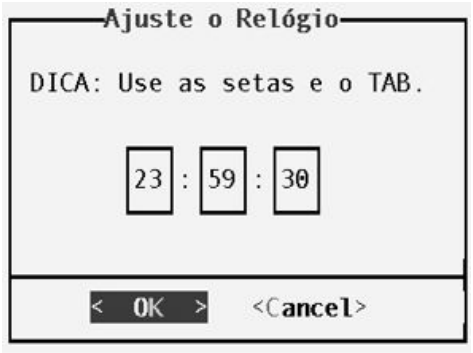

```
tail -f /var/log/messages > out &
```

```
dialog                                          \
   --title 'Monitorando Mensagens do Sistema'   \
   --tailbox out                                \
   15 60
```

Textbox

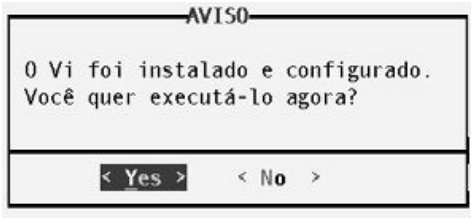

```
dialog                                          \
   --title 'Conteúdo do arquivo /etc/shells' \
   --textbox /etc/shells                        \
   0 0
```

Timebox




```
dialog                                          \
   --title 'Ajuste o Relógio'                   \
   --timebox '\nDICA: Use as setas e o TAB.'    \
   0 0                                          \
   23 59 30
```

Yesno




```
dialog                                                  \
   --title 'AVISO'                                      \
   --yesno '\nO Vi foi instalado e configurado.
            \nVocê quer executá-lo agora?\n\n'  \
   0 0
```

Opções de linha de comando

Opções para definir os textos da caixa

Opção	Descrição
`--backtitle` *texto*	Especifica o título do topo da tela, que fica no plano de fundo, atrás da caixa.
`--title` *texto*	Define o título da caixa, colocado centralizado na borda superior.
`--cancel-label` *texto*	Especifica o texto para ser mostrado no botão Cancel.
`--exit-label` *texto*	Especifica o texto para ser mostrado no botão Exit.
`--help-label` *texto*	Especifica o texto para ser mostrado no botão Help.
`--ok-label` *texto*	Especifica o texto para ser mostrado no botão OK.

Opções para fazer ajustes no texto da caixa

Opção	Descrição
`--cr-wrap`	Mantém as quebras de linha originais do texto da caixa, para não precisar colocar os `\n`. Mas lembre-se que caso a linha fique muito grande, o dialog a quebrará no meio para caber na caixa.
`--no-collapse`	Mantém o espaçamento original do texto, não retirando os TABs nem os espaços em branco consecutivos.
`--tab-correct`	Converte cada TAB para *N* espaços. O *N* é especificado na opção `--tab-len` ou o padrão 8 é assumido.
`--tab-len` *N*	Especifica o número de espaços que serão colocados no lugar de cada TAB, quando usar o opção `--tab-correct`.
`--trim`	Limpa o texto da caixa, apagando espaços em branco no início, espaços consecutivos e quebras de linha literais.

Opções para fazer ajustes na caixa

Opção	Descrição
`--aspect` *taxa*	Taxa que ajusta o dimensionamento automático das caixas. É a relação largura / altura, sendo o padrão 9, que significa 9 colunas para cada linha.
`--begin` *y x*	Especifica a posição inicial da caixa, relativo ao canto superior esquerdo.
`--defaultno`	Faz o botão Não ser o padrão da caixa YesNo.
`--default-item` *item*	Define qual vai ser o item pré-selecionado do Menu. Se não especificado, o primeiro item será o selecionado.
`--shadow`	Desenha a sombra da caixa. Opção já usada normalmente.
`--no-shadow`	Não desenha a sombra da caixa.
`--no-cancel` ou `--nocancel`	Não mostra o botão Cancel nas caixas Checklist, Inputbox e Menu. A tecla Esc continua valendo para sair da caixa.
`--item-help`	Usada nas caixas Checklist, Radiolist ou Menu, mostra uma linha de ajuda no rodapé da tela para o item selecionado. Esse texto é declarado se adicionando uma nova coluna no final da definição de cada item.
`--help-button`	Mostra um botão de Help. Seu código de retorno é 2.

Opções relativas aos dados informados pelo usuário

Opção	Descrição
`--separate-output`	Na caixa Checklist, retorna os itens selecionados, um por linha e sem aspas. Boa opção para usar em seus programas!
`--separate-widget` *sep*	Define o separador que será colocado entre os retornos de cada caixa. Útil quando se trabalha com múltiplas caixas. O separador padrão é o TAB.
`--stderr`	Retorna os dados na saída de erros (`STDERR`). Opção já usada normalmente.
`--stdout`	Retorna os dados na saída padrão (`STDOUT`) em vez da `STDERR`.
`--max-input` *tamanho*	Tamanho máximo do texto que o usuário pode digitar nas caixas. O tamanho padrão é 2000 caracteres.

Outras opções

Opção	Descrição
`--ignore`	Ignora as opções inválidas. Serve para manter compatibilidade apenas.
`--size-err`	Opção antiga que não é mais usada.

`--beep`	Apita cada vez que a tela é desenhada.
`--beep-after`	Apita na saída com o Ctrl+C.
`--sleep` *N*	Faz uma pausa de N segundos após processar a caixa. Útil para a Infobox.
`--timeout` *N*	Sai do programa com erro caso o usuário não faça nada em *N* segundos.
`--no-kill`	Coloca a caixa Tailboxbg em segundo plano (desabilitando seu `SIGHUP`) e mostra o ID de seu processo na `STDERR`.
`--print-size`	Mostra o tamanho de cada caixa na `STDERR`.
`--and-widget`	Junta uma ou mais caixas numa mesma tela (sem limpá-la).

Opções que devem ser usadas sozinhas na linha de comando

Opção	Descrição
`--clear`	Restaura a tela caso o dialog a tenha bagunçado.
`--create-rc` *arquivo*	Gera uma arquivo de configuração do dialog.
`--help`	Mostra a ajuda do dialog, com as opções disponíveis.
`--print-maxsize`	Mostra o tamanho atual da tela na `STDERR`.
`--print-version`	Mostra a versão do dialog na `STDERR`.
`--version`	O mesmo que `--print-version`.

Parâmetros obrigatórios da linha de comando

No dialog, é obrigatório passar o texto e o tamanho da caixa, sempre. Com isso, a cada chamada do programa, deve haver pelo menos quatro opções na linha de comando. O formato genérico de chamada é:

```
dialog --tipo-da-caixa <texto> <altura> <largura>
```

Opção	Descrição
texto	O texto é a palavra ou frase que aparece no início da caixa, logo após a primeira linha (borda superior). Passe uma string vazia "" caso não deseje texto. Caso o texto seja maior que o tamanho da janela, ele será ajustado automaticamente, quebrando a linha. Para colocar as quebras de linhas manualmente, insira um `\n` onde desejar as quebras. Exemplo: `"Primeira linha.\nSegunda."`
altura	A altura é o número de linhas que serão utilizadas para desenhar a caixa, inclusive a primeira e a última que fazem as bordas superior e inferior. Se informado o número zero, o dialog ajusta automaticamente a altura da caixa para caber o conteúdo.

largura	A largura é o número de colunas que serão utilizadas para desenhar a caixa, inclusive a primeira e a última que fazem as bordas esquerda e direita. Se informado o número zero, o dialog ajusta automaticamente a largura da caixa para caber o conteúdo.

Na prática, é melhor deixar que o dialog quebre o texto e ajuste o tamanho das caixas automaticamente. Então evite usar quebras de linha manuais (\n) e sempre especifique os tamanhos como 0 0 (zero zero).

As caixas do tipo Menu (Menu, Radiolist, Checklist), precisam de linhas de comando mais complexas e têm outras opções obrigatórias além das comuns para todas as caixas. Este é o formato genérico da linha de comando da caixa Menu:

```
dialog --menu <texto> 0 0 <núm-itens> <item1> <desc1> ... <itemN>
  <descN>
```

Opção	Descrição
núm-itens	O número máximo de itens do menu que serão mostrados na caixa. Os demais ficarão ocultos e podem ser acessados rolando a lista com as setas do teclado. Caso especificado como zero, o dialog mostra todos os itens, ou ajusta automaticamente o número ideal para que a caixa caiba na tela.
item	O item deve ser um nome único, diferente para cada item. O item é o texto retornado pelo dialog ao script, quando o usuário escolhe uma opção.
descrição	A descrição é um texto explicativo que serve para detalhar do que se trata o item. A descrição pode ser omitida passando a string vazia "". Exemplo: `dialog --menu 'texto' 0 0 0 item1 "" item2 "" item3 ""`

Se for uma caixa do tipo Checklist ou Radiolist, ainda se deve colocar mais um argumento após cada item do menu, que é ON ou OFF, para dizer se o item já aparecerá selecionado ou não.

Respostas e ações do usuário

- Para saber qual botão o usuário apertou, confira o valor de retorno (*exit code*):

```
$ dialog --yesno 'sim ou não?' 0 0 ; echo Retorno: $?
```

Zero para Sim, um para Não. Assim como todos os outros comandos do sistema, se retornar zero é por está tudo Ok. Na prática, basta usar o `if` para testar o valor do `$?`:

```
dialog --yesno 'Quer ver as horas?' 0 0
if [ $? = 0 ]; then
```

```
    echo "Agora são: $( date )"
else
    echo "Ok, não vou mostrar as horas."
fi
```

Ou usar o operador AND para definir o valor de uma chave do seu programa:

```
dialog --yesno 'Mostrar arquivos ocultos?' 0 0 &&
mostrar_ocultos=1
```

- Para testar todos os botões de uma só vez, bem como a tecla Esc, faça:

```
case $? in
      0) echo O usuário apertou o botão OK / Yes ;;
      1) echo O usuário apertou o botão Cancel / No ;;
      2) echo O usuário apertou o botão Help ;;
    255) echo O usuário apertou a tecla Esc ;;
      *) echo Retorno desconhecido ;;
esac
```

- Para obter o texto digitado pelo usuário:

```
nome=$( dialog --stdout --inputbox 'Digite seu nome:' 0 0 )
echo "O seu nome é: $nome"
```

- Para obter o Item Escolhido (Menu, Radiolist):

```
cor=$( dialog --stdout --menu 'As cores:' 0 0 0 \
       amarelo 'a cor do sol' \
       verde 'a cor da grama' \
       azul 'a cor do céu' )
echo Você escolheu a cor $cor
```

- Para obter Itens Múltiplos (Checklist):

```
estilos=$( dialog --stdout
      --checklist 'Você gosta de:' 0 0 0 \
    rock  '' ON  \
    samba '' OFF \
    metal '' ON  \
    jazz  '' OFF \
    pop   '' ON  \
    mpb   '' OFF )
echo "Você escolheu: $estilos"
```

Note que para o Checklist todos os itens escolhidos são retornados em uma única linha, protegidos por aspas. Para ficar mais fácil extrair tais dados, existe a opção `--separate-output`, que retorna os itens um por linha e sem as aspas:

```
estilos=$( dialog --stdout --separate-output \
      --checklist 'Você gosta de:' 0 0 0 \
    rock  '' ON  \
    samba '' OFF \
    metal '' ON  \
    jazz  '' OFF \
    pop   '' ON  \
    mpb   '' OFF )
echo "$estilos" | while read LINHA; do echo "--- $LINHA"; done
```

Tipos de navegação entre telas

Dependendo do tipo de sua aplicação, a tecla Esc pode gerar o mesmo procedimento que apertar o botão Cancelar geraria. Ou, ainda, pode-se ter dois procedimentos diferentes, um para cada evento. Tudo depende do tipo de navegação que seu programa utiliza, algumas sugestões:

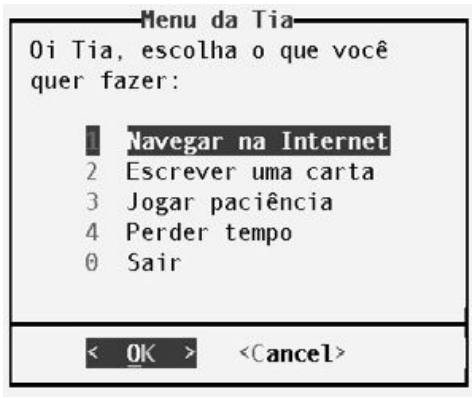
Navegação amarrada a um Menu Principal:

- Se apertar Cancelar no Menu Principal, sai do programa.
- Se apertar Cancelar em uma tela secundária, volta ao Menu Principal.
- Se apertar Esc em qualquer tela, sai do programa.

Navegação tipo Ida e Volta:

- Se apertar Cancelar volta à tela anterior.
- Se apertar Esc sai do programa.

Menu amarrado (em loop)

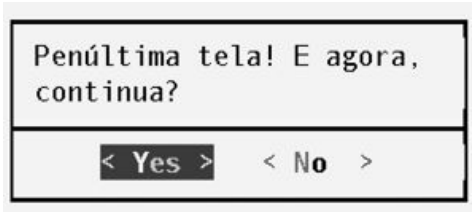


⌨ tia.sh

```
#!/bin/bash
# tia.sh - o script da tia que precisa usar o computador
#
# Exemplo de como amarrar o script em um menu principal usando
# o 'while'. O 'case' é usado para identificar qual foi a ação
# escolhida. Após cada ação, ele sempre retorna ao menu
# principal. Só sai do script caso escolha a última opção,
# aperte CANCELAR ou ESC.
#
# Útil para usar como login shell de pessoas inexperientes ou
# fazer utilitários de ações restritas e definidas.
#
# FLUXOGRAMA
#                       INÍCIO                    FIM
#                    +-----------+           +----------+
#          +------> |    menu   |--Esc-----> |  sai do  |
#          |        | principal |--Cancel--> | programa |
#          |        +-----Ok----+       +--> +----------+
#          |              |             |
#          +--<--1 2 3-4--+--Zero--->---+
# Loop que mostra o menu principal
while : ; do
    # Mostra o menu na tela, com as ações disponíveis
    resposta=$(
      dialog --stdout               \
             --title 'Menu da Tia'  \
             --menu 'Oi Tia, escolha o que você quer fazer:' \
            0 0 0                   \
```

```
            1 'Navegar na Internet' \
            2 'Escrever uma carta'  \
            3 'Jogar paciência'     \
            4 'Perder tempo'        \
            0 'Sair'                )
    # Ela apertou CANCELAR ou ESC, então vamos sair...
    [ $? -ne 0 ] && break
    # De acordo com a opção escolhida, dispara programas
    case "$resposta" in
         1) firefox 'http://google.com.br' ;;
         2) nano /tmp/carta.txt ;;
         3) /usr/games/solitaire ;;
         4) xsnow ; xeyes ;;
         0) break ;;
    esac
done
echo 'Tchau Tia!'       # Mensagem final :)
```

Telas encadeadas (navegação sem volta)



encadeado.sh

```
#!/bin/bash
# encadeado.sh - o script que chega até o final
#
# Exemplo de como encadear telas usando o operador && (AND).
# Caso o usuário desista em qualquer tela (apertando CANCELAR
# ou ESC), o script executa o primeiro comando após a cadeia
# de &&.
#
# Útil para fazer programas ou brincadeiras onde só há um
# caminho certo a seguir para chegar ao final.
#
# FLUXOGRAMA
#             INÍCIO
```

```
#            +-------+
#            | tela1 |--Cancel/Esc--->---+
#            +--Ok---+                   |
#            | tela2 |--Cancel/Esc--->---+     +----------+
#            +--Ok---+                   |---> | desistiu |
#            | tela3 |--Cancel/Esc--->---+     +----------+
#            +--Ok---+                   |
#            | tela4 |--Cancel/Esc--->---+
#            +--Ok---+
#            | final |
#            +-------+
#               FIM
#
# Função rápida para chamar a caixa YesNo
simnao(){
    dialog --yesno "$*" 0 0
}
# Aqui começa o encadeamento de telas com o &&.
# Somente apertando o botão OK vai para a próxima tela.
# Há um 'exit' no final, que sai do script caso o usuário
# tenha chegado até o fim da cadeia.
simnao 'Quer continuar?'                    &&
simnao 'Estamos na segunda tela. Continua?' &&
simnao 'Terceira. Continua continuando?'    &&
simnao 'Penúltima tela! E agora, continua?' &&
echo 'Você chegou até o final!'             && exit
# Este trecho já não faz mais parte do encadeamento e só
# será alcançado caso o usuário tenha apertado CANCELAR/Esc.
echo Você desistiu antes de chegar no final...
```

Navegação completa (ida e volta)

navegando.sh

```
#!/bin/bash
# navegando.sh - o script que vai e volta
#
# Exemplo de como ligar todas as telas do programa entre si,
# guardando informações de ida e volta. O botão CANCELAR faz
# voltar para a tela anterior e o OK faz ir à próxima. Para
# sair do programa a qualquer momento basta apertar o ESC.
#
# Útil para fazer programas interativos, de contexto, ou que
# se pode voltar para corrigir informações.
#
# FLUXOGRAMA
#                    INÍCIO
#                 +-----------+
#                 | primeira  |--Esc--->---+
#      .--------> +----Ok-----+            |
#      `--Cancel--|   nome    |--Esc--->---+
#      .--------> +----Ok-----+            |     +----------+
#      `--Cancel--|   idade   |--Esc--->---+---> |  Sai do  |
#      .--------> +----Ok-----+            |     | Programa |
#      `--Cancel--| est.civil |--Esc--->---+     +----------+
#      .--------> +----Ok-----+            |
#      `--Cancel--|   gostos  |--Esc--->---+
#                 +----Ok-----+
```

```
#                  |   final   |
#                  +-----------+
#                       FIM
#
proxima='primeira'

# loop principal
while : ; do
    # Aqui é identificada qual tela deve ser mostrada.
    # Em cada tela são definidas as variáveis 'anterior'
    # e 'proxima' # que definem os rumos da navegação.
    case "$proxima" in
        primeira)
            proxima='nome'
            dialog --backtitle 'Pegador de Dados' \
                --msgbox 'Bem-vindo ao pegador de dados!' 0 0
            ;;
        nome)
            anterior='primeira'
            proxima='idade'
            nome=$(dialog --stdout \
                --backtitle 'Pegador de Dados' \
                --inputbox 'Seu nome:' 0 0)
            ;;
        idade)
            anterior='nome'
            proxima='casado'
            idade=$(dialog --stdout \
                --backtitle 'Pegador de Dados'   \
                --menu 'Qual a sua idade?' 0 0 0 \
                'menos de 15 anos'   '' \
                'entre 15 e 25 anos' '' \
                'entre 25 e 40 anos' '' \
                'mais de 40 anos'    '' )
            ;;
        casado)
            anterior='idade'
            proxima='gostos'
            casado=$(dialog --stdout \
```

```
        --backtitle 'Pegador de Dados'    \
        --radiolist 'Estado civil:' 0 0 0 \
        'solteiro' 'livre leve solto' ON  \
        'noivo'    'quase amarrado'   OFF \
        'casado'   'já era'           OFF \
        'viúvo'    'livre de novo'    OFF )
    ;;
gostos)
    anterior='casado'
    proxima='final'
    gostos=$(dialog --stdout \
        --separate-output                     \
        --backtitle 'Pegador de Dados'        \
        --checklist 'Do que você gosta?' 0 0 0 \
        'jogar futebol'      '' off \
        'pescar'             '' off \
        'ir ao shopping'     '' off \
        'andar de bicicleta' '' off \
        'ficar na internet'  '' off \
        'dormir'             '' off )
    ;;
final)
    dialog \
        --cr-wrap \
        --sleep 5 \
        --backtitle 'Pegador de Dados'   \
        --title 'Obrigado por responder' \
        --infobox "
        Os dados informados foram
        Nome  : $nome
        Idade : $idade
        Casado: $casado
        Gostos: \n$gostos
        " 14 40
    break
    ;;
*)
    echo "Janela desconhecida '$proxima'."
    echo "Abortando programa..."
```

```
            exit
    esac
    # Aqui é feito o tratamento genérico de Código de Retorno
    # de todas as telas. Volta para a tela anterior se for
    # CANCELAR, sai do programa se for ESC.
    retorno=$?
    [ $retorno -eq 1   ] && proxima="$anterior"        # cancelar
    [ $retorno -eq 255 ] && break                        # Esc
done
```

Configuração das cores das caixas

É possível configurar as cores de todos os componentes das caixas, como textos, bordas, botões e fundo da tela. Dessa maneira você pode personalizar seu programa, dando-lhe uma aparência diferenciada, fazendo com que sua aparência integre-se às cores da empresa ou do seu cliente.

Primeiro é preciso gerar um arquivo de configuração padrão, para então editá-lo. O próprio dialog possui uma opção que gera esse arquivo. A localização esperada deste arquivo é dentro de seu HOME, com o nome `.dialogrc`. Acompanhe:

```
dialog --create-rc $HOME/.dialogrc
```

Com este comando, o arquivo de configuração do dialog será criado no seu HOME. Agora basta editá-lo e executar o dialog após cada mudança para ver a diferença. Cada linha tem um comentário explicativo (em inglês) sobre qual o componente que leva a cor indicada. Mas se não entender não se preocupe, vá na tentativa e erro que é fácil.

O formato das configurações é:

```
nome_do_componente = (letra, fundo, letra brilhante?)
```

As cores para os parâmetros `letra` e `fundo` são:

BLACK	Preto
RED	Vermelho
GREEN	Verde
YELLOW	Amarelo

BLUE	Azul
MAGENTA	Rosa
CYAN	Ciano
WHITE	Branco

Os valores para o parâmetro `letra brilhante` são:

ON	Letra brilhante
OFF	Letra escura

Exemplos:

```
(GREEN, BLACK, OFF) = fundo preto, letra verde escuro
(GREEN, BLACK, ON)  = fundo preto, letra verde claro
```

Depois de terminar de configurar as cores, você pode salvar tudo em um arquivo separado e fazer vários arquivos diferentes para vários temas de cores. Para instruir o dialog a utilizar um arquivo de configuração que não o padrão `$HOME/.dialogrc`, basta definir a variável de ambiente `$DIALOGRC` com o seu caminho completo, por exemplo:

```
export DIALOGRC=$HOME/dialog/tema-verde.cfg
```

Uma dica legal é deixar a borda da mesma cor do fundo da janela, fazendo-a sumir. Para completar o visual "das antigas", escolha um par de cores e o aplique para todos os componentes, compondo um visual unificado. Exemplos clássicos são:

- **Fundo azul, letra branca:** usado no instalador do MS-DOS (lembra dos vários disquetes?) e em vários de seus programas.
- **Fundo preto, letra verde:** e viva os monitores trambolhosos monocromáticos da época do epa! Aquele que você usava no seu 286 para jogar Double Dragon e Battle Chess (aquele xadrez onde as peças eram bonecos que lutavam e se esquartejavam).

Dialog com o tema verde

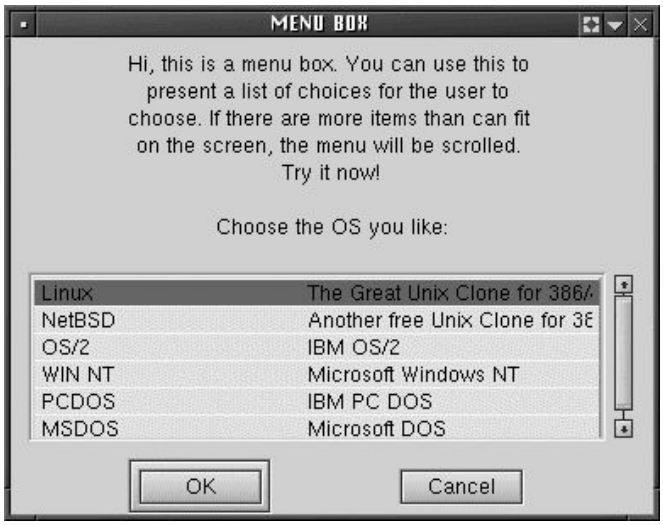
tema-verde.cfg

```
# Tema Verde, tipo monitor monocromatico, para o Dialog.
# http://aurelio.net/shell/dialog
#
#   Salvar este arquivo como $HOME/.dialogrc
#   ou definir a variavel $DIALOGRC
# screen
use_shadow   = OFF
use_colors   = ON
screen_color = (GREEN,BLACK,ON)
# box
dialog_color = (BLACK,GREEN,OFF)
title_color  = (BLACK,GREEN,OFF)
border_color = (BLACK,GREEN,OFF)
# button
button_active_color         = (BLACK,GREEN,OFF)
button_inactive_color       = (BLACK,GREEN,OFF)
button_key_active_color     = (GREEN,BLACK,OFF)
button_key_inactive_color   = (BLACK,GREEN,OFF)
button_label_active_color   = (GREEN,BLACK,OFF)
button_label_inactive_color = (BLACK,GREEN,OFF)
# input
inputbox_color        = (GREEN,BLACK,ON)
inputbox_border_color = (GREEN,BLACK,ON)
# textbox
```

```
searchbox_color          = (GREEN,BLACK,ON)
searchbox_title_color    = (GREEN,BLACK,OFF)
searchbox_border_color   = (GREEN,BLACK,OFF)
position_indicator_color = (BLACK,GREEN,OFF)
# Menu box
menubox_color          = (GREEN,BLACK,OFF)
menubox_border_color   = (GREEN,BLACK,OFF)
# Menu window
item_color             = (GREEN,BLACK,OFF)
item_selected_color    = (BLACK,GREEN,OFF)
tag_color              = (GREEN,BLACK,OFF)
tag_selected_color     = (BLACK,GREEN,OFF)
tag_key_color          = (GREEN,BLACK,OFF)
tag_key_selected_color = (BLACK,GREEN,OFF)
check_color            = (GREEN,BLACK,OFF)
check_selected_color   = (BLACK,GREEN,OFF)
uarrow_color           = (GREEN,BLACK,ON)
darrow_color           = (GREEN,BLACK,ON)
# Menu item help
itemhelp_color         = (GREEN,BLACK,ON)
```

Dialog na interface gráfica (X11)

Se o seu programa não estiver confinado a um servidor modo texto e puder desfrutar de todos os benefícios da interface gráfica, você pode rodar seu dialog dentro de um Xterm normal ou usar direto o Xdialog, que é o dialog gráfico.



Janela de menu no Xdialog

A melhor parte da brincadeira é que o `Xdialog` possui exatamente a mesma sintaxe na linha de comando do que o dialog, ou seja, nenhuma adaptação é necessária para que scripts que já utilizem o dialog rodem na interface gráfica com o Xdialog. Além de fazer tudo o que o dialog já faz, ele ainda traz vários tipos novos de caixa como:

Caixa	Descrição
buildlist	Adiciona e remove itens de uma lista.
colorsel	Escolhe uma cor.
combobox	Mostra um menu onde o usuário também pode digitar algo.
editbox	Edita um texto (é uma Textbox editável).
fontsel	Escolhe uma fonte (nome, tipo, tamanho).
logbox	Similar à Tailbox, mas mostra linhas em destaque (usando cores).
rangebox	Muda um valor numérico, como o volume de uma música.
spin boxes	Similar ao rangebox, aceita até três componentes em uma janela.
treeview	Mostra dados hierárquicos em formato de árvore.

Para fazer o mesmo programa utilizar o Xdialog se estiver na interface gráfica e o dialog caso não esteja, é simples. Primeiramente, troque todas as chamadas ao dialog no programa, por `$DIALOG`, ficando assim:

```
$DIALOG --yesno "X é legal?" 0 0
```

Essa variável nova guardará qual o dialog que vai ser usado, o console ou o gráfico. Para saber se o usuário está na interface gráfica ou não, basta

checar a existência da variável `$DISPLAY`, que só é definida quando o X11 está sendo executado. Então, bem no início do programa, coloque o seguinte teste:

```
if [ "$DISPLAY" ]; then
    DIALOG=xdialog
else
    DIALOG=dialog
fi
```

Mas o Xdialog não é o único clone do dialog. Se você gosta de diversidade, experimente todos os sabores:

Variante	Sistema	Endereço
Xdialog	X11	http://xdialog.dyns.net
kdialog	KDE	http://developer.kde.org/documentation/tutorials/kdialog/t1.html
Zenity	Gnome	http://www.linuxmanpages.com/man1/zenity.1.php
whiptail	Console	http://www.wlug.org.nz/whiptail(1)
Python Dialog	Python	http://pythondialog.sourceforge.net
UDPM	Perl	http://search.cpan.org/~kck/UDPM/UDPM.pm

Capítulo 12

Programação Web (CGI)

Apesar de todas as melhorias já feitas em seu programa, como opções de linha de comando e interface amigável, ele continua restrito ao terminal. Que tal libertar de vez seu programa, deixando-o disponível para que qualquer pessoa do mundo possa usá-lo por meio da Internet? Aprenda a fazer seu programa funcionar como um CGI, usando a Internet como meio de comunicação e o navegador como interface com o usuário.

O CGI (Common Gateway Interface) é uma maneira simples de se rodar programas pela Internet (e Intranet). O executável fica armazenado em um diretório especial do servidor e você pode dispará-lo com um simples apertar de botão em seu navegador, de qualquer parte do mundo. Parece tentador, não? E fica melhor ainda quando você descobre que esse programa pode ser um shell!

Prepare-se para ampliar seus horizontes e sair da "casca". Você continuará codificando em shell, mas os seus programas poderão ganhar o mundo. Novas possibilidades se abrem, e se você já teve uma avalanche de ideias quando conheceu o dialog, com CGI isso se multiplica, pois uma vez dominado o conceito, não há limites para programar seguindo sua imaginação.

O funcionamento básico do CGI é o seguinte:

- Pelo navegador (browser), o usuário acessa o endereço do seu programa. Algo como http://seuservidor.com.br/programa.
- Ele preenche alguns dados em um formulário e aperta o botão Enviar.
- Seu programa é chamado, faz o processamento e retorna uma outra página para o usuário, contendo o resultado.
- Essa outra página pode ter mais campos e botões para que outros programas sejam executados, ou pode ser o resultado final que o usuário esperava.

Bem diferente do que estamos acostumados, não é mesmo? Este é nosso último assunto do livro, é a fronteira final do shell. Juntando todo o conhecimento adquirido até aqui, você está preparado agora para uma mudança radical de pensamento, pois no ambiente CGI:

- Não há entrada padrão (`STDIN`).
- Não há opções de linha de comando.
- Não há linha de comando!
- A saída padrão (`STDOUT`) é enviada diretamente ao navegador.
- Não há interatividade, cada chamada ao programa deve chegar até o fim e mostrar algo na saída.

Em compensação, há uma troca. Ao mesmo tempo que perdemos estas características poderosas que estamos tão acostumados, ganhamos novas maneiras de programar, que substituem (e até superam) alguns destes conceitos:

- Os dados são passados ao programa por meio de formulários preenchidos pelo usuário. (Elimina entrada padrão e opções).
- O acesso ao programa é transparente para o usuário. Ele vê apenas o resultado. (Elimina linha de comando).
- A saída do programa deixa de ser limitada pelo console e sua fonte fixa. Em vez disso, seu CGI construirá uma atraente página de Internet, com alinhamento, imagens, tabelas e todos os recursos de um site elegante.
- Acabam as preocupações com permissões de usuário, PATH, pré-requisitos, configurações do sistema e outros problemas importantes que ocorrem quando o programa precisa rodar em um ambiente desconhecido. No CGI há uma única cópia do código, que roda em um servidor com ambiente configurado e conhecido pelo programador.
- Por rodar pela Internet, seu programa atinge um novo público. Não importa se o usuário está em um Linux, Windows, Mac ou outro sistema. Se o navegador funcionar, seu programa funciona também.
- Formulário em páginas de Internet já é uma interface conhecida pelo grande público. Não há curva de aprendizado.
- Não há instalação! O usuário apenas usa. Simples assim.

Da linha de comando para o navegador. Do shell para a Internet. Do público restrito para o mundo. É este o avanço que a programação em CGI lhe proporciona.

"Uau! Me dá três desse!"

Vantagens e desvantagens do CGI em Shell

O CGI não é um conceito exclusivo do shell, qualquer linguagem de

programação pode ser utilizada. Perl já foi muito utilizada para fazer CGIs no final dos anos 90. Depois vieram o PHP e o ASP, executando códigos diretamente na página HTML, sem precisar de um programa separado como no CGI. Atualmente, a nova tecnologia é o AJAX, tornando as páginas ainda mais interativas ao possibilitar o carregamento de informações em segundo plano, sem a necessidade de recarregar a página inteira.

Mas uma tecnologia não exclui a outra. Há situações em que é melhor usar um CGI e em outras o AJAX é imbatível. Uma regra básica que você pode guardar é que se o processamento puder ser feito em batch (obtém todas as informações necessárias e executa de uma vez, sem interação), o CGI é uma boa escolha. E ao escolher o shell como linguagem, temos alguns pontos a se observar:

Vantagens

- Várias das vantagens que já conhecemos de se programar em shell, aplicam-se ao CGI: integração com o sistema, facilidade e rapidez de desenvolvimento, redirecionamento e encadeamento de comandos, testes rápidos pela linha de comando.
- Programas pequenos e médios têm execução instantânea, mas a performance de um programa grande em shell pode deixar a desejar. A notícia boa é que uma eventual demora durante a execução não será impactante. Isso porque, ao rodar pela Internet, o programa já vai sofrer com o tempo de resposta da conexão em si. O usuário que navega já está acostumado a esperar um pouco até que a página carregue. Se o seu programa demorar um pouco, esse tempo de espera será percebido como conexão lenta, não se preocupe :)
- Você já programa em shell. Sabe como fazer arquivos de configuração, banco de dados e código limpo. Você já possui o conhecimento necessário para fazer grandes programas.
- Os requisitos de sistema para rodar seu programa são mínimos. Não é preciso pagar mais caro por hospedagem em um servidor com suporte a Python, PHP e seus diversos frameworks.
- Configuração mínima. Basta configurar o servidor HTTP (Apache)

para aceitar CGIs e pronto. Nada precisa ser instalado.

- Você pode transformar em CGI qualquer programa em shell (não interativo) que já tenha feito. Com algumas mudanças no código seu programa pode funcionar pela Internet.
- Shell é sexy, você sabe.

Desvantagens

- A obtenção dos dados preenchidos no formulário é trabalhosa, eles não veem prontos como no PHP. Mas uma vez feita a função que os extrai, não precisa mexer mais.
- É possível usar modelos (templates) para as páginas HTML, mas deve-se tomar o cuidado de evitar construções que possam ser interpretadas pelo shell, como `$variaveis` e `` `subshell` ``.
- Tanto para obtenção dos dados quanto para o uso de modelos, a tentação em usar o `eval` é grande. E, ao usá-lo, várias verificações de segurança precisarão ser feitas, pois o usuário poderá enviar textos com códigos embutidos. Outro problema será a legibilidade do código, que ficará prejudicada.

Não há grandes desvantagens no uso do shell para CGI. Só será mais trabalhoso desempenhar algumas tarefas, pois o shell não foi projetado para o uso na Internet. Mas isso não é um impeditivo, e uma vez feitas as funções auxiliares para contornar estas limitações, não precisa mais preocupar-se.

É divertido programar em shell. Será divertido programar em shell para CGI.

É recomendado que você esteja no computador para continuar a leitura. Invista um tempo para configurar sua máquina e testar os exemplos que serão ensinados. Há muitos detalhes que podem ser ignorados durante a leitura sem a prática. Você só sentirá falta deles quando precisar fazer um CGI. Então poupe seu tempo futuro, aprendendo agora do jeito certo. Você não vai se arrepender, é bom brincar de CGI!

Preparação do ambiente CGI

Cá entre nós, CGI funciona que é uma beleza. Mas até chegar no ponto

de você ver um simples "Hello World" aparecendo no navegador, há um caminho chatíssimo a percorrer, com configuração do servidor e aprendizado de detalhes básicos de como os CGIs funcionam. Então não estranhe se você bocejar nos parágrafos seguintes... Mas não desista que vale a pena!

O único requisito para que o esquema de CGI funcione em uma máquina é que você tenha um servidor HTTP instalado nela. É este servidor quem entende o protocolo CGI e vai servir de ponte entre seus programas e seu navegador. Dentre os vários servidores disponíveis para uso, o Apache (www.apache.org) é o mais popular e está presente no Linux e Mac, mas também pode ser instalado no Windows. Então ele será nosso escolhido.

A instalação e configuração do Apache foge do escopo deste livro, então sua primeira missão é fazê-lo funcionar. Mas não se assuste pois não é difícil. No Linux e no Mac ele já vem instalado de fábrica, basta ativá-lo. No Windows é preciso baixar e instalar o Apache, porém este procedimento é rápido e fácil.

Para testar o funcionamento do Apache, abra seu navegador preferido e tente acessar algum dos seguintes endereços:

- http://localhost
- http://127.0.0.1

Se você estiver em uma máquina sem interface gráfica, use o navegador de modo texto lynx.

Se aparecer uma página do Apache, parabéns, está tudo funcionando. Se der algum erro, leia a mensagem e procure por soluções na documentação ou na Internet. Confira os passos de instalação, tente novamente, não desista.

Servidor Apache instalado e funcionando!

Configuração do Apache

Agora é hora de configurar o Apache para aceitar nossos futuros CGIs em shell. Use algum configurador gráfico ou edite o arquivo de configuração na unha, que geralmente é o `/etc/httpd/httpd.conf` no Linux/Mac e `C:\Program Files\Apache Group\Apache\conf\httpd.conf` no Windows. Estas são as configurações que você deve verificar:

- O módulo de CGI deve estar ativo (não comentado). Procure por `cgi_module`:

```
LoadModule cgi_module /caminho/para/mod_cgi.so
```

- As extensões de arquivos `.cgi` e `.sh` devem ser reconhecidas como CGIs pelo Apache. Então esteja avisado desde já que os seus programas devem ter uma dessas duas extensões, senão não vai funcionar. Procure por `cgi-script`:

```
AddHandler cgi-script .cgi .sh
```

- Para nossos testes, é importante saber qual usuário do sistema que o Apache utiliza quando está em execução. Geralmente é `nobody` ou `www`. Procure por uma linha que inicia com `User`:

```
    User www
```

Agora você precisa escolher um diretório do seu sistema para colocar todos os CGIs e páginas HTML que faremos a seguir. De preferência crie um diretório novo, para evitar confusão com arquivos já existentes. Um detalhe importante é que este diretório deve ser acessível pelo usuário do Apache (o `nobody` ou `www`, que você acabou de verificar). Então ele não pode estar dentro de seu diretório pessoal, que é protegido (`$HOME`). Vou usar o `/tmp/shell` como exemplo:

```
# Configuracoes para o estudo de shell e CGI
Alias /shell "/tmp/shell/"
<Directory "/tmp/shell/">
    Options Indexes FollowSymlinks ExecCGI
    AllowOverride None
    Order allow,deny
    Allow from all
</Directory>
```

Coloque estas linhas no final do `httpd.conf`, depois de todas as configurações já existentes. Com isso estamos criando uma área nova chamada `shell`, que apontará para o diretório escolhido para abrigar os CGIs. Troque `/tmp/shell` pelo seu diretório de estudo.

Feitas as alterações é preciso reiniciar o Apache para que as novas configurações sejam utilizadas. É importante acessar novamente o endereço http://localhost em seu navegador para ter certeza de que tudo continua funcionando. Vá que você mexeu onde não devia e quebrou o Apache? Ah, eu devia ter avisado antes, mas é bom fazer uma cópia do arquivo de configuração original antes de você começar a mexer nele. Sabe como é, cacas acontecem.

Agora vamos testar se o Apache entendeu a configuração para ler o diretório que criamos para os estudos. Vá no navegador e tente acessar a área `shell`, usando o endereço http://localhost/shell/.

Apache mostrando a listagem de diretório

Apache configurado e funcionando, sua primeira missão está completa. E com isso terminamos a primeira parte dos preparativos. Até que não foi tão chato, né? O bom é que isso só é preciso fazer uma vez. Agora estamos prontos para começar.

O primeiro CGI

Ah, o primeiro CGI... É uma emoção que fica guardada na memória. Saboreie os próximos passos com calma e satisfação. No final você será recompensado com a alegria de ver o seu programa funcionando dentro do navegador!

O primeiro programa que faremos será bem simples. O que importa agora não é a lógica ou os algoritmos, mas, sim, testar o ambiente CGI e aprender sobre as suas pegadinhas. Chame seu editor de textos preferido e crie o promissor `oi.sh`:

oi.sh

```
#!/bin/bash
# oi.sh
echo Content-type: text/plain
echo
echo Oi
```

Este programa deve ser salvo dentro do diretório dos CGIs, que você acabou de configurar no Apache e acessar no navegador. Todos os exemplos deste capítulo

devem ser guardados neste diretório, sem exceção.

...deve ser um executável

É importante que todo CGI que você criar seja um arquivo executável. No momento em que o Apache for chamado pelo navegador, ele vai tentar executar seu programa. Lembre-se de que CGI não é uma exclusividade do shell e que o Apache não tem como adivinhar em qual linguagem seu programa está escrito. Por isso, é obrigatória a primeira linha mágica, indicando que este é um programa em shell e que é o Bash quem sabe executá-lo. Outro requisito é dar a permissão de execução do programa para **todos** os usuários:

```
$ chmod +x oi.sh
```

Note a ênfase na palavra todos. Não basta que o programa seja executável somente por você, pois o Apache possui seu próprio usuário (que já vimos na configuração) e é este usuário quem irá rodá-lo. Por isto, guardar CGIs dentro do seu diretório pessoal (`$HOME`) não é uma boa ideia, pois somente seu usuário tem acesso a ele.

Então sempre, sempre que criar um CGI, a primeira ação é torná-lo executável. A segunda é ver se ele funciona normalmente na linha de comando. Afinal, um CGI é um programa como qualquer outro, que deve funcionar sem erros quando chamado:

```
$ ./oi.sh
Content-type: text/plain
Oi
$
```

Já veremos o porquê dessa primeira linha com o `Content-type`. O importante agora é que temos um programa executável e ele não contém nenhum erro de sintaxe, está funcionando redondo. Este teste é muito importante que se faça agora, antes de tentar acessar o CGI no navegador, pois na linha de comando é mais fácil resolver os problemas do programa.

...deve ser executável pelo Apache

Isolar problemas é uma arte que somente após muitas noites mal dormidas você aprende a valorizar e praticar. Então, antes de irmos ao

navegador, é prudente fazer mais um teste. Se é o usuário do Apache quem vai executar nosso programa, que tal verificar se ele realmente consegue fazer isso? Ele irá conseguir entrar no diretório onde está seu CGI? Será que o programa depende de alguma variável que este usuário não tem configurada?

```
(joao)$ su -                # vire root
Password:
(root)# su - www            # vire usuário do Apache
(www )$ cd /tmp/shell     # entre no diretório dos CGIs
(www )$ ./oi.sh             # teste o programa
Content-type: text/plain
Oi
(www )$
```

Muito bem, tudo parece estar funcionando como deveria. Temos um executável dentro do diretório de CGIs que o usuário do Apache consegue rodar. Caso qualquer problema apareça em algum destes passos, corrija.

No procedimento anterior, primeiro é preciso virar root para depois virar usuário www. Essa volta é necessária porque este usuário do Apache não possui senha nem shell por motivos de segurança. Somente o root pode virar `www` (ou `nobody`, ou qualquer que seja o usuário configurado no seu Apache).

...deve informar o tipo do conteúdo (Content-type)

Sim, o `Content-type`. Você deve estar curioso. Essa primeira linha que o programa mostra, informando um tipo `text/plain`, está dizendo ao Apache que o conteúdo seguinte (após a linha em branco) é um texto simples. Poderia ser código HTML, poderia ser uma imagem ou até mesmo um arquivo compactado. Veja alguns exemplos de tipos possíveis:

Principais tipos de Content-type

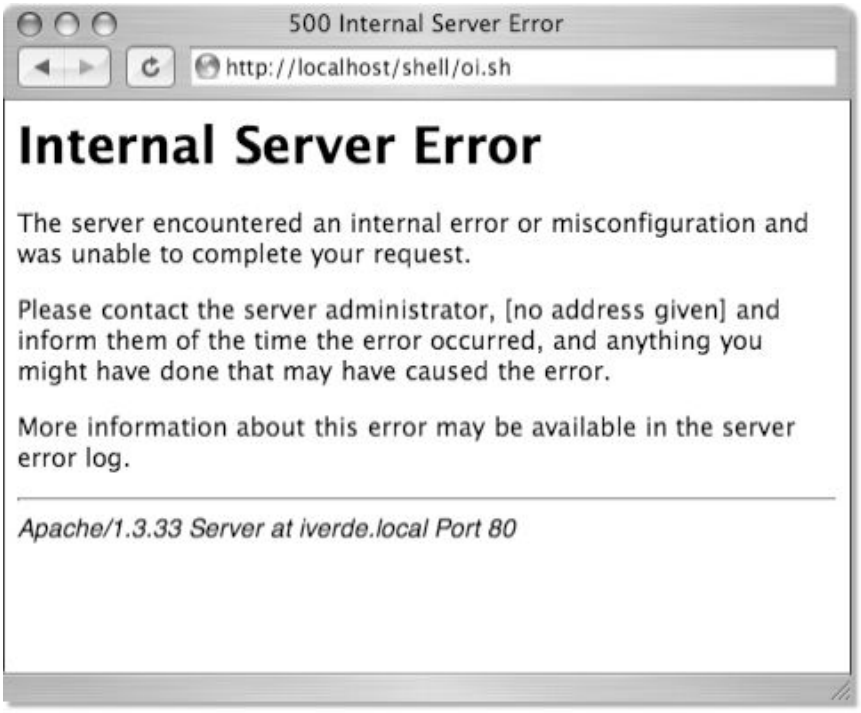
Content-type	Descrição
text/plain	Texto
text/html	Código HTML
text/css	Folha de Estilo (CSS)
text/xml	Documento XML
image/gif image/jpeg	Imagens nos formatos GIF, JPG e PNG

image/png	
application/javascript	Código Javascript
application/pdf	Documento PDF
application/zip	Arquivo compactado com ZIP
audio/mpeg	Áudio no formato MPEG

A (imensa) lista oficial de todos os tipos possíveis está no site da IANA, em www.iana.org/assignments/media-types. Mas em nossos estudos tudo o que você vai precisar é do `text/plain` para mostrar texto e do `text/html` para mostrar uma página HTML.

É muito importante frisar: essas duas primeiras linhas são **obrigatórias**. A primeira informando o tipo e a segunda em branco. Não podem haver linhas em branco ou espaços antes do `Content-type`. Todo CGI deve mandar estas duas linhas antes de qualquer outra saída. Se você se esquecer disso e quebrar esta regra, será presenteado com o famoso Internal Server Error.

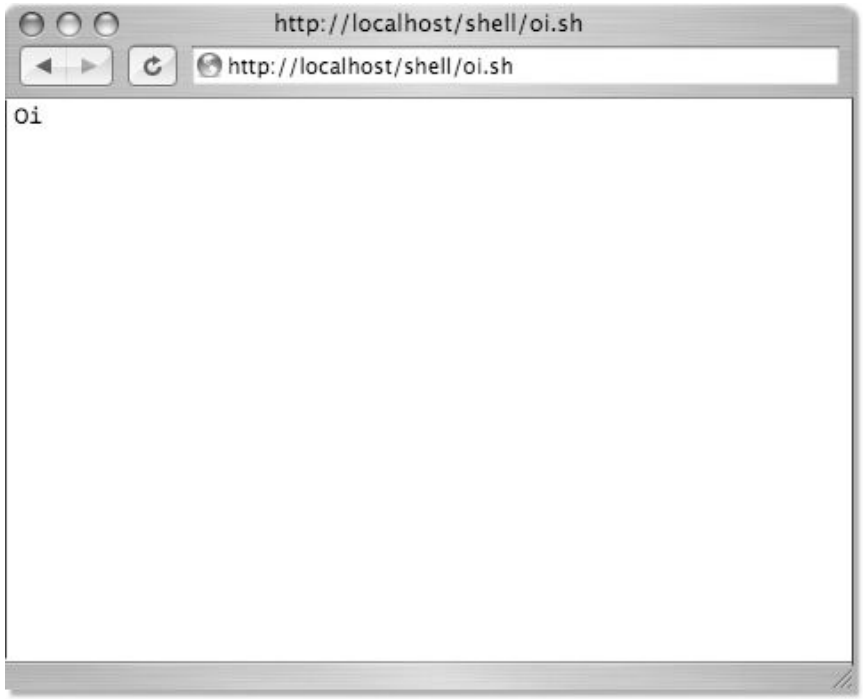
Este é um mandamento, uma regra sagrada, um tabu que não pode ser quebrado. Todo CGI deve mandar para a saída padrão (STDOUT) uma linha com o `Content-type` seguido de uma linha em branco. A saída normal de seu programa deve vir logo em seguida. Em outras palavras, o resultado deve sempre iniciar com `Content-type:...\n\n`.



Erro de falta do Content-type

Testando no navegador

Se todas verificações foram satisfeitas, então agora você pode fazer o tão esperado teste final: rodar seu programa em shell direto no navegador. Prepare a câmera, chame a família e acesse o endereço http://localhost/shell/oi.sh. Rufam os tambores, momento de tensão, dedos cruzados.



Funcionamento do primeiro CGI

Caso o texto "Oi" não tenha aparecido, procure por mensagens de erro relevantes nos logs do Apache, que geralmente são guardados em `/var/log/httpd/error_log` no Linux/Mac e `C:\Program Files\Apache Group\Apache\logs\error`.log no Windows.

E não é que funciona?

O que aconteceu aqui é que o seu programa em shell mandou uma mensagem ao navegador. O Apache foi o intermediário, o canal de comunicação que conectou ambos. Este é apenas o ponto de partida, o funcionamento básico. Estabelecida esta comunicação, agora você pode liberar a criatividade e dar passos mais largos:

- Em vez de uma simples mensagem "Oi", você pode mostrar códigos em HTML e montar um site com páginas dinâmicas, construídas em tempo real por seu programa. Por exemplo: `echo "<P><B>Data atual: </B> $(date)</P>"`
- Você está acessando um endereço local de sua própria máquina. Mas, colocando esse programa em um servidor conectado à Internet, qualquer pessoa do mundo pode utilizá-lo! Então o acesso fica global. O endereço ficará algo como http://www.seuservidor.com.br/~joao/oi.sh
- Como dizem: O céu é o limite! Você já sabe programar em Shell. Você já sabe como colocar este programa rodando na Internet. Agora basta ter uma ideia boa, programar e colocar na rede. Se vai fazer uma calculadora, um sistema de vendas ou um novo Google, é com você. A dica mais importante é: **faça!**

Recapitulando, estes são os requisitos para que seu CGI funcione:

- O Apache deve estar configurado para aceitar a execução de CGIs.
- O CGI deve estar no diretório específico, configurado no Apache.
- O CGI deve ser executável na linha de comando pelo usuário `www` (ou similar).
- O CGI deve mandar o resultado para a saída padrão (STDOUT), sendo a primeira linha o `Content-type` e a segunda em branco.

CGI gerando uma página HTML

Foi bom ver o "Oi" no navegador, não foi? Mas já que não estamos mais limitados pela saída somente texto do console, agora podemos usufruir de todo o poder de um navegador, que entre outras funções, mostra páginas formatadas que usam títulos, negrito, listas, imagens e tabelas, entre outros.

Introdução ao HTML

Esta formatação é definida por uma linguagem chamada de HTML, que, vista de uma maneira bem simplista, apenas adiciona etiquetas (tags) às palavras, dando-lhes formatos novos. Por exemplo, na frase "Tinha que

ser o <U>Chaves</U> mesmo!", a palavra Chaves está marcada com a etiqueta chamada "U", que significa sublinhado. Então quando mostrada na tela, essa frase ficará: Tinha que ser o Chaves mesmo!

Há dezenas de tags disponíveis no HTML, e seu detalhamento está fora do escopo deste livro. Mas não é preciso dominar o assunto, pois as tags necessárias para compor páginas simples e apresentáveis, são poucas:

Etiquetas (tags) HTML mais utilizadas

Tag	Descrição	Exemplo
H1	Título	<H1>Capítulo 1</H1>
H2	Subtítulo	<H2>Capítulo 1.1</H2>
P	Parágrafo	<P>Shell é bom.</P>
B	Negrito	<B>destaque</B>
I	Itálico	<I>ênfase</I>
U	Sublinhado	<U>grifado</U>
FONT	Fonte (tipografia)	<FONT COLOR="green">texto em verde</FONT>
BR	Quebra de linha (\n)	Esse texto aparecerá em três linhas.
PRE	Códigos (fonte fixa, mantém quebras de linha)	<PRE> if test -f /tmp/foo; then echo existe fi </PRE>
HR	Linha separadora	<HR>
IMG	Imagem, foto	<IMG SRC="foto.jpg">
A	Link	<A HREF="http://google.com">Página do Google</A>
UL, LI	Lista de itens	<UL> <LI>primeiro</LI> <LI>segundo</LI> </UL>
TABLE, TR, TD	Tabela	<TABLE BORDER="1"> <TR> <TD>Linha 1, coluna 1</TD> <TD>Linha 1, coluna 2</TD> </TR> <TR> <TD>Linha 2, coluna 1</TD> <TD>Linha 2, coluna 2</TD> </TR> </TABLE>

Algumas dicas gerais para escrever código HTML:

- Um arquivo HTML é um arquivo de texto normal, porém com tags marcando as palavras.
- As tags sempre são colocadas entre os sinais de menor e maior, <assim>.
- Algumas tags são sozinhas, outras são em pares (abre e fecha). Por exemplo, `<HR>` é sozinha e `<B> ... </B>` é um par. Note que o segundo item do par sempre tem uma barra / para indicar que está fechando a tag.
- Algumas tags aceitam argumentos opcionais, modificando seu comportamento original. Por exemplo:

```
<TABLE BORDER="1" CELLPADDING="5"> ... </TABLE>
```

- Tanto faz colocar o nome da tag em maiúsculas ou minúsculas. Então `<hr>` é o mesmo que `<HR>`.
- Vários espaços consecutivos e linhas em branco aparecem como um único espaço em branco no navegador (exceto se usar a tag `PRE`).
- Pode-se colocar tags dentro de tags:

```
<P>Um <B>negrito</B> e um <I>itálico</I>.</P>
```

- Pode-se acumular tags ao redor de um trecho, porém lembre-se de fechar as tags em ordem inversa, para não se intercalarem:

```
<P>Agora um <B><I><U>negrito-itálico-sublinhado</U></I></B>.
</P>
```

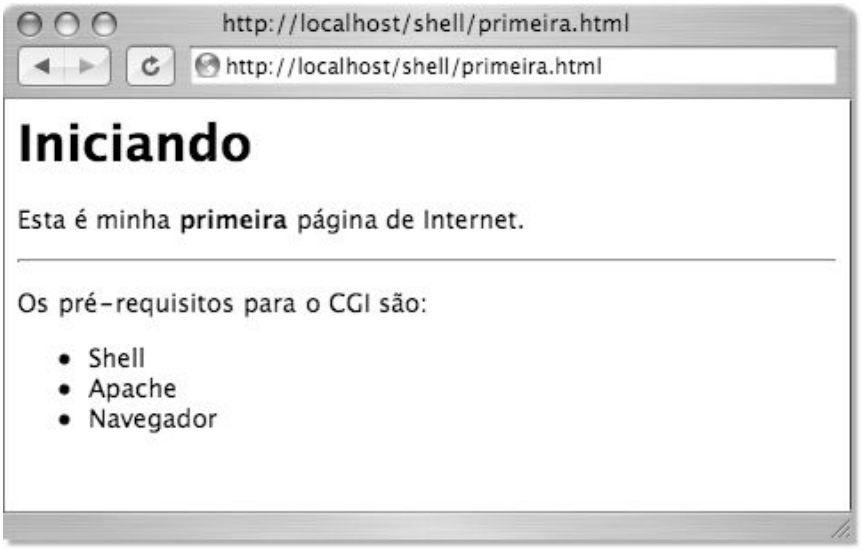
- Não tenha medo de experimentar!

É fácil escrever código HTML. Afinal, você já sabe shell que é muito mais difícil! Vamos praticar? Volte ao editor e digite o seguinte texto, marcado com tags:

```
<H1>Iniciando</H1>
<P>Esta é minha <B>primeira</B> página de Internet.</P>
<HR>
<P>Os pré-requisitos para o CGI são:</P>
<UL>
```

```
    <LI>Shell</LI>
    <LI>Apache</LI>
    <LI>Navegador</LI>
</UL>
```

Salve este arquivo dando o nome de `primeira.html` (em nosso diretório de estudo) e abra-o no navegador. Note que é importante a extensão ser `.html`, para que o navegador saiba que este é um arquivo marcado, e não um texto comum. Veja como o texto fica formatado conforme as tags que utilizamos:



Funcionamento da primeira página HTML

Agora aproveite para brincar de HTML. Altere o texto do arquivo, use outras tags, experimente combinações diferentes. É importante você investir um tempo nesse primeiro contato para ter a visão geral e saber que tipo de formatação você pode utilizar em uma página. Quando estiver programando seu CGI, você vai precisar desse conhecimento. Então, agora é a hora do recreio!

Mensagem aos puristas: Você deve ter ficado horrorizado quando viu a menção à tag `FONT`. É sabido que o CSS é mais indicado para este tipo de formatação. Mas como este é um livro de Shell, não há como aprofundar o estudo de HTML. Então em prol da didática, usar `FONT` é mais indicado. Repita esta mensagem para qualquer outra tag que lhe tenha causado espanto :)

Agora o CGI entra na brincadeira

Já desfigurou todo o seu `primeira.html`? Experimentou todas as tags, descobriu novidades? Sempre volte a este arquivo quando quiser testar algum formato novo, ou não lembrar direito como se usa determinada tag.

Vamos voltar à programação em shell. Nosso primeiro CGI simplesmente mostrava uma palavra na tela. Sem formatação nenhuma, era texto puro. O próximo programa será mais sofisticado e irá gerar um texto formatado em HTML, que será interpretado pelo navegador e mostrado ao usuário.

Se vamos mudar o formato da saída do CGI de texto para HTML, precisamos antes de tudo alterar a primeira linha obrigatória, a do `Content-type`, para refletir essa mudança. Se antes usávamos `text/plain`, agora o conteúdo passará a ser identificado por `text/html`. Em seguida a linha em branco e só então um texto marcado:

teste_html.sh

```
#!/bin/bash
# teste_html.sh
echo Content-type: text/html
echo
echo "<h1>Testando... 1, 2, 3</h1>"
```

Vamos testar? Salve este arquivo no diretório de estudo e lembre-se que você deve torná-lo executável (`chmod +x`). Feito isso, podemos chamá-lo na linha de comando:

```
$ ./teste_html.sh
Content-type: text/html
<h1>Testando... 1, 2, 3</h1>
$
```

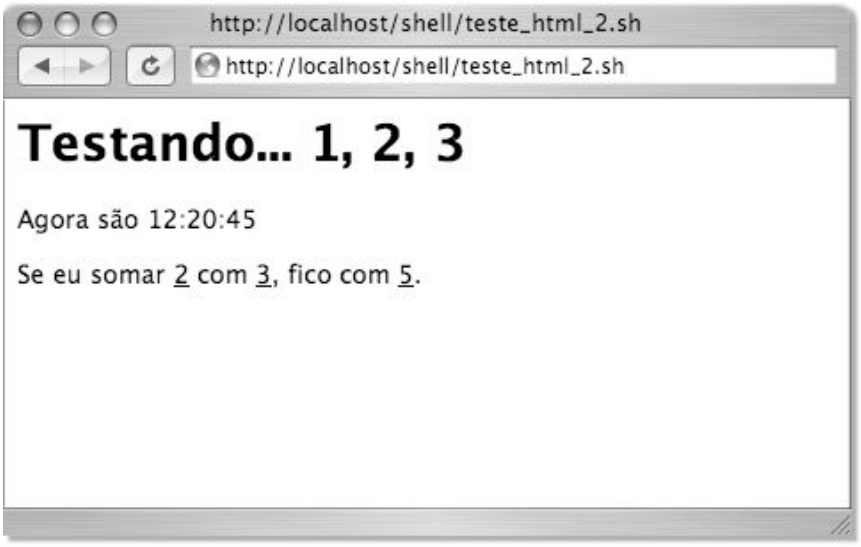
Tudo bem, funciona. No console tudo é texto, sem leitura de tags nem formatação. Mas isso muda quando executamos este mesmo programa pelo navegador, que reconhece o tipo HTML e sabe como interpretar e formatar suas tags:

CGI gerando uma página HTML

Legal! A tag `H1` foi interpretada e o texto virou um título, com fonte grande e em negrito. A primeira linha do `Content-type` não é mostrada, pois ela é um cabeçalho informativo e não faz parte do conteúdo. Isso está ficando bom, hein? Mas já que estamos programando em shell, que tal colocar mais alguns comandos em nosso programa?

```
hora_atual=$(date +%X)
x=2
y=3
echo "<P>Agora são $hora_atual</P>"
echo "Se eu somar <U>$x</U> com <U>$y</U>, fico com <U>$((x+y))
  </U>."
```

Repare dentro do `echo`, como as tags do HTML misturam-se às variáveis e comandos do shell, que se misturam ao texto normal. É assim mesmo, essa salada de formatos e tecnologias, que no final resulta em uma página limpa, formatada e dinâmica, que obtém e calcula as informações no momento em que o endereço é acessado.



Resultado com informações processadas dinamicamente

Transforme um programa normal em CGI

Você já tem seu programa em shell. Ele já funciona e mostra no console as informações que o usuário quer ver. Que tal transformá-lo em um CGI? Assim, além da vantagem de funcionar pelo navegador, ainda é possível deixar mais atraente a sua saída, mudando de texto puro para HTML. Será que isso é muito difícil de fazer? Que nada!

Usaremos como exemplo um programa simples, chamado de `mostra_path.sh`. Conforme o nome indica, ele mostra qual o valor de sua variável `$PATH`. Veja um exemplo de sua execução:

```
$ ./mostra_path.sh
/bin (existe)
/sbin (existe)
/usr/bin (existe)
/usr/sbin (existe)
/usr/libexec (existe)
/System/Library/CoreServices (existe)
/noel (não existe)
$
```

Além de mostrar os diretórios de pesquisa de seu `PATH` um por linha, o programa também faz uma verificação se cada um destes diretórios realmente existe no sistema. Um diretório falso `/noel` é adicionado ao

PATH para ilustrar o que acontece com diretórios não existentes. A saída, apesar de informativa, parece um pouco bagunçada, com os dados misturados. Não é fácil enxergar rapidamente quais diretórios não existem. Seu código é bem simples:

mostra_path.sh

```
#!/bin/bash
# mostra_path.sh
# Adiciona um diretório falso ao PATH
PATH=$PATH:/noel
# Para cada diretório do $PATH...
IFS=:
for diretorio in $PATH; do
    # Confirme se ele existe
    if test -d $diretorio; then
        extra="existe"
    else
        extra="não existe"
    fi
    # E mostre o resultado na tela
    echo "$diretorio ($extra)"
done
```

Tranquilo, não? Mudando o IFS conseguimos fazer o loop pelos componentes do PATH, fazendo um teste pela existência do diretório a cada volta. Agora vamos transformar este programa em um CGI. O primeiro passo é copiá-lo para o nosso diretório de estudo. Feito isso, podemos editá-lo e colocar as duas primeiras linhas obrigatórias, que você já sabe quais são:

```
echo Content-type: text/html
echo
```

Sempre mostre o `Content-type` já no início do programa, antes mesmo de definir as funções e variáveis globais. Assim, qualquer erro que possa ocorrer durante a execução será mostrado no navegador, não passará despercebido.

Se fizermos esta única mudança e chamarmos o programa pelo navegador, veja o que acontece:

Linhas grudadas no navegador

Dissemos que o conteúdo era HTML, mas não colocamos nenhuma tag nele. Não há etiquetas para parágrafos, nem quebras de linha. Então o navegador junta tudo em uma única linha.

Sempre que o resultado de seu programa aparecer embaralhado em uma tripa interminável, é porque faltam as tags HTML para formatação! Na dúvida, use a tag `PRE` para colocar a casa em ordem.

Há uma tag muito prática para nós, programadores, que mantém toda a formatação original do texto: `PRE`. Ela indica que o texto está **pré**-formatado, ou seja, o navegador deve respeitar as quebras de linha e espaços em branco já existentes. Com ela, a mesma saída vista na linha de comando aparecerá no navegador, em fonte fixa (monoespaçada).

O uso da tag `PRE` é muito simples, basta abri-la no início do programa e fechá-la no final. Assim toda e qualquer saída do programa estará marcada como pré-formatada.

mostra_path_pre.sh

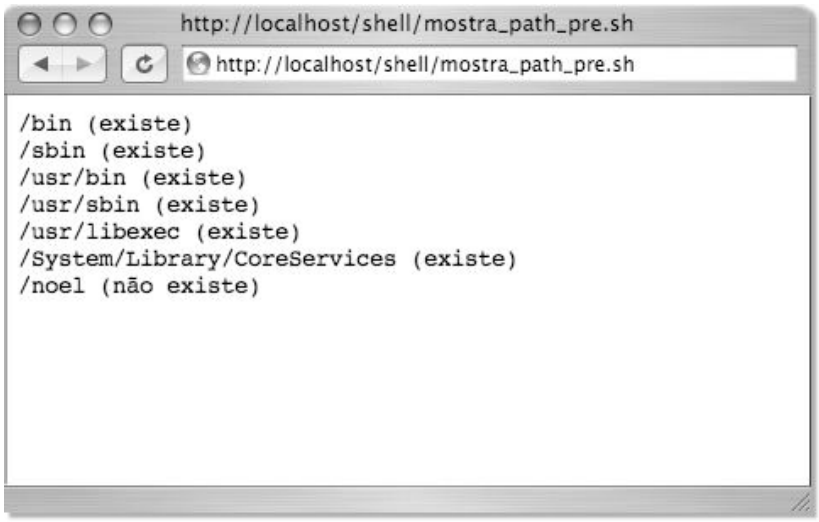
```
#!/bin/bash
# mostra_path_pre.sh

# Vamos mostrar uma página HTML
echo Content-type: text/html
echo
# Adiciona um diretório falso ao PATH
```

```
PATH=$PATH:/noel
# Inicia um texto pré-formatado
echo "<PRE>"
# Para cada diretório do $PATH...
IFS=:
for diretorio in $PATH; do

    # Confirme se ele existe
    if test -d $diretorio; then
        extra="existe"
    else
        extra="não existe"
    fi

    # E mostre o resultado na tela
    echo "$diretorio ($extra)"
done
# Fecha o texto pré-formatado
echo "</PRE>"
```



Saída igual à do console, em fonte fixa

Com poucas modificações adicionamos as tags e agora temos no navegador a mesma saída do console, com um diretório por linha. Mas lembre-se: **não estamos mais no console!** Isso significa que temos mais possibilidades de formatação, então podemos melhorar a saída desse programa. Temos toda a gama de tags do HTML disponíveis. Nos

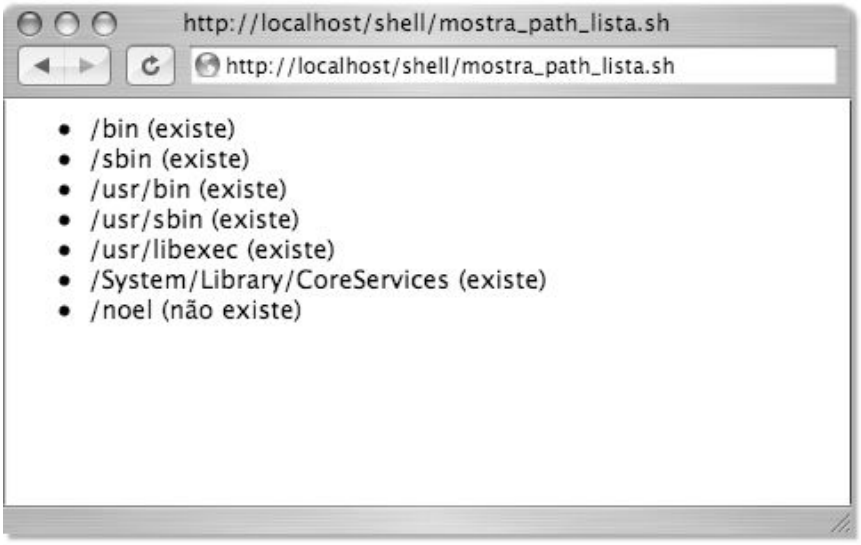
próximos passos vamos melhorar essa saída, tornando-a mais atraente e organizada.

Primeiro vamos analisar qual a estrutura dos dados que o programa mostra: uma lista de diretórios. Uma lista... Bem, no HTML há uma tag especial para mostrar listas de itens: `UL`. Que tal usá-la em nosso programa? Confira as mudanças, em negrito:

mostra_path_lista.sh

```
#!/bin/bash
# mostra_path_lista.sh
# Vamos mostrar uma página HTML
echo Content-type: text/html
echo
# Adiciona um diretório falso ao PATH
PATH=$PATH:/noel
# Inicia uma lista de itens
echo "<UL>"
# Para cada pasta do $PATH...
IFS=:
for pasta in $PATH; do

    # Confirme se ela existe
    if test -d $pasta; then
        extra="existe"
    else
        extra="não existe"
    fi
    # E mostre o resultado na tela (item da lista)
    echo "<LI>$pasta ($extra)</LI>"
done
# Fecha a lista
echo "</UL>"
```



Diretórios do PATH em uma lista

Já melhorou um pouco, com as bolotas identificando cada diretório. Mas ainda pode melhorar mais. Uma página sem título parece algo incompleto, concorda? Então vamos colocar um. Que tal "Componentes de seu PATH"? Outra melhoria é realçar os textos "existe" e "não existe". Seria bom uma diferenciação maior entre eles, para o usuário poder bater o olho é já perceber quais diretórios existem e quais não. As cores sempre ajudam nesses momentos. Verde e vermelho são as cores clássicas para indicar positivo e negativo, vamos utilizá-las.

mostra_path_lista_cor.sh

```
#!/bin/bash
# mostra_path_lista_cor.sh
# Vamos mostrar uma página HTML
echo Content-type: text/html
echo
# Adiciona um diretório falso ao PATH
PATH=$PATH:/noel
# Título da página
echo "<h1>Componentes de seu PATH</h1>"
# Inicia uma lista de itens
echo "<UL>"
# Para cada pasta do $PATH...
IFS=:
for pasta in $PATH; do
```

```
    # Confirme se ela existe
    if test -d $pasta; then
        extra='<FONT COLOR="green">existe</FONT>'
    else
        extra='<FONT COLOR="red">não existe</FONT>'
    fi
    # E mostre o resultado na tela (item da lista)
    echo "<LI>$pasta $extra</LI>"
done
# Fecha a lista
echo "</UL>"
```

Diretórios do PATH usando cores

Opa, mas este programa está ficando bonito! O título dá um visual mais agradável e as cores melhoram o entendimento das informações, dispensando o uso de parênteses ao redor. Aliás, pausa obrigatória para mais uma regra sagrada dos programadores de bom senso: **informe sem distrair!** Tenha muito cuidado com o uso de cores e outros elementos que chamam a atenção do usuário. Não abuse de negritos e itálicos, não use várias fontes diferentes, não coloque imagens desnecessárias. Não faça da saída do seu programa um carnaval. Dê destaque somente ao que é importante: a informação. Afinal, se **tudo** chamar a atenção, então na verdade nada chamará a atenção. Pense simples, não exagere.

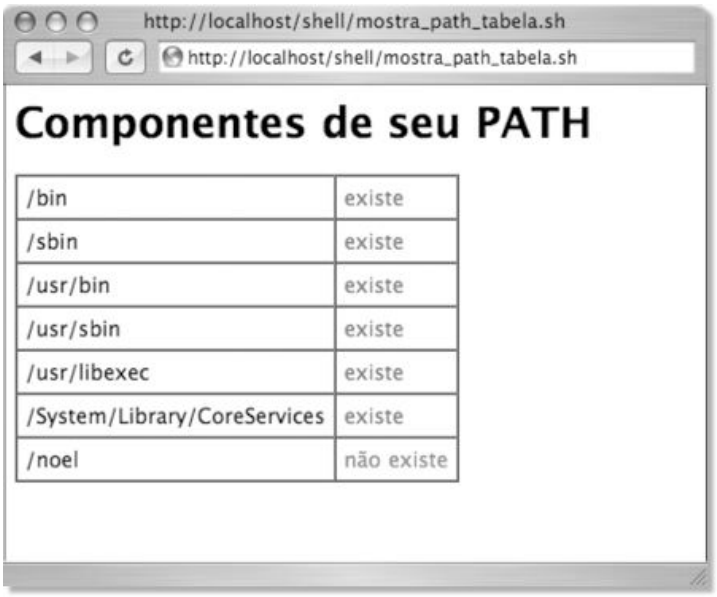
A página já está bonita e informativa, mas ainda temos outra alternativa

que se encaixa com este formato. Temos várias linhas, cada uma com dois tipos de informação: o diretório e a indicação se ele existe ou não. Pense comigo: linhas, dois campos, colunas... Hein? Hein?

mostra_path_tabela.sh

```
#!/bin/bash
# mostra_path_tabela.sh
# Vamos mostrar uma página HTML
echo Content-type: text/html
echo
# Adiciona um diretório falso ao PATH
PATH=$PATH:/noel
# Título da página
echo "<h1>Componentes de seu PATH</h1>"
# Inicia a tabela
echo '<TABLE BORDER="1" CELLSPACING="0" CELLPADDING="5">'
# Para cada pasta do $PATH...
IFS=:
for pasta in $PATH; do

    # Confirme se ela existe
    if test -d $pasta; then
        extra='<FONT COLOR="green">existe</FONT>'
    else
        extra='<FONT COLOR="red">não existe</FONT>'
    fi
    # E mostre o resultado na tela (linha da tabela)
    echo "<TR><TD>$pasta</TD><TD>$extra</TD></TR>"
done
# Fecha a tabela
echo "</TABLE>"
```



Diretórios do PATH em uma tabela, com dados alinhados

E assim vai. Sempre melhorando, sempre buscando o formato mais legível. É até covardia comparar esta página com a saída sem sal da versão original do programa, rodando no console. Se você possuía algum preconceito em relação à formatação, espero que com este exemplo isso tenha mudado. Ela é apenas mais uma ferramenta, que pode ser usada para ajudar ou atrapalhar. Cabe ao programador escolher qual caminho seguir.

E aí, captou o funcionamento geral do CGI com HTML? Seu programa faz todos os cálculos e processamentos que precisar normalmente, o importante é que tudo o que você queira mostrar para o usuário seja mandado para a saída padrão (`STDOUT`). E para não mandar um texto normal e sem graça, você adiciona umas tags do HTML que se encarregarão de formatar tudo em uma página bonita. Assim você formata a saída do seu programa, dando destaque aos dados mais importantes, organizando as informações, fazendo tabelas, mostrando gráficos...

As páginas resultantes podem ser tão complexas quanto necessário, não há limites. Com o CGI você não precisa se limitar ao texto puro, com as linhas todas iguais e sem muito destaque. Produza páginas atraentes e informativas, que podem ser impressas, que deem destaque à informação que o usuário procura.

Use modelos (Templates)

À medida que seus programas forem crescendo, você ficará cada vez mais cansado de colocar echos para cada linha que precisar mostrar, preocupando-se com aspas e escapes. O código também fica feio com as tags do HTML misturadas aos seus algoritmos, dificultando a leitura da lógica e do formato. Se eu lhe contar que há uma maneira mais limpa e prática, você promete não arrancar os cabelos?

A técnica é bem simples. Primeiro você vai fazer seu programa normalmente, com todos os laços, condicionais e cálculos necessários. O grande detalhe é que você não vai usar nenhum `echo`. Nada será mandado para a saída padrão, não se preocupe com formatação. Toda e qualquer informação processada deverá ser guardada em variáveis. Crie várias, tantas quantas necessitar.

Somente no final do programa, o último comando será o responsável por guardar toda a estrutura da página HTML e mostrá-la de uma vez, expandindo o valor das variáveis que você veio acumulando nas linhas anteriores. Veja um exemplo desta técnica:

modelo.sh

```
#!/bin/bash
# modelo.sh
# Vamos mostrar uma página HTML
echo Content-type: text/html
echo
# Obtém dados do usuário
login=$(whoami)
grupos=$(groups)
nome=$(grep "^$login:" /etc/passwd | cut -d : -f 5)
# Obtém dados do sistema
data=$(date "+%d.%m.%Y")
raiz=$(df / | tail -n 1 | tr -s ' \t' | cut -d ' ' -f 5)
barra_bin=$(ls -1 /bin | wc -l)
# Mostra o modelo, com os dados coletados
cat <<FIM
<H1>Dados do usuário</H1>
<UL>
```

```
    <LI><B>Nome:</B>   $nome</LI>
    <LI><B>Login:</B>  $login</LI>
    <LI><B>Grupos:</B> $grupos</LI>
    <LI><B>Home:</B>   $HOME</LI>
</UL>
<H1>Dados do sistema</h1>
<UL>
    <LI><B>Data atual:</B>         $data</LI>
    <LI><B>Diretório corrente:</B> $PWD</LI>
    <LI><B>Partição raiz:</B>      $raiz ocupada</LI>
    <LI><B>Programas no /bin:</B>  $barra_bin</LI>
</UL>
FIM
```

Execução do CGI usando modelo (template)

Veja como o código fica mais legível com esta separação clara entre o processamento e o formato. Se você precisar mudar a aparência da página, basta alterar o modelo no final do programa, sem correr riscos de criar algum bug ou bagunçar a lógica. De maneira similar, ao alterar algum algoritmo, não precisa preocupar-se que o formato final da página permanecerá inalterado.

Como diria o velho sábio: Uma coisa é uma coisa, outra coisa é outra coisa.

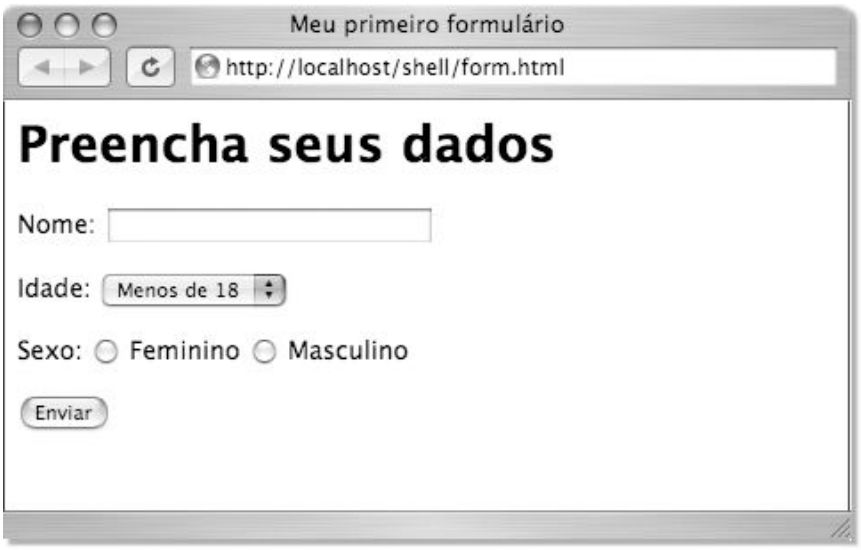
Outra vantagem em usar um modelo é ver a página toda de uma vez, sem precisar ficar pescando tags em meio a vários echos espalhados pelo código. Para facilitar ainda mais essa visão geral do formato, note que o texto foi alinhado e espaçado. O navegador não se importa com os espaços excedentes e as linhas em branco, então você pode usá-los sem medo, tornando seus modelos realmente fáceis de ler.

Formulários, a interação com o usuário

Já fizemos vários exemplos, aprendemos sobre HTML, montamos páginas bonitas e já sabemos como separar o código da formatação usando modelos. Os CGIs são programas bem mais atraentes do que seus primos executados na linha de comando. Mas, percebeu que está faltando alguma coisa?

O que fizemos até agora foram programas autônomos, que são chamados sem opções e mostram sempre a mesma saída. O usuário não tem o poder de informar dados ou fazer qualquer tipo de escolha. Na linha de comando é fácil contornar essa limitação utilizando a entrada padrão (`STDIN`) e as opções curtas (`-o`) e longas (`--`*`opção`*). Mas como fazer isso em um CGI se não temos linha de comando?

Lembre-se de que com o dialog montamos várias telas com caixas de texto, menus e botões, fazendo interfaces amigáveis. Agora vamos fazer algo parecido, montando páginas com formulários, que serão a interface do nosso programa com o usuário. Veja o exemplo que vamos construir:



Exemplo de formulário web

Etiquetas (tags) usadas em formulários

Resumo das tags usadas em formulários

Tipo	Descrição	Exemplo
`Form`	Formulário	`<form method="POST" action="/cgi-bin/programa.sh"> ... </form>`
`Text`	Caixa de texto	`<input type="text" name="nome" value="Texto" size="20">`
`Textarea`	Caixa de texto (várias linhas)	`<textarea name="notas" cols="40" rows="5">Texto</textarea>`
`Checkbox`	Opção (liga/desliga)	`<input type="checkbox" name="opcao" value="1" checked>Item 1`
`Radio`	Opção (exclusiva)	`<input type="radio" name="opcao" value="1" checked>Item 1`
`Select`	Menu	`<select name="menu"><option value="1">item 1</option></select>`
`Hidden`	Valor escondido	`<input type="hidden" name="nome" value="valor">`
`Submit`	Botão de envio	`<input type="submit" value="Enviar">`

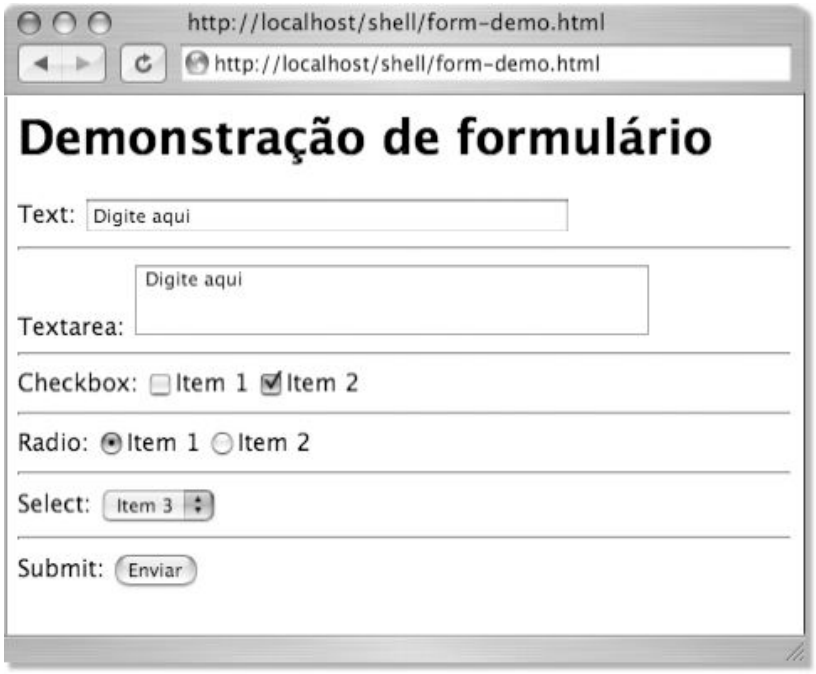
Detalhes:

- As tags de formulário são como as outras tags do HTML, você pode misturá-las com texto e com outras tags de formatação, como listas e tabelas.

- A tag `FORM` abriga seu formulário. Todas as outras tags do formulário devem estar dentro dela.
- O atributo `METHOD` da tag `FORM` geralmente é preenchido com `post`, pois vamos enviar dados a um CGI. Já o atributo `ACTION` é o endereço do CGI para quem os dados serão enviados ao apertar o botão Submit.
- A diferença entre o `Text` e o `Textarea` é que no segundo pode-se digitar várias linhas.
- A diferença entre o `Radio` e o `Checkbox` é poder escolher uma ou mais opções. O `Checkbox` deixa que várias opções sejam escolhidas, como ao escolher comidas e bebidas em um cardápio. Já no `Radio` somente uma opção pode ser escolhida, como ao preencher Masculino ou Feminino em um cadastro.
- O atributo `NAME` deve ser o mesmo para todas as tags de um grupo de `Checkbox` ou `Radio`.
- Coloque um atributo `CHECKED` para que um `Checkbox` ou um `Radio` já apareçam marcados no carregamento da página.
- De maneira similar, o atributo `SELECTED` indica qual o item selecionado por padrão em um `Select`.
- Em `Checkbox`, `Radio` e `Select`, a(s) escolha(s) do usuário são retornadas com o conteúdo do atributo `VALUE`, e não com o texto mostrado na tela.
- Use o atributo `SIZE="4"` para que um `Select` expanda e mostre quatro opções de uma só vez. É claro, troque 4 pelo seu número preferido.
- O atributo `SIZE` também pode ser usado para definir o tamanho de um Text. Use o atributo `MAXLENGTH` para definir o tamanho máximo do texto que o usuário vai digitar.
- Para definir o tamanho de um Textarea, use os atributos `ROWS` e `COLS` para informar o número de linhas e colunas.
- Por último, mas importantíssimo: o `Hidden` é o melhor amigo do programador. Como ele você pode esconder variáveis no formulário, que vão ajudar o programa a ter dados sobre o estado atual da

execução. Sempre que precisar guardar algum dado entre uma tela e outra, use o `Hidden`.

Agora, um exemplo completo de todas estas tags em ação:



Demonstração de todas as tags de formulário

form-demo.html

```
<h1>Demonstração de formulário</h1>
<form method="post" action="/cgi-bin/programa.cgi">

<!-- Hidden: -->
<input type="hidden" name="login" value="joaozinho">
<input type="hidden" name="email" value="joao@example.com">
<!-- Hidden são campos escondidos para passar informações ao CGI -
  ->
<!-- Ah, isso é um comentário em HTML ;) -->
Text:
<input type="text" name="texto" value="Digite aqui" size="40"
  maxlength="40">

<hr>
Textarea:
<textarea name="notas" cols="40" rows="3">Digite aqui</textarea>
<hr>
```

```
Checkbox:
<input type="checkbox" name="opcao" value="1">Item 1
<input type="checkbox" name="opcao" value="2" checked>Item 2
<hr>
Radio:
<input type="radio" name="opcao" value="1" checked>Item 1
<input type="radio" name="opcao" value="2">Item 2
<hr>
Select:
<select name="menu" size="1">
    <option value="1">Item 1</option>
    <option value="2">Item 2</option>
    <option value="3" selected>Item 3</option>
</select>
<hr>
Submit:
<input type="submit" name="botao" value="Enviar">
</form>
```

O primeiro formulário

Ficou empolgado ao ver todas as possibilidades que formulários em HTML oferecem? Você pode montar páginas realmente complexas, cheias de opções e menus, organizados dentro de tabelas, um verdadeiro painel de avião controlando a execução do seu CGI. Mas antes de dominar o mundo, vamos começar simples: faremos o exemplo mostrado no início deste tópico, o pequeno formulário que pergunta seu nome, idade e sexo.

Primeiro vou contar um pequeno detalhe que deixamos de lado para simplificar o aprendizado do HTML: uma página tem outras tags que definem a sua estrutura: HTML, HEAD e BODY. A primeira é o documento completo, e dentro dele temos um cabeçalho (head) e o conteúdo da página (body). No cabeçalho, podemos usar a tag TITLE que indica qual o título da janela do navegador. Até agora não usamos nenhuma destas tags, pois são todas opcionais, mas veja como fica uma página HTML completa:

esqueleto.html

```
<html>
    <head>
```

```
        <title> Título </title>
    </head>
    <body>
        Conteúdo
    </body>
</html>
```

Página HTML com título da janela e conteúdo

Então todos os exemplos que fizemos até agora encaixam-se ali no Conteúdo. Apesar de opcionais, é boa prática sempre usar estas tags para evitar problemas com navegadores antigos. E uma página que não define o título da janela não parece profissional, então sempre use o `TITLE` (e junto com ele, seus amigos `HTML`, `HEAD` e `BODY`).

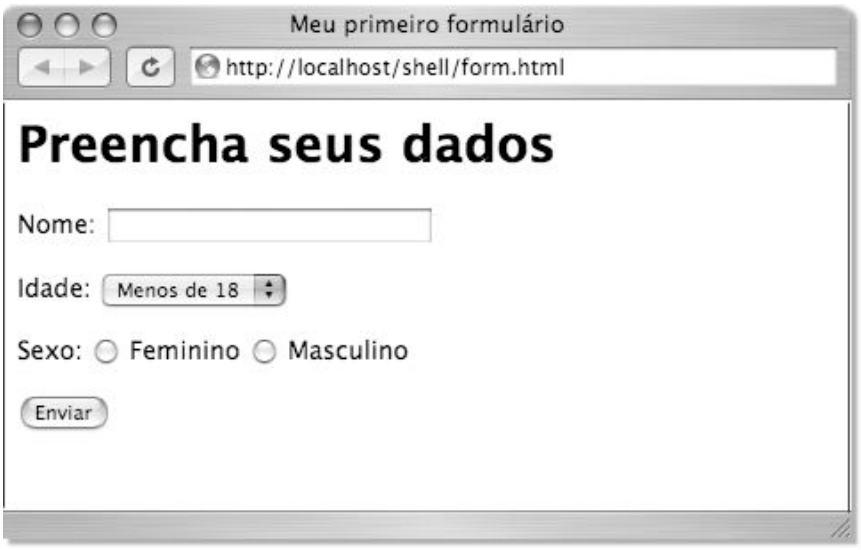
Agora que já sabemos como montar uma página HTML completa, podemos fazer o formulário de exemplo. Note que este formulário é uma página HTML normal e não um CGI em shell! O usuário preencherá seus dados e somente quando apertar o botão `Enviar` é que nosso programa será chamado. Então o primeiro passo é fazer o formulário:

form.html

```
<!-- form.html - Exemplo simples de formulário -->
<!-- Tag mãe de todas as outras. Abre no início e só fecha na
  última linha. -->
<html>
<!-- Ah, é assim que se define um título para a janela do
```

```
 navegador -->
<head>
    <title>Meu primeiro formulário</title>
</head>
<!-- Agora começa a sua página -->
<body>

<h1>Preencha seus dados</h1>
<!-- Aqui começa o formulário -->
<!-- O script form.sh é quem receberá os dados -->
<form method="POST" action="form.sh">
<p>Nome:
<input type="text" name="nome" size="25">
</p>
<!-- Note que o alinhamento com espaços não interfere na página --
 >
<p>Idade:
<select name="idade">
  <option value="17"   >Menos de 18  </option>
  <option value="18-30">Entre 18 e 30</option>
  <option value="30"   >Mais de 30   </option>
</select>
</p>
<!-- Note que o Radio usa o mesmo "name" para ambos -->
<p>Sexo:
<input type="radio" name="sexo" value="F"> Feminino
<input type="radio" name="sexo" value="M"> Masculino
</p>
<!-- O botão de envio -->
<p><input type="submit" value="Enviar"></p>
<!-- E fechamos o formulário -->
</form>
<!-- A tag mãe também deve ser fechada -->
</html>
```



Exemplo de formulário simples

A página vista no navegador é simples, mas seu código HTML já não é mais tão trivial. Leia com atenção este código, estude-o e perceba as diferenças da sintaxe de cada componente. Alguns detalhes importantes:

- Todo o formulário está contido dentro das tags `<FORM>` e `</FORM>`.
- Na abertura da tag `FORM` está indicado o programa `form.sh`, que é quem será chamado quando o usuário apertar o botão `Enviar` para submeter os dados. Este programa ainda não existe, logo adiante vamos escrevê-lo.
- A caixa de texto é uma única tag `INPUT` com o tipo `text`.
- O menu da idade é uma tag `SELECT` que contém várias tags `OPTION`, uma para cada item do menu.
- Ainda no menu da idade, o que importa para nosso programa é o conteúdo do atributo `VALUE` de cada `OPTION`, e não o texto mostrado para o usuário. Assim sendo, se o usuário escolher o segundo item, o programa receberá "18-30" e não "Entre 18 e 30".
- Para a escolha do sexo são usados dois `INPUT` do tipo `radio`, porém ambos possuem o mesmo valor no atributo `NAME`, para identificar a ligação entre eles.
- Assim como no `SELECT`, é o conteúdo do atributo `VALUE` que será enviado ao nosso programa, e não o texto mostrado na tela. Ou seja,

receberemos F e M em vez de Feminino e Masculino.

- Um outro `INPUT` do tipo `submit` representa o botão Enviar. O texto do botão é definido pelo atributo `VALUE`.

Não se assuste com o excesso de informações. Sempre que precisar volte a este código e releia os detalhes. É normal levar um tempinho até acostumar-se com a sintaxe diferenciada de cada componente.

Copie este código para um arquivo e use-o como modelo para seus formulários. Na prática é muito mais fácil copiar e colar os trechos desejados para o seu programa e alterar o que for necessário. Não preocupe-se em saber de cabeça todas as tags e seus atributos, reaproveite o que já foi digitado!

O formulário é a interface de seu programa com o usuário. Uma grande vantagem para o programador ao montar seus formulários é não precisar compilar nem executar nenhum programa para testar a interface. Basta abrir o arquivo HTML no navegador e já está tudo funcionando. É possível brincar com os botões e menus antes mesmo de escrever o CGI que vai lidar com as informações enviadas por eles. Por falar neste CGI, que tal o escrevermos agora?

Não se esqueça de que o form.sh deve ser um executável. Dá-lhe `chmod +x` nele!

form.sh (versão 1)

```
#!/bin/bash
# form.sh
# Vamos mostrar uma página HTML
echo Content-type: text/html
echo
echo "<H1>Os dados enviados foram:</H1>"
echo "<PRE>"
# Lê os dados do formulário via STDIN
read TRIPA
# Mostra os dados na tela
echo "$TRIPA"
```

Dados do formulário em uma tripa feia

Temos algumas novidades! Prepare-se para aprender aspectos importantes do uso de CGI com formulários. Os mistérios serão desvendados. As perguntas serão respondidas.

Se você está meio sonolento, agora é uma ótima oportunidade para levantar, esticar as pernas e tomar um copo d'água. Considere também jogar um pouco de água no rosto, caso a situação esteja mais grave :)

A famigerada tripa

Pelo código do `form.sh` é possível perceber que os dados do formulário são enviados ao nosso programa via entrada padrão (`STDIN`). Sim, ela voltou! Com um `read` carregamos todos estes dados para uma variável `TRIPA` e em seguida a mostramos na tela. Mas o que é essa sopa de letrinhas que apareceu?

```
nome=Maria+da+Silva&idade=18-30&sexo=F
```

Pode não parecer, mas estes são os dados preenchidos pelo usuário lá no formulário. O detalhe é que eles são empacotados em uma única "tripa", formando uma única linha sem espaços em branco nem quebras de linha. Esta tripa pode ser realmente comprida caso o usuário tenha digitado bastante texto ou seu formulário tenha muitos campos. Olhe com atenção, perceba que na tripa estão os pares de nomes dos campos e seus respectivos conteúdos, separados entre si pelo caractere `&`. São eles:

```
nome=Maria+da+Silva
```

```
idade=18-30
sexo=F
```

Já dentro do conteúdo de cada campo, todos os espaços em branco são trocados pelo sinal de mais "+". Com uma mudança na última linha de nosso programa podemos mostrar os dados na tela de maneira mais amigável:

form.sh (versão 2)

```
#!/bin/bash
# form.sh
# Vamos mostrar uma página HTML
echo Content-type: text/html
echo
echo "<H1>Os dados enviados foram:</H1>"
echo "<PRE>"
# Lê os dados do formulário via STDIN
read TRIPA
# Mostra os dados na tela
echo "$TRIPA" | tr '&+' '\n '
```

Dados da tripa foram desvendados

Que diferença faz um `tr` no fim do comando, hein? Ele está trocando cada "&" por uma quebra de linha (`\n`) e cada "+" por um espaço em branco. Mas simplesmente mostrar na tela não é o suficiente. Um programa precisa ler e manipular os dados enviados pelo usuário.

Analisando a situação atual, o que recebemos do navegador foi um texto (`string`) com todos os dados grudados. Porém há um padrão para a organização destes dados, tornando possível sua separação, classificação e processamento. Acabamos de conhecer este padrão, que não é nada complicado:

Caracteres especiais da tripa

Caractere	Significado
&	Separa os pares de campo/valor (Exemplo: `nome=Carlos&idade=30`).
=	Separa os componentes de cada par (Exemplo: `nome=Carlos`).
+	Máscara para os espaços em branco (Exemplo: `Maria+da+Silva`).

Então agora o que precisamos fazer é percorrer a tripa e ir identificando um por um os pares de *`campo=valor`*, para poder guardar em variáveis o conteúdo de cada campo. Com os dados guardados em variáveis, podemos programar normalmente em shell, pois daí adiante o ambiente CGI não influencia mais no processamento. Parece fácil, não? Quebra no "&", depois quebra cada "=" e por fim troca os "+" por espaços. Vamos lá:

form.sh (versão 3)

```
#!/bin/bash
# form.sh

# Vamos mostrar uma página HTML
echo Content-type: text/html
echo
########## PARTE 1 - Extrair os dados
# Lê os dados do formulário via STDIN
read TRIPA
# Usa o IFS para separar os pares de campo=valor
IFS='&'
set - $TRIPA
# Para cada par de dados...
for nome_valor; do

    # Extrai cada componente do par
    nome_campo=$( echo "$nome_valor" | cut -d= -f1)
    valor_campo=$(echo "$nome_valor" | cut -d= -f2 | tr + ' ')
```

```
    # Usa o eval para guardar em uma variável
    # Ex.: eval idade="18-30"
    eval $nome_campo=\"$valor_campo\"
done
########## PARTE 2 - Processar os dados
primeiro_nome=$(echo $nome | cut -d' ' -f1)
case "$idade" in
       17) idade="Menos de 18 anos";;
    18-30) idade="Entre 18 e 30 anos";;
       31) idade="Mais de 30 anos";;
esac
[ "$sexo" = F ] && sexo=Feminino || sexo=Masculino
########## PARTE 3 - Mostrar os dados (usando um modelo)
cat <<FIM
    <H1>Dados de $primeiro_nome:</H1>
    <UL>
        <LI><B>Nome: </B> $nome </LI>
        <LI><B>Idade:</B> $idade</LI>
        <LI><B>Sexo: </B> $sexo </LI>
    </UL>
FIM
```

Dados da tripa foram processados e formatados

Na primeira parte do código, os dados são extraídos um por um,

usando a nossa já conhecida técnica de mudar o IFS para poder percorrer cada par da tripa. Então o `cut` é usado para separar o nome do campo de seu valor, tendo o "=" como separador. Uma vez que os dados do par estão em variáveis (`$nome_campo` e `$valor_campo`), é usado o `eval` para criar variáveis do shell, com o mesmo nome dos campos: `$nome`, `$idade` e `$sexo`.

Com os dados já guardados em variáveis, na segunda parte do código é feito o processamento, manipulando os dados fornecidos. É extraído o primeiro nome da pessoa, a idade é analisada e seu texto correspondente é fixado, e para o sexo são usadas palavras completas em vez de F e M.

A terceira parte usa a técnica já aprendida dos modelos (templates) para construir uma página HTML simples, mostrando os dados que estão guardados nas variáveis. Mais uma vez fica clara a vantagem de se usar um modelo em vez de embutir o código HTML em vários echos pelo código. Tudo fica mais organizado e fácil de dar manutenção.

Perceba os pequenos detalhes que fazem a diferença para o usuário ter uma boa impressão de seu programa: (1) o primeiro nome da pessoa foi colocado no título, assim o resultado fica mais pessoal e simpático; (2) o negrito foi usado para destacar o nome de cada campo, facilitando a leitura das informações.

A segunda e terceira partes do código não trazem muitas novidades, pois é programação normal em shell. Já a primeira parte é a que decifra e extrai os dados do usuário, requerendo atenção dobrada para que nenhum detalhe escape. Acompanhe:

- Mudando o IFS para "&" são separados os pares de *`campo=valor`* da tripa.
- O "`set -`" encarrega-se de guardar cada par nas variáveis `$1`, `$2` etc.
- O loop em cada par (`$1`, `$2` etc.) é feito com o `for`, sem parâmetros.
- Ao ler o valor do campo, os espaços em branco (que estão como "+") são restaurados.
- O `eval` encarrega-se de executar a string *`campo="valor"`*, fazendo uma declaração de variável do shell.

Esta primeira parte do código é um trecho chato de fazer, porém pode ser reaproveitado em todos os seus CGIs. Basta copiar e colar. Ou até criar

uma função em sua biblioteca para que tudo fique centralizado.

A tripa não é assim tão simples (urldecode)

O que vimos no tópico anterior foi que a codificação da tripa resumia-se em "&" como separador de pares e "+" para os espaços em branco. Porém, esta foi uma simplificação para ajudar no aprendizado, pois a tripa esconde mais alguns segredinhos. Experimente digitar o seguinte nome no formulário:

```
José Simão (zé)
```

Veja como o nome aparece no resultado do programa:

Acentuação bagunçada: Jos%E9 ???

Ops! De onde apareceram todos estes "%"? Cadê os acentos? Cadê os parênteses do "(zé)"? Vamos fazer um comparativo do que foi digitado e do que o nosso programa realmente recebeu do navegador:

Texto digitado	Texto recebido
José	Jos%E9
Simão	Sim%E3o
(	%28
zé	z%E9
)	%29

Para que não restem dúvidas de que este é um problema do navegador e

não do nosso programa, vamos colocar um `echo` logo após a leitura da tripa para vermos exatamente o que o programa recebeu, antes de qualquer processamento:

```
########## PARTE 1 - Extrair os dados
# Lê os dados do formulário via STDIN
read TRIPA
echo "Tripa original: $TRIPA"
```

Acentuação codificada na tripa

É. Não foi o nosso programa quem bagunçou os dados... O que acontece é que há um mascaramento de símbolos e letras acentuadas. No empacotamento dos dados do formulário pelo navegador, somente as letras, números e alguns poucos símbolos são passados literais, como são. Todos os outros caracteres são mascarados para o formato *%HH*, onde *HH* é o código do caractere na tabela ASCII, em notação hexadecimal. Veja:

Hexa	Decimal	ASCII
E3	227	ã
E9	233	é
28	40	(
29	41	)

Seu CGI precisa decodificar estes caracteres também. Transformar cada *%HH* em seu equivalente ASCII. Que tal, agora complicou, né? Vou dar um tempinho para você pensar em uma solução.

Não, sério. Pensa aí. Volta aqui daqui a pouco.

E aí, tem algo? Bolou um esquema complicado de ler a tripa a cada caractere, achar os "%", extrair o número em hexadecimal, convertê-lo para decimal e depois obter o caractere ASCII equivalente? Muito bem, isso funciona perfeitamente. Mas lembre-se: estamos programando em shell, e não C! Temos toda a gama de comandos do sistema para nos auxiliar nesta tarefa.

Um bom programador shell sabe quais são suas ferramentas e, principalmente, quais as opções e funcionalidades que cada ferramenta possui. Sabe quem vai nos ajudar nesta tarefa complexa? O `echo`. Isso, aquele equinho de nada...

```
$ echo -e '\xE9'
é
$ echo -e '\x28'
(
$ echo -e 'Jos\xE9 Sim\xE3o'
José Simão
$
```

Usando a opção `-e` alguns caracteres especiais são interpretados pelo `echo`. Entre eles, o `\x` indica que os dois próximos caracteres representam um número hexadecimal, que o `echo` então converte para ASCII automaticamente. Muito prático!

Mas em vez de `\xE9` e `\x28`, temos `%E9` e `%28` na tripa. E agora?

Vamos lá, lembre-se de que você está no shell. Nada de percorrer vetores e lidar com ponteiros. O que um programador shell faz quando precisa que um *%HH* vire
`\x`*HH*? Ele simplesmente troca "%" por "\x" e é isso.

```
$ echo 'Jos%E9 Sim%E3o %28z%E9%29'
Jos%E9 Sim%E3o %28z%E9%29
$ echo 'Jos%E9 Sim%E3o %28z%E9%29' | sed 's/%/\\x/g'
Jos\xE9 Sim\xE3o \x28z\xE9\x29
$ echo 'Jos%E9 Sim%E3o %28z%E9%29' | echo -e $(sed 's/%/\\x/g')
José Simão (zé)
$
```

Vai, pode sorrir. Diga se não é lindo ver quando problemas que em um primeiro momento parecem monstros complexos e intimidantes são reduzidos a um mosquitinho inofensivo, com uma única linha de shell?

Agora ficou fácil. Basta separar este código em uma função bacana e usá-la na hora de extrair o valor de cada campo. Como o `sed` já lê da entrada padrão (`STDIN`) caso nenhum arquivo seja especificado, podemos usar esta função como filtro mesmo, encadeando sua chamada com o comando anterior via "`|`", veja:

```
$ urldecode() { echo -e $(sed 's/%/\\x/g') ; }
$
$ echo 'Jos%E9 Sim%E3o %28z%E9%29' | urldecode
José Simão (zé)
$
```

E com isso encerramos o assunto tripa. Sei que é chato e tem alguns detalhes a observar, mas você só vai codificar o tratamento da tripa uma única vez. Nos próximos CGIs, basta copiar e colar o código inicial. Ou melhor, fazer sua biblioteca de CGI e centralizar as ferramentas que serão compartilhadas por todos os seus programas. Além das funções para extração dos dados da tripa, você também pode fazer ferramentas bacanas para facilitar a criação de códigos HTML e formulários dinâmicos.

No site Shelldorado (www.shelldorado.com), há ferramentas prontas para o programação CGI em shell, que podem lhe interessar: `urldecode` (idêntico à nossa função de mesmo nome, porém usando AWK em vez de `echo -e`), `urlencode` (codifica um texto normal para usar os *%HH*, é o inverso da `urldecode`) e `urlgetopt` (formata a tripa para uso direto do eval sem precisar de loop nos pares, já decodificando os *%HH*).

Agora vamos atualizar a tabela dos caracteres especiais da tripa, bem como o código do nosso programa, que agora sabe como interpretar acentos e outros símbolos nos formulários.

Caracteres especiais da tripa

Caractere	Significado
&	Separa os pares de campo/valor(Exemplo: `nome=Carlos&idade=30`).
=	Separa os componentes de cada par (Exemplo: `nome=Carlos`).
+	

	Máscara para os espaços em branco (Exemplo: `Maria+da+Silva`).
%HH	HH é o código ASCII do caractere, em hexadecimal (Exemplo: `nome=Jos%E9`).

A acentuação da tripa foi decodificada

form.sh (versão final)

```
#!/bin/bash
# form.sh

# Vamos mostrar uma página HTML
echo Content-type: text/html
echo
# Função para decodificar acentos e símbolos na tripa
# Exemplo: "%E3" vira "á"
urldecode() {
    echo -e $(sed 's/%/\\x/g')
}
########## PARTE 1 - Extrair os dados
# Lê os dados do formulário via STDIN
read TRIPA
# Usa o IFS para separar os pares de campo=valor
IFS='&'
set - $TRIPA
# Para cada par de dados...
for nome_valor; do

    # Extrai cada componente do par
```

```
    nome_campo=$( echo "$nome_valor" | cut -d= -f1)
    valor_campo=$(echo "$nome_valor" | cut -d= -f2 | tr + ' ' |
  urldecode)
    # Usa o eval para guardar em uma variável
    # Ex.: eval idade="18-30"
    eval $nome_campo=\"$valor_campo\"
done
########## PARTE 2 - Processar os dados
primeiro_nome=$(echo $nome | cut -d' ' -f1)
case "$idade" in
       17) idade="Menos de 18 anos";;
    18-30) idade="Entre 18 e 30 anos";;
       31) idade="Mais de 30 anos";;
esac
[ "$sexo" = F ] && sexo=Feminino || sexo=Masculino
########## PARTE 3 - Mostrar os dados (usando um modelo)
cat <<FIM
    <H1>Dados de $primeiro_nome:</H1>
    <UL>
        <LI><B>Nome: </B> $nome </LI>
        <LI><B>Idade:</B> $idade</LI>
        <LI><B>Sexo: </B> $sexo </LI>
    </UL>
FIM
```

Mais alguns segredos revelados

Se você acha que com a decodificação da tripa já esgotamos todo o assunto CGI, enganou-se. Agora que você já sabe como fazer seu programa funcionar na Internet, lendo dados do usuário via formulário e mostrando o resultado no navegador, está na hora de conhecer mais alguns detalhes que serão muito importantes para uma programação de qualidade.

Variáveis especiais do ambiente CGI

Não contei porque ainda não precisamos delas, mas você sabia que seu CGI tem acesso a algumas variáveis especiais, que um shell normal não tem? Pois é. O Apache define algumas variáveis adicionais no ambiente de

execução do CGI, que guardam informações que podem ser muito úteis ao seu programa. São dados sobre o servidor e sobre o cliente. Veja alguns exemplos:

Variável	Descrição
DOCUMENT_ROOT	Diretório raiz dos documentos HTML.
HTTP_USER_AGENT	Nome do navegador do cliente.
QUERY_STRING	Parâmetros de formulário (a tripa!).
REMOTE_ADDR	IP do cliente.
REQUEST_METHOD	Método requisitado (GET ou POST).
REQUEST_URI	Endereço da página requisitada.
SERVER_ADDR	IP do servidor.
SERVER_NAME	Nome do servidor (configurado no Apache).

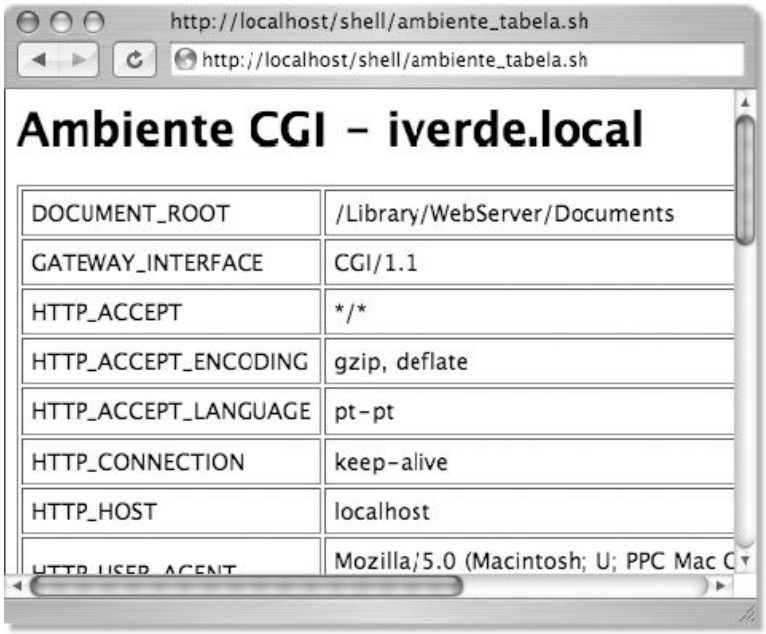
Depois de aprendermos todo esse HTML, formulários, tripa e codificações, é até covardia fazer um CGI para mostrar todas as variáveis de ambiente. Este é para degustar na hora do recreio, junto com os amiguinhos:

ambiente.sh

```
#!/bin/bash
# ambiente.sh
# Vamos mostrar uma página HTML
echo Content-type: text/html
echo
# Mostra o nome do servidor no título
echo "<h1>Ambiente CGI - $SERVER_NAME</h1>"
# Mostra as variáveis especiais do ambiente CGI
echo '<PRE>'
env | sort
```

env | sort no navegador

Mas este foi muito fácil! Já estamos chegando no final do livro, então agora é a sua vez de pôr em prática os conhecimentos aprendidos até aqui. Que tal melhorar a saída deste programa e colocar estes dados em uma tabelinha, facilitando sua leitura? Não é nada complicado, mas vai ser um bom teste para você mesmo saber se aprendeu o que estudou até aqui. Tente deixar a saída do programa o mais parecido possível com isso aqui:



Saída desejada do ambiente_tabela.sh

Uma possível resposta (sim, há mais de uma maneira de fazê-lo) está no final deste capítulo. Mas resista à tentação de olhar agora. Pense um pouco, faça testes, empenhe-se. A fixação do aprendizado é muito importante, e este exemplo vai ajudar bastante a consolidar todos estes conhecimentos que você estudou até aqui.

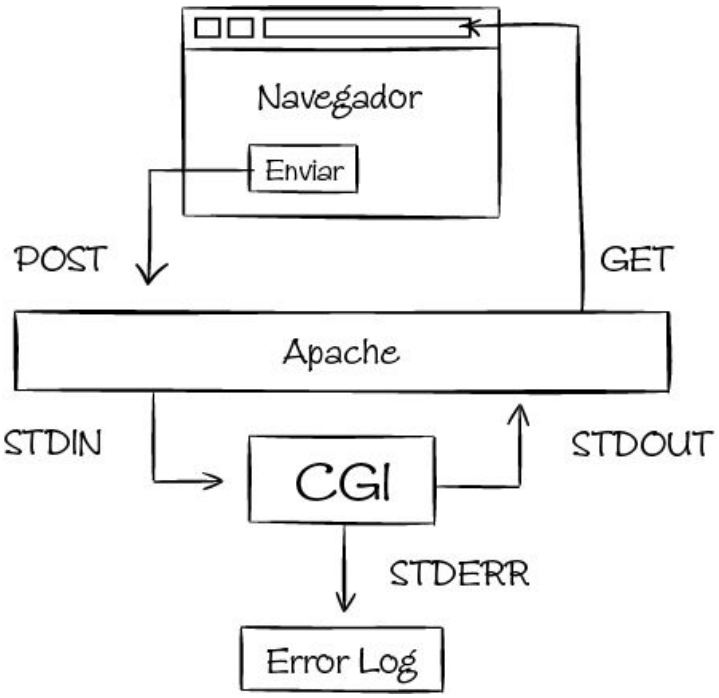
Enquanto você quebra a cabeça divertindo-se com o programinha, segue um exemplo de listagem completa das variáveis, para você consultar quando não estiver na frente do computador. Em sua máquina esta lista pode ter mais ou menos variáveis, dependendo da versão do Apache e de seu sistema operacional.

```
DOCUMENT_ROOT=/Library/WebServer/Documents
GATEWAY_INTERFACE=CGI/1.1
HTTP_ACCEPT=*/*
HTTP_ACCEPT_ENCODING=gzip, deflate
HTTP_ACCEPT_LANGUAGE=pt-pt
HTTP_CONNECTION=keep-alive
HTTP_HOST=localhost
HTTP_USER_AGENT=Mozilla/5.0 (Macintosh; U; PPC Mac OS X; pt-pt)
  AppleWebKit/419.2.1 (KHTML, like Gecko) Safari/419.3
PATH=/bin:/sbin:/usr/bin:/usr/sbin:/usr/libexec:/System/Library/Co
  reServices
PWD=/a/livro/shell/fontes/cgi
QUERY_STRING=
REMOTE_ADDR=127.0.0.1
REMOTE_PORT=49183
REQUEST_METHOD=GET
REQUEST_URI=/shell/ambiente.sh
SCRIPT_FILENAME=/a/livro/shell/fontes/cgi/ambiente.sh
SCRIPT_NAME=/shell/ambiente.sh
SCRIPT_URI=http://iverde.local/shell/ambiente.sh
SCRIPT_URL=/shell/ambiente.sh
SERVER_ADDR=127.0.0.1
SERVER_ADMIN=[no address given]
SERVER_NAME=iverde.local
SERVER_PORT=80
SERVER_PROTOCOL=HTTP/1.1
SERVER_SIGNATURE=Apache/1.3.33 Server at iverde.local Port 80
SERVER_SOFTWARE=Apache/1.3.33 (Darwin) PHP/4.4.4
```

```
SHLVL=1
_=/usr/bin/env
```

STDIN, STDOUT e STDERR

Talvez a maior dificuldade para quem começa a programar em CGI é a demora em acostumar-se a não poder usar a entrada e saída padrão, bem como a saída de erro. Mas elas todas ali, disponíveis para uso, embora de uma maneira diferente. Acompanhe como são as conexões entre seu programa, o Apache e o navegador do usuário:



Ligação das entradas e saídas do CGI

Conector	Fonte/Destino
`STDIN`	Método `POST` (dados do formulário).
`STDOUT`	Método `GET` (código HTML).
`STDERR`	Arquivo `error_log`.

A saída padrão de seu CGI será sempre o navegador do usuário. Sempre que você for colocar um `echo` ou um `cat`, lembre-se de que o resultado não irá para o console, mas, sim, para o navegador. Este resultado pode ser texto, código HTML ou outros formatos, será o `Content-type` quem indicará. O Apache faz o meio de campo, conectando as duas pontas.

Tudo o que seu programa mandar para a saída, o Apache repassa ao navegador.

A entrada padrão também está disponível, mas só será alimentada quando o usuário enviar um formulário lá no navegador. Você sabe, a tripa. Esta é a única maneira de o usuário manipular o funcionamento do seu programa: por intermédio de formulários. Quando ele aperta o botão Enviar, a tripa é mandada para a entrada padrão do seu CGI. Já vimos todos os detalhes de como os dados são codificados com os `&`, *`%HH`* e amigos.

A saída de erro é mandada para um arquivo de log no servidor. Leu a frase anterior? Então leia novamente, com atenção. Você vai se esquecer disso diversas vezes, mas sempre que algo não estiver funcionando direito e você não vir nenhuma mensagem de erro aparecendo, é bom olhar lá no arquivo de log que tudo será esclarecido.

Sistema	Arquivo de log dos erros (STDERR)
Linux, Mac	`/var/log/httpd/error_log`
Windows	`C:\Program Files\Apache Group\Apache\logs\error.log`

Mas é incômodo ter que monitorar o arquivo de log geral do servidor, ainda mais se várias pessoas o utilizam, as mensagens podem se perder em meio a outros erros. O melhor mesmo seria mandar os erros para o navegador também, assim fica fácil identificá-los durante o desenvolvimento do CGI.

A solução é conectar a saída de erro com a saída padrão, de maneira que o Apache envie ambas ao usuário. Há um comando no shell especial para esta tarefa, o exec. Para usá-lo, basta colocar a linha seguinte logo no início do CGI:

```
exec 2>&1
```

E pronto. Com este comando todos os erros serão mandados para a STDOUT, ou seja, o navegador. Que tal um teste rápido para conferir se essa dica funciona mesmo? Vamos fazer um programa rápido que gerará um erro. Este erro deve aparecer no navegador.

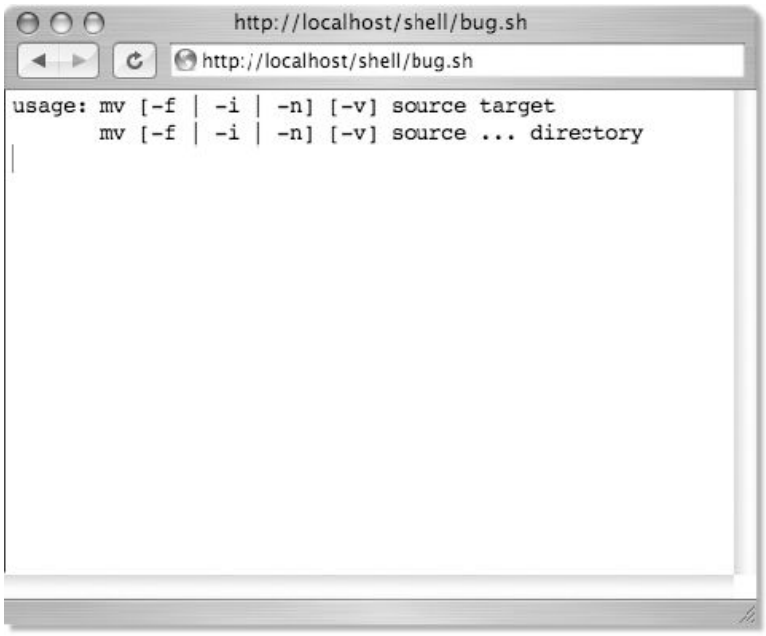
bug.sh

```
#!/bin/bash
```

```
# bug.sh
# Vamos mostrar texto
echo Content-type: text/plain
echo

# Conecta a STDERR com a STDOUT
exec 2>&1

# Ops!
mv
```



Saída de erro sendo mostrada no navegador

Como depurar CGIs (Debug)

Não se preocupe, pois a maioria das técnicas já aprendidas de como depurar programas em shell também são aplicáveis aos CGIs. Você pode continuar usando o `set -x` ou sua própria função `Debug()` e as mensagens de depuração vão aparecer no navegador para a sua comodidade. Veja a lista de compatibilidade:

Tipo de depuração	Funciona no ambiente CGI?
Verificação da sintaxe (`-n`)	SIM (pela linha de comando)
Debug com `echo`	SIM
Debug com a opção `-x`	Não (vem antes do `Content-type`)
Debug com o `set -x`	SIM (colocar depois do `Content-type`)

Execução passo a passo	Não (CGI não é interativo)
Debug personalizado	SIM
Debug categorizado	SIM

A verificação da sintaxe do script (`bash -n`) pode ser feita normalmente na linha de comando, já que ela independe de opções e de ambiente. O programa não é executado, porém toda a estrutura dos comandos é testada.

```
$ bash -n programa-cgi.sh
$
```

A depuração simples com o `echo` e a liga/desliga com o `set -x` também podem ser utilizadas normalmente, lembrando de que como o `set -x` usa a `STDERR`, esta deve ser conectada à `STDOUT` com o `exec`. Há uma única ressalva: a primeira linha que o programa manda para a saída deve continuar sendo a do `Content-type`, esta regra não pode ser quebrada. Então tome o cuidado de ligar o `set -x` e colocar os echos de debug somente depois da linha do `Content-type`. Por exemplo:

Errado:

```
#!/bin/bash

# Conecta a STDERR (usada pelo set -x) com a STDOUT
exec 2>&1

# Liga a depuração
set -x

echo Content-type: text/html
echo
```

Certo:

```
#!/bin/bash

echo Content-type: text/html
echo

# Conecta a STDERR (usada pelo set -x) com a STDOUT
exec 2>&1

# Liga a depuração
set -x
```

Por este motivo o uso da opção `-x` direto no cabeçalho (`#!/bin/bash -x`) não pode ser usada em ambiente CGI. As mensagens de depuração serão enviadas ao Apache antes do `Content-type`, resultando em erro no navegador. Outra depuração que não funciona é a passo a passo, pois o ambiente CGI não é interativo. Não há com o usuário mandar um Enter lá do navegador.

Por fim, as depurações personalizada e categorizada da sua função `Debug()` ganham ainda mais exposição. No console elas estão restritas aos caracteres de controle para dar algum destaque às mensagens. Já no CGI é possível formatar os textos utilizando as tags do HTML, deixando as mensagens bem visíveis e fáceis de ler. Até imagens podem ser utilizadas, caso o programador julgue que isso melhore a legibilidade dos registros. Um exemplo:

```
# Mostra mensagens de depuração em vermelho, com fundo amarelo
Debug(){
    [ "$DEBUG" != 1 ] && return
    echo '<P STYLE="color:red;background:yellow">'$*'</P>'
}
```

Como testar CGIs na linha de comando

Tá, tudo isso é muito legal. Mas durante o desenvolvimento do seu CGI, ficar toda hora recarregando a página no navegador ou preenchendo formulários vai ser muito chato. Felizmente, é possível testar o programa diretamente pela linha de comando, sem precisar de Apache, navegador e formulários.

Pense comigo:

- Já sabemos que todo CGI espera receber uma tripa pela entrada padrão.
- Já sabemos exatamente como é o formato desta tripa, com seus dados codificados.
- Já sabemos como alimentar a entrada padrão de um script pela linha de comando.

Então o que falta? Nada! Basta alimentar o CGI com uma tripa feita "na

mão" e esperar ele mandar o resultado para a tela. Simples assim. Vamos compor uma tripa com os dados fictícios da Ana Maria, 55 anos:

```
$ echo 'nome=Ana+Maria&idade=55&sexo=F' | ./form.sh
Content-type: text/html

        <H1>Dados de Ana:</H1>

        <UL>
                <LI><B>Nome: </B> Ana Maria </LI>
                <LI><B>Idade:</B> 55</LI>
                <LI><B>Sexo: </B> Feminino </LI>
        </UL>
$
```

Funciona perfeitamente. É exatamente este texto que o Apache recebe de nosso CGI quando o executamos pelo navegador. Assim conseguimos testá-lo rapidamente pela linha de comando. Ainda é possível fazer mais uma melhoria, limpando os espaços (`tr`) e tags HTML (`sed`) do resultado, para ficar mais legível a saída do programa:

```
$ tripa='nome=Ana+Maria&idade=55&sexo=F'
$ echo "$tripa" | ./form.sh | tr -d '\t' | sed 's/<[^>]*>//g'
Content-type: text/html
Dados de Ana:
Nome:  Ana Maria
Idade: 55
Sexo:  Feminino
$
```

Outra alternativa é interpretar o código HTML com um navegador de modo texto, como o lynx. Assim caso haja tabelas e outros formatos mais complexos, o texto aparecerá melhor formatado. Veja o exemplo:

```
$ echo "$tripa" | ./form.sh | lynx -dump -force_html /dev/stdin
   Content-type: text/html
                                  Dados de Ana:
     * Nome: Ana Maria
     * Idade: 55
     * Sexo: Feminino
$
```

A opção `-force_html` é necessária, pois a linha do `Content-type` no início não é

esperada em uma página HTML. Assim você diz ao lynx: Pode confiar, isso é mesmo uma página HTML.

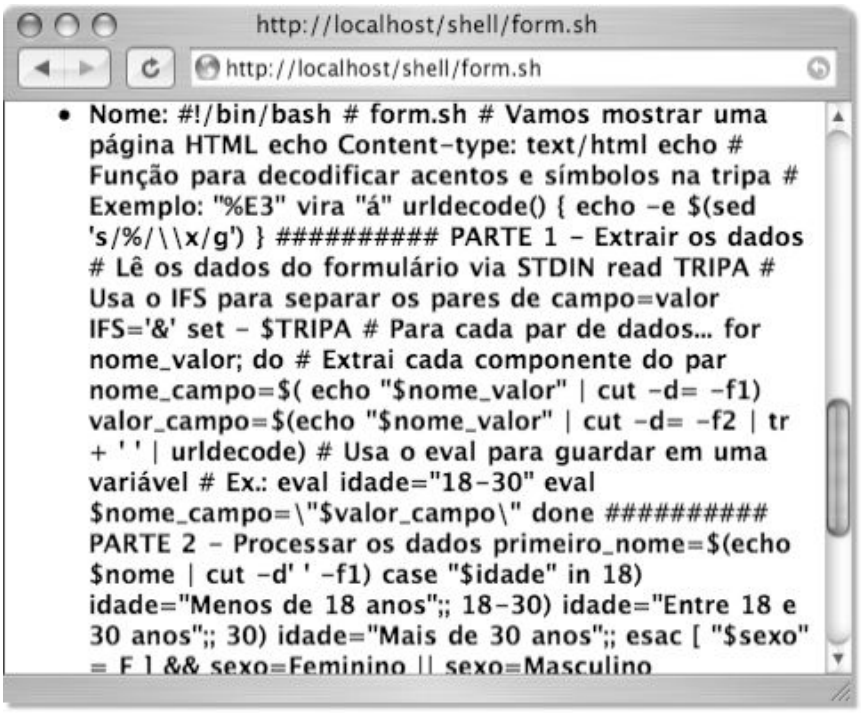
Outra utilidade do lynx é poder testar o programa pela linha de comando, porém usando toda a infraestrutura do Apache e seu ambiente do CGI. Com sua opção `-post-data` é possível simular o envio de um formulário, como se este tivesse mesmo sido feito pelo navegador, apertando o botão `Enviar`. Ele pega a tripa da entrada padrão, a envia ao Apache, recebe o resultado, mostra na tela e sai (por causa da opção `-dump`).

```
$ echo "$tripa" | lynx -post-data -dump
  http://localhost/shell/form.sh
                                    Dados de Ana:
     * Nome: Ana Maria
     * Idade: 55
     * Sexo: Feminino
$
```

O lynx já renderizou o HTML e veio somente o conteúdo da página, sem as tags. Caso você precise ver o código HTML gerado, basta trocar a opção `-dump` pela `-source`.

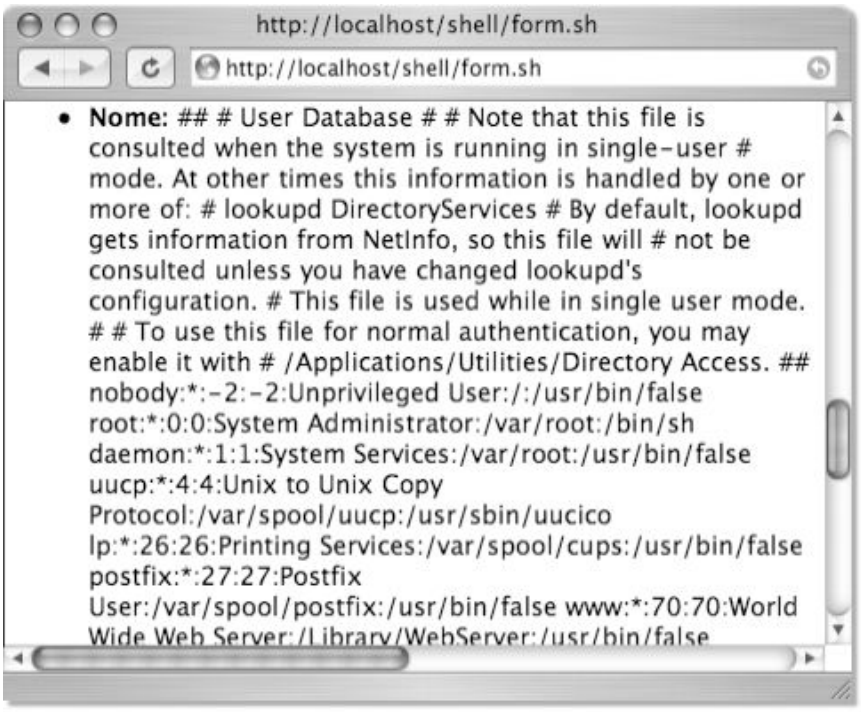
Considerações de segurança

Tudo ia bem na sua vida de programador CGI, o sol brilhava sempre e os passarinhos cantavam todas as manhãs. Mas em um dia qualquer, um piá de 14 anos com um conhecimento básico de Unix resolve preencher o seguinte nome em seu formulário: `$(cat $0)`



Hack: Código do programa

Com um sorrisinho cínico de satisfação estampado no rosto espinhento, ele resolve acabar mesmo com o seu dia, e talvez com seu emprego. Você já deve estar antecipando qual o comando, certo? Isso mesmo: `$(cat /etc/passwd)`



Hack: Conteúdo do /etc/passwd

Com apenas dois comandos embutidos no campo Nome do seu formulário, um visitante anônimo qualquer consegue ver todo o código do seu programa e o registro de todos os usuários do servidor. Mas hein? Furado? Que é isso... Seu CGI é uma peneira!

Outros "nomes" criativos que ele poderia tentar são:

- `$(whoami)`
- `$(set)`
- `$(uname -a)`
- `$(rm -rf *)` -- **Não teste este comando!**

E em poucos minutos vários dados importantes de seu servidor estarão nas mãos de um moleque. Nem preciso dizer o tamanho da encrenca que isso gera... Mas por que isso acontece?

Tem uma frase que diz que o `eval` é a raiz de todo mal (*eval is the root of all evil*). De fato, `eval` e `evil` são muito parecidos. Não que o comando em si seja diabólico, mas os estragos que seu uso descuidado pode causar, são muitos. Exatamente como estes furos de segurança que estamos descobrindo em nosso CGI. Aqui está o vilão:

```
eval $nome_campo=\"$valor_campo\"
```

Uma linha que parece tão inocente... O problema é que o `eval` **executa** o código, como se ele tivesse sido digitado por você, o programador. E sempre que o `eval` é usado em textos digitados pelo usuário, é ele quem se torna o programador. Você espera um nome de pessoa, mas ganhou uma subshell com comandos maliciosos.

O remédio é rápido, basta remover ou escapar os caracteres perigosos do texto digitado pelo usuário, antes da execução do `eval`. Para o shell, os caracteres mais críticos são o "`$`" e a crase. O cifrão pode mostrar conteúdos de variáveis e abrir subshells com o `$(...)`, sendo o mais perigoso. Vamos atualizar a `urldecode`:

```
# Função para decodificar acentos e símbolos na tripa
# Exemplo: "%E3" vira "á"
```

```
# Obs.: Remove os caracteres $ e ` por segurança
urldecode() {
    echo -e $(sed 's/%/\\x/g') | tr -d '$`'
}
```

Proteção anti-hackers funcionando

E com isso encerramos o assunto CGI. Agora você já pode codificar seu próprio Google e dominar o mundo. Quando estiver tomando champanhe em seu iate luxuoso no passeio pelas ilhas do Caribe, lembre-se do pobre mortal aqui que lhe deu umas diquinhas...

Ah! Antes que eu me esqueça:

ambiente_tabela.sh

```
#!/bin/bash
# ambiente_tabela.sh
# Vamos mostrar uma página HTML
echo Content-type: text/html
echo
# Mostra o nome do servidor no título
echo "<h1>Ambiente CGI - $SERVER_NAME</h1>"
# Mostra as variáveis especiais do ambiente CGI
echo '<TABLE BORDER="1" CELLPADDING="5">'
env | sort | fgrep = | sed 's/^/<TR><TD>/ ; s/=/<\/TD><TD>/'
```

;)

Capítulo 13

Dicas preciosas

Nenhum livro pode ensinar a experiência vivida, mas ele pode documentá-la. Programo em shell há vários anos, tendo feito muitos scripts e programas. Eles rodaram em muitas máquinas, configurações e sistemas diferentes. Cada ambiente trouxe seus próprios desafios, limitações e peculiaridades, exigindo adaptações nos programas. Aprenda com esta experiência, estudando com atenção cada uma destas dicas. Seus cabelos agradecerão ;)

Evite o bash2, bash3, bashN

Sempre que possível, resista à tentação de usar as funcionalidades novas, exclusivas de uma versão específica do Bash. Ao utilizá-las, seu programa não vai mais funcionar em outras máquinas que ainda tenham uma versão mais antiga do interpretador.

Por exemplo, no Bash versão 2 foi incluída a possibilidade de se usarem arrays. Muitos programadores esperavam ansiosamente por esta novidade, que facilita o trabalho de lidar com listas de dados. Mas para que o array funcione, a máquina onde seu programa for rodar precisa ter a mesma versão 2 instalada, ou alguma mais recente. Se for um servidor mais tradicional que há tempos não é atualizado, não vai funcionar.

Percebeu que os arrays não foram usados em nenhum código deste livro? Isso foi proposital, para assegurar que os exemplos irão funcionar no maior número possível de sistemas. Isso também prova que os arrays não são absolutamente necessários. São úteis e facilitam a programação, claro, mas se você se preocupa com compatibilidade, evite-os.

Se você tiver absoluta certeza que os sistemas nos quais seu programa vai rodar são de uma versão recente, então use os recursos novos. Ainda assim, é aconselhável colocar uma verificação já no início do programa para saber se o sistema tem a versão mínima necessária. Isso evita problemas imprevisíveis durante a execução. A versão atual do Bash pode ser obtida pela variável `$BASH_VERSION`.

Sempre use aspas

O título diz tudo: sempre use aspas. Sempre.

Use aspas na declaração de variáveis, nos argumentos para comandos, ao redor de toda e qualquer variável sendo referenciada. Use-as mesmo quando julgar que não é necessário. Não faz mal, o shell vai simplesmente ignorar. O mais importante é você adquirir o hábito de usá-las.

As aspas resolvem dois problemas muito corriqueiros, que qualquer pessoa que brincou de programar em shell por mais que alguns minutos deve ter enfrentado: testes em variáveis vazias e conteúdos com espaços

em branco. Veja como é fácil cair nestes erros:

Erros quando as aspas são esquecidas

Problemas	Tranquilo
`$ nome=` `$ test $nome = maria` `-bash: test: =: unary operator expected` `$ nome=Maria Silva` `-bash: Silva: command not found` `$ nome="Maria Silva"` `$ test $nome = maria` `-bash: test: too many arguments` `$ echo José Silva (zé)` `-bash: syntax error near unexpected token `('` `$`	`$ nome=` `$ test "$nome" = maria` `$ nome="Maria Silva"` `$ test "$nome" = maria` `$ echo "José Silva (zé)"` `José Silva (zé)` `$`

Os exemplos falam por si próprios, não é mesmo? Se você não usar aspas, as chances de enfrentar problemas são muito grandes. E veja como os erros que o Bash mostra na tela não ajudam em nada. Ele não vai dizer "coloque aspas", em vez disso vai sempre mostrar um erro que é a consequência direta pela falta de aspas. Por isso o conselho não abre concessões: **sempre** use aspas.

É importante que você crie o hábito de utilizá-las. Sua vida ficará mais tranquila. Não precisa ficar pensando "será que aqui precisa de aspas?", simplesmente coloque e pronto.

É igual dirigir um carro em cidades com radar de 60 Km/h. Você pode correr sempre e ter que ficar diariamente preocupado onde estão todos os radares, para frear em cima e não tomar multa. Ou você simplesmente anda sempre abaixo de 60 e deixa de se preocupar.

Exemplos de uso das aspas

Perigoso	Recomendado

`nome=Maria`
`idade=33`
`arquivo=$PWD/dados.txt`
`echo Aperte ENTER para continuar`
`echo Hoje é $(date)`
`echo Estou no diretório $PWD`
`test $arquivo = /tmp/dados.txt`
`grep root /etc/passwd`
`sed s/isso/aquilo/g $arquivo`
	`nome="Maria"`
	`idade="33"`
	`arquivo="$PWD/dados.txt"`
	`echo "Aperte ENTER para continuar"`
	`echo "Hoje é $(date)"`
	`echo "Estou no diretório $PWD"`
	`test "$arquivo" = "/tmp/dados.txt"`
	`grep "root" /etc/passwd`
	`sed "s/isso/aquilo/g" "$arquivo"`

Na declaração de variáveis, sempre use aspas. Mesmo que o conteúdo for apenas um número. Assim você se acostumará a usá-las e garantirá que nada vai quebrar quando aquele 33 um dia mudar para "33 anos". E a `$PWD`, será que não tem algum diretório com espaços em branco no nome? Aspas nela!

Para o `echo` também. Use aspas e coloque todo o conteúdo dentro delas, seja texto, variáveis ou subshells. O mesmo vale para variáveis usadas em comandos, seja no argumento ou no nome do arquivo.

Tem uma técnica bem simples que vai ajudar a nunca mais esquecer das aspas. Vai escrever um `echo`? Primeiro faça `echo ""`, depois volte para preencher o conteúdo das aspas. Vai declarar uma variável? Faça o mesmo: `nome=""` e volte o cursor para preencher o conteúdo.

Esta mesma técnica é útil ao fazer uma função, um `if` ou um `while`. Primeiro digite toda a estrutura do comando: `if<ENTER>then<ENTER>else<ENTER>fi` depois volte para colocar os comandos no meio. Assim você garante que não vai se esquecer de fechar os blocos.

São raras as situações nas quais você realmente precisará do conteúdo de uma variável sem as aspas. Por exemplo, quando não quiser mostrar os espaços em branco consecutivos, deixando que o Bash retire-os automaticamente. Mas isso é exceção, sempre use aspas.

Cuide com variáveis vazias

Tenha muito cuidado sempre que for usar variáveis quando fizer um comando perigoso que vá remover arquivos (`rm`, `mv`) ou usar o operador de redirecionamento >. Veja três exemplos de comandos que podem ser encontrados em um instalador de um software qualquer:

```
rm -rf $INSTALL_DIR/*
mv ~/$INSTALL_DIR /tmp
grep root /etc/passwd > $INSTALL_DIR/etc/passwd
```

Nada de errado com estes comandos, certo? O primeiro remove o software, o segundo move o programa para o diretório temporário (para fazer algum processamento) e o terceiro cria um arquivo de usuários dentro do diretório do programa, para seu uso. Agora, imagine que por

algum motivo (bug ou intervenção do usuário) nestes pontos do código a variável `INSTALL_DIR` esteja vazia. Veja como o shell receberá estes comandos:

```
rm -rf /*
mv ~/ /tmp
grep root /etc/passwd > /etc/passwd
```

O que você prefere? (a) Que todos os arquivos e diretórios do sistema sejam removidos, (b) que todos os arquivos do seu diretório pessoal (`HOME`) sejam movidos para o `/tmp` ou (c) que seu arquivo `/etc/passwd` seja corrompido? :)

Fica o conselho: sempre que variáveis estiverem sendo usadas em comandos perigosos, avalie o que acontecerá se ela for vazia. Caso haja perigo de perda de arquivos ou dados, defina um valor padrão para a variável:

```
rm -rf ${INSTALL_DIR:-vaziavaziavazia}/
```

Viu como é muito fácil definir um valor padrão usando a própria expansão de variáveis do Bash? Neste caso, se a variável estiver vazia, o shell receberá o inofensivo comando:

```
rm -rf vaziavaziavazia/
```

Variáveis são variáveis! Nunca confie cegamente em uma! :)

Evite o eval

Essa dica você já deve saber, não é mesmo? Já vimos problemas sérios de segurança pelo uso descuidado do `eval` nos capítulos Arquivos de configuração (página 192) e Programação Web (página 323). É bom reler estes dois trechos agora para refrescar a memória.

O ideal é não usar o `eval`. Nunca.

Mas se não houver outra maneira, sempre tenha em mente que o conteúdo da variável será executado como um comando normal do shell. E se este conteúdo vier de um texto digitado pelo usuário, é ele quem se torna o programador. Um prato feito para hackers de plantão abusarem

de seu programa.

Nestes casos, é **extremamente** recomendado que você remova ou escape os caracteres perigosos (cifrão e crase) antes da execução do `eval`. Não custa nada, é baratinho, um simples `tr` resolve: `tr -d '$`'`.

Lembre-se: eval is the root of all evil.

Use && e || para comandos curtos

Salvo algumas raras exceções, comandos em uma forma compacta são menos legíveis que seus equivalentes na forma normal. Agora veremos uma destas exceções. Na verdade, já vimos em Opções de linha de comando (página 81).

Ao fazer um `if` que tem apenas um único comando dentro do `then`, pode ser melhor usar os operadores `&&` e `||` em seu lugar. O comando ficará todo em uma única linha, sendo mais fácil de visualizar a relação direta entre o teste e o comando atrelado a ele.

Especialmente útil dentro de um `while`, quando você for fazer um teste para em seguida executar um simples `break` ou um `continue`. No par de teste/criação de diretórios também pode ser uma boa. Veja alguns exemplos:

Comandos mais legíveis com && e ||

if / then / fi	&& e ||

`if test $i -gt 10`
`then`
`  break`
`fi`
	`test $i -gt 10 && break`

`if grep -q "ana" nomes.txt`
`then`
`  existe=1`
`fi`
	`grep -q "ana" nomes.txt && existe=1`

`if ! test -d /tmp/cache`
`then`
`  mkdir /tmp/cache`
`fi`
	`test -d /tmp/cache || mkdir /tmp/cache`

Prefira o $(...) ao `...`

Ao precisar de uma subshell, sempre use a sintaxe com cifrão e parênteses em vez das crases.

Ambos são similares, porém a crase é uma fonte eterna de problemas visuais. É muito fácil confundi-la com um apóstrofo ('), um acento agudo (´) ou aspas ("). Por ser um caractere pequeno, também é difícil enxergá-lo no meio do código. Veja a diferença:

```
$ echo "'`whoami`'"
'aurelio'
$ echo "'$(whoami)'"
'aurelio'
$ echo "'`whoami`'whoami'`whoami`'"    # mas hein?
'aurelio'whoami'aurelio'
$ echo "'$(whoami)'whoami'$(whoami)'"  # ah, bom
'aurelio'whoami'aurelio'
$
```

Outra vantagem dos parênteses fica bem clara quando é preciso aninhar comandos, colocando uma subshell dentro da outra. Com as crases isso é possível, porém é necessário escapá-las, e, a cada nível que se desce, é preciso um número maior de escapes, para escapar o escape, sabe como?

```
$ echo `echo \`echo \\\`echo \\\\\\\`echo uau\\\\\\\`\\\`\``   #
  1, 3, 7, ?
uau
$ echo $(echo $(echo $(echo uau)))    # ufa, bem melhor
uau
$ set -x                                        # só para
  conferir
$ echo `echo \`echo \\\`echo \\\\\\\`echo uau\\\\\\\`\\\`\``
+++++ echo uau
++++ echo uau
+++ echo uau
++ echo uau
+ echo uau
uau
$
```

Quer ainda mais um motivo? Dou-lhe. Ao colocar vários escapes

consecutivos dentro das crases, o resultado beira o imprevisível.

Escapes consecutivos em subshells

Estranho		Certo
`$ echo `echo \\``	2	`$ echo $(echo \\)`
`\`	4	`\`
`$ echo `echo \\\\``	6	`$ echo $(echo \\\\)`
`\\`	8	`\\`
`$ echo `echo \\\\\\``	10	`$ echo $(echo \\\\\\)`
`\\\`	12	`\\\`
`$ echo `echo \\\\\\\\``		`$ echo $(echo \\\\\\\\)`
`\\\\`		`\\\\`
`$ echo `echo \\\\\\\\\\``		`$ echo $(echo \\\\\\\\\\)`
`\\\\\`		`\\\\\`
`$ echo `echo \\\\\\\\\\\\``		`$ echo $(echo \\\\\\\\\\\\)`
`\\\\\\`		`\\\\\\`

A próxima vez que aquele seu amigo sabichão vier lhe dizer que as crases são idênticas ao `$(...)`, bata com este livro na cabeça dele. Depois você mostra esta página e ensina a diferença, mas primeiro ele ganha uma livrada :)

Para não ser totalmente contra as crases, elas têm uma única vantagem sobre a sintaxe com os parênteses: são mais portáveis, pois funcionam na grande maioria dos Unix antigos.

Evite o uso inútil do ls

Não use o `ls` para fazer uma lista de arquivos quando o próprio shell já pode fazer isto diretamente.

É comum ver o uso desnecessário do `ls` em códigos que fazem loops em todos os arquivos de um diretório, ou, ainda, todos os arquivos com uma determinada extensão:

```
for arquivo in $(ls *.txt)
```

Lembre-se que o próprio shell sabe expandir o `*.txt` para "todos os arquivos com a extensão `.txt`". Não é necessário chamar o `ls` neste caso, você está usando recursos do sistema à toa. E pior: usando o `ls` o resultado pode não ser exatamente o que você espera. Sabe os arquivos com espaços em branco no nome? Sempre eles...

```
$ mkdir uuol
$ cd uuol
$ touch "didi moco" "dede santana" mussum zacarias
```

```
$ ls -1
dede santana
didi moco
mussum
zacarias
$ for arquivo in *; do echo "---- $arquivo"; done
---- dede santana
---- didi moco
---- mussum
---- zacarias
$ for arquivo in $(ls *); do echo "---- $arquivo"; done
---- dede
---- santana
---- didi
---- moco
---- mussum
---- zacarias
$
```

Evite o uso inútil do cat

Essa dica é bem curtinha, mas é a campeã de vendas da região: não use o `cat` quando não precisa (`UUOC`).

```
$ cat /etc/passwd | grep ^root
root:*:0:0:System Administrator:/var/root:/bin/sh
$ grep ^root /etc/passwd
root:*:0:0:System Administrator:/var/root:/bin/sh
$ grep ^root < /etc/passwd
root:*:0:0:System Administrator:/var/root:/bin/sh
$
```

Usar o `cat` para mandar apenas um arquivo para um pipe é um desperdício de recursos do sistema. Muitos comandos (como o `grep`) já aceitam receber o nome do arquivo diretamente e sabem o que fazer com ele. Para os outros comandos (como o `tr`) que não aceitam receber arquivos, basta usar o redirecionamento de entrada < para alimentar a `STDIN`. Por exemplo: `tr a-z A-Z < /etc/passwd`.

Evite a pegadinha do while

Essa pegadinha geralmente acontece ao fazer um loop nas linhas de um arquivo. Você faz o que tem que fazer, define variáveis, faz cálculos. Mas ao sair do loop, todas as variáveis estão vazias!

Pegadinha do while

Não funciona	Funciona

`i=0`
`cat /etc/passwd | while read LINHA`
`do`
`    i=$((i+1))`
`    echo "Linha $i: $LINHA"`
`done`
`echo "Número total de linhas: $i"`
	`i=0`
	`while read LINHA`
	`do`
	`    i=$((i+1))`
	`    echo "Linha $i: $LINHA"`
	`done < /etc/passwd`
	`echo "Número total de linhas: $i"`

Em ambos os programas, todas as linhas do `/etc/passwd` serão mostradas na tela, precedidas do seu número. A diferença está na última linha que mostra o número total: no programa da esquerda a variável `$i` estará zerada enquanto no da direita ela estará com o número total.

Por que um funciona e o outro não? O que acontece é que ao fazer *`comando`* `| while`, por causa do pipe este `while` é executado em uma nova shell. Assim, todas as variáveis gravadas dentro dele só valem para este ambiente novo e serão descartadas quando o loop terminar. Já usando o redirecionamento de entrada não há pipe, então não há nova shell, então as variáveis permanecem. Veja um exemplo rápido:

```
$ cat /etc/passwd | while read L; do gasparzinho="bu"; done
$ echo $gasparzinho

$ while read L; do gasparzinho="bu"; done < /etc/passwd
$ echo $gasparzinho
bu
$
```

Cuidado com o IFS

A variável especial `IFS` pode ser uma amiga ou uma pedra no sapato, dependendo da maneira como você se relaciona com ela.

Você sabe, Internal Field Separator. Mudar o separador padrão do shell pode ser muito útil nos casos em que os dados com os quais queremos interagir não usam brancos como separador padrão.

Usamos o `IFS` duas vezes no capítulo Programação Web (páginas 286 e 306): uma vez para fazer loop nos diretórios do `$PATH` e outra para um loop na tripa do navegador, que traz o `&` como separador.

Muito cuidado ao alterar o valor do `IFS` pois isso afeta o funcionamento interno do shell. Para prevenir-se contra comportamentos estranhos, primeiro guarde em uma variável o valor original do `IFS`, então mude-o para fazer seu loop, depois restaure o valor original. Veja um exemplo:

```
OLDIFS="$IFS"                  # salva o valor original
IFS=:                          # muda o IFS
for diretorio in $PATH; do     # faz o loop
    echo "$diretorio"
done
IFS="$OLDIFS"                  # restaura valor original
```

No Bash, basta fazer `unset IFS` para a variável voltar ao seu valor padrão.

Leia man pages de outros sistemas

Sempre que possível, leia as man pages dos comandos básicos (`grep`, `cut`, `tr`, `sed` e amigos) de outros sistemas. Assim você saberá qual o denominador comum que funcionará na maioria deles. A opção `-u` do `sort` funciona no Unix? Quais das opções do `grep` são exclusivas da versão GNU?

Man Pages Online	
AIX	http://publib16.boulder.ibm.com/cgi-bin/ds_rslt#1
HP-UX	http://docs.hp.com/en/hpuxman_pages.html
IRIX	http://techpubs.sgi.com/library/tpl/cgi-bin/browse.cgi?coll=0650&db=man&pth=ALL
Linux	http://linuxreviews.org/man
Mac OS X	http://developer.apple.com/documentation/Darwin/Reference/ManPages/
Solaris	http://docs.sun.com/app/docs/coll/40.7
Unix	http://www.opengroup.org/onlinepubs/007908799/idx/

Aprenda lendo códigos

Ler um livro como este vai lhe dar uma base muito boa para fazer seus próprios programas de qualidade. Mas certos ensinamentos só são aprendidos ao ver um exemplo funcional, do mundo real.

Acostume-se a ler o código-fonte de outros programas em shell, para aprender técnicas novas. Cada código traz a experiência do programador que o criou, com soluções para diversos tipos de problemas que apareceram durante o tempo de vida do programa. Se o código estiver limpo e comentado, melhor ainda!

Não deixe de estudar o tópico Análise das Funções ZZ (página 367), que, além de trazer um exemplo de código-fonte alinhado e comentado, ainda explica em detalhes várias das técnicas utilizadas nos trechos mais importantes.

E mais: você sabia que a sua própria máquina é uma ótima fonte de referência para aprender sobre shell? Há muitos scripts nela, desde os componentes vitais para o processo de inicialização (boot), os chamados initscripts, até comandos que você pode utilizar no seu dia a dia. Veja:

```
$ file /bin/* /usr/bin/* | grep -i shell
/usr/bin/alias:          Bourne shell script text executable
/usr/bin/apropos:        Bourne shell script text executable
/usr/bin/autoconf:       Bourne shell script text executable
/usr/bin/bashbug:        Bourne shell script text executable
/usr/bin/bg:             Bourne shell script text executable
/usr/bin/bzdiff:         Bourne shell script text executable
/usr/bin/bzgrep:         Bourne shell script text executable
/usr/bin/bzmore:         Bourne shell script text executable
/usr/bin/cd:             Bourne shell script text executable
/usr/bin/command:        Bourne shell script text executable
...
```

Esta lista é gigantesca. Agora você não pode mais reclamar que não tem exemplos de programas em shell... Estude estes programas com atenção. Alguns são bem-escritos, alguns são comentados, outros são enigmáticos. Mas a melhor parte é que o código de todos eles está ali, esperando para ser desvendado e dominado. Agora é com você!

Faça uso minimalista das ferramentas

Esta última dica é bem subjetiva e abrangente, mas o seu domínio vai fazer toda a diferença se você quiser que seus programas sejam reconhecidos por sua portabilidade e eficiência: menos é mais.

Primeiro, alguns dados fatuais:

```
$ du -h $(which cut tr sed grep awk)
20K    /usr/bin/cut
20K    /usr/bin/tr
36K    /usr/bin/sed
104K    /usr/bin/grep
124K    /usr/bin/awk
$
```

Agora, as sementes da discórdia:

- Procure usar o menor número de recursos do sistema em seus programas.
- O cut tem uma fração do tamanho do `awk`. Você realmente precisa usar o `awk`? Será que o cut não faria a mesma tarefa? Ou quem sabe o `sed`?
- E o `tr`, ele é menor que o `sed`. Será que ele não poderia substituir (ha!) algum comando `sed` do seu programa?
- Olha o tamanhão do `grep`! Para comandos simples o `sed` pode simular o `grep`.
- Bem, o `sed` também pode simular o `cut` e o `tr` em alguns casos...
- Por outro lado, o `awk` sozinho pode simular todos os outros. É melhor usar somente ele e ter menos dependências externas?
- Saiba quais são os comandos internos (builtin) do shell e procure usá-los sempre, pois são mais rápidos e consumirão menos recursos do que os comandos externos.
- Fora os comandos internos do shell, quantas ferramentas externas seu programa está utilizando? Todas elas são realmente necessárias?
- Os sistemas em que seu programa será executado terão todas as

ferramentas que ele precisa? Se sim, a versão delas será compatível com a que você utilizou?

- É possível simular um `grep` usando só comandos internos do shell, como `while`, `read` e `if`. Mas fica muito mais lento...
- Se você usar as ferramentas GNU, seu programa não vai funcionar em um Unix. Se você usar as ferramentas mais limitadas do Unix, vai ter mais trabalho para codificar tarefas que as ferramentas GNU já fazem sozinhas... O que é melhor?
- **Portabilidade vs eficiência vs manutenção facilitada**. Aprenda a equilibrar estas três forças opostas e você será um Jedi do shell.

Apêndice A

Shell básico

Se você nunca brincou de shell e quer saber do que se trata, então estude este capítulo com atenção, digite os exemplos e tente fazer todos os exercícios. Se você já conhece o shell e está precisando de um refresco para a memória antes de mergulhar nos estudos deste livro, passeie por este capítulo e teste seus conhecimentos. Se você já leu todo o livro e acompanhou o conteúdo sem dificuldade, pode saltar este capítulo sem dó.

Apresentação

O que é o shell

O shell é o *prompt* da linha de comando do Unix e do Linux, é o servo que recebe os comandos digitados pelo usuário e os executa.

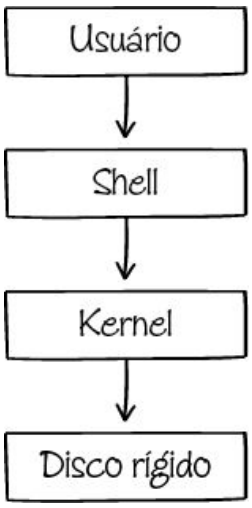
O shell é aquele que aparece logo após digitar-se a senha do usuário e entrar na tela preta. Ou na interface gráfica, ao clicar no ícone do Xterm, rxvt, Terminal ou Console.

```
localhost login: root
Password:
Last login: Fri Apr 16 01:57:28 on tty5
[root@localhost root]# _
```

Ali está o shell, esperando ansiosamente por algum comando para ele poder executar. Essa é a sua função: esperar e executar. Cada comando digitado é lido, verificado, interpretado e enviado ao sistema operacional para ser de fato executado.

No Mac OS X, o shell está em **Aplicativos > Utilitários > Terminal**. No Windows é preciso instalá-lo com o Cygwin. Veja mais detalhes no tópico Shell no Linux, Mac e Windows, página 364.

Funcionando como uma ponte, o shell é a ligação entre o usuário e o kernel. O kernel é quem acessa os equipamentos (hardware) da máquina, como disco rígido, placa de vídeo e modem. Por exemplo, para o usuário ler um arquivo qualquer, toda esta hierarquia é seguida:



Para os usuários do Windows, é fácil pensar no shell como um MSDOS melhorado. Em vez do `C:\>` aparece um `[root@localhost root]#`, mas o funcionamento é similar:

- Basta digitar um comando, suas opções e apertar a ENTER que ele será executado.
- O comando deve estar no PATH.
- Mensagens de aviso são mandadas para a tela.
- Ctrl+C interrompe o funcionamento..

Isso tudo é igual em ambos. Mas o shell é muito mais poderoso que seu primo distante. Além dos comandos básicos para navegar entre diretórios e manipular arquivos, ele também possui todas as estruturas de uma linguagem de programação, como `IF`, `FOR`, `WHILE`, variáveis e funções. Com isso, também é possível usar o shell para fazer scripts e automatizar tarefas.

Este será o nosso foco: scripts em shell.

Shell script

Um script é um arquivo que guarda vários comandos e pode ser executado sempre que preciso. Os comandos de um script são exatamente os mesmos que se digita no prompt, é tudo shell.

Por exemplo, se de tempos em tempos você quer saber informações do sistema como horário, ocupação do disco e os usuários que estão logados,

é preciso digitar três comandos:

```
[root@localhost root]# date
[root@localhost root]# df
[root@localhost root]# w
```

É melhor fazer um script chamado de `sistema` e colocar estes comandos nele. O conteúdo do arquivo sistema seria o seguinte:

```
#!/bin/bash
date
df
w
```

E para chamar este script, basta agora executar apenas um comando:

```
[root@localhost root]# sistema
```

Isso é um shell script. Um arquivo de texto que contém comandos do sistema e pode ser executado pelo usuário.

Antes de começar

Se você está acessando o sistema como usuário administrador (root), saia e entre como um usuário normal. É **muito perigoso** estudar shell usando o superusuário, você pode danificar o sistema com um comando errado.

Se você não tem certeza qual o seu usuário, use o comando `whoami` para saber.

Como o prompt de usuário normal é diferente para cada um, nos exemplos seguintes será usado `prompt$` para indicar o prompt da linha de comando.

O primeiro shell script

O primeiro shell script a fazer será o `sistema` do exemplo anterior, de simplesmente juntar três comandos em um mesmo script.

Passos para criar um shell script

1. Escolher um nome para o script

Já temos um nome: sistema.

Use apenas letras minúsculas e evite acentos, símbolos e espaço em branco.

2. Escolher o diretório onde colocar o script

Para que o script possa ser executado de qualquer parte do sistema, mova-o para um diretório que esteja no seu PATH. Para ver quais são estes diretórios, use o comando:

```
echo $PATH
```

Se não tiver permissão de mover para um diretório do PATH, deixe-o dentro de seu diretório pessoal ($HOME).

3. Criar o arquivo e colocar nele os comandos

Use o nano, VI ou outro editor de textos de sua preferência para pôr todos os comandos dentro do arquivo.

4. Colocar a chamada do shell na primeira linha

A primeira linha do script deve ser:

```
#!/bin/bash
```

Para que ao ser executado, o sistema saiba que é o shell quem irá interpretar estes comandos.

5. Tornar o script um arquivo executável

Use o seguinte comando para que seu script seja reconhecido pelo sistema como um comando executável:

```
chmod +x sistema
```

Problemas na execução do script

Comando não encontrado

O shell não encontrou o seu script.

Verifique se o comando que você está chamando tem exatamente o mesmo nome do seu script. Lembre-se de que no Unix/Linux as letras maiúsculas e minúsculas são diferentes, então o comando `SISTEMA` é diferente do comando `sistema`.

Caso o nome esteja correto, verifique se ele está no `PATH` do sistema. O

comando `echo $PATH` mostra quais são os diretórios conhecidos, mova seu script para dentro de um deles, ou chame-o passando o caminho completo.

Se o script estiver no diretório corrente, chame-o com um `./` na frente, assim:

```
prompt$ ./sistema
```

Caso contrário, especifique o caminho completo desde o diretório raiz:

```
prompt$ /tmp/scripts/sistema
```

Permissão negada

O shell encontrou seu script, mas ele não é executável.

Use o comando `chmod +x seu-script` para torná-lo um arquivo executável.

Erro de sintaxe

O shell encontrou e executou seu script, porém ele tem erros.

Um script só é executado quando sua sintaxe está 100% correta. Verifique os seus comandos, geralmente o erro é algum `IF` ou aspas que foram abertos e não foram fechados. A própria mensagem informa o número da linha onde o erro foi encontrado.

O primeiro shell script (melhorado)

Nesse ponto, você já sabe o básico necessário para fazer um script em shell do zero e executá-lo. Mas apenas colocar os comandos em um arquivo não torna este script útil. Vamos fazer algumas melhorias nele para que fique mais compreensível.

Melhorar a saída na tela

Executar os três comandos seguidos resulta em um bolo de texto na tela, misturando as informações e dificultando o entendimento. É preciso trabalhar um pouco a saída do script, tornando-a mais legível.

O comando `echo` serve para mostrar mensagens na tela. Que tal

anunciar cada comando antes de executá-lo?

```
#!/bin/bash
echo "Data e Horário:"
date
echo
echo "Uso do disco:"
df
echo
echo "Usuários conectados:"
w
```

Para usar o `echo`, basta colocar o texto entre "aspas". Se nenhum texto for colocado, uma linha em branco é mostrada.

Interagir com o usuário

Para o script ficar mais completo, vamos colocar uma interação mínima com o usuário, pedindo uma confirmação antes de executar os comandos.

```
#!/bin/bash
echo "Vou buscar os dados do sistema. Posso continuar? [sn] "
read RESPOSTA
test "$RESPOSTA" = "n" && exit

echo "Data e Horário:"
date
echo
echo "Uso do disco:"
df
echo
echo "Usuários conectados:"
w
```

O comando `read` leu o que o usuário digitou e guardou na variável `RESPOSTA`. Logo em seguida, o comando `test` verificou se o conteúdo dessa variável era "n". Se afirmativo, o comando `exit` foi chamado e o script foi finalizado. Nessa linha há vários detalhes importantes:

- O conteúdo da variável é acessado colocando-se um cifrão "`$`" na frente.

- O comando `test` é muito útil para fazer vários tipos de verificações em textos e arquivos.
- O operador lógico `&&` só executa o segundo comando caso o primeiro tenha sido OK. O operador inverso é o `||`.

Melhorar o código do script

Com o tempo, o script vai crescer, mais comandos vão ser adicionados e quanto maior, mais difícil encontrar o ponto certo onde fazer a alteração ou corrigir algum erro.

Para poupar horas de estresse, e facilitar as manutenções futuras, é preciso deixar o código visualmente mais agradável e espaçado, e colocar comentários esclarecedores.

```
#!/bin/bash
# sistema - script que mostra informações sobre o sistema
# Autor: Fulano da Silva

# Pede uma confirmação do usuário antes de executar
echo "Vou buscar os dados do sistema. Posso continuar? [sn] "
read RESPOSTA

# Se ele digitou 'n', vamos interromper o script
test "$RESPOSTA" = "n" && exit

# O date mostra a data e a hora correntes
echo "Data e Horário:"
date
echo

# O df mostra as partições e quanto cada uma ocupa no disco
echo "Uso do disco:"
df
echo

# O w mostra os usuários que estão conectados nesta máquina
echo "Usuários conectados:"
w
```

Basta iniciar a linha com um `#` e escrever o texto do comentário em seguida. Estas linhas são ignoradas pelo shell durante a execução. O cabeçalho com informações sobre o script e seu autor também é

importante para ter-se uma visão geral do que o script faz, sem precisar decifrar seu código.

Também é possível colocar comentários no meio da linha, `# como este`

Rebobinando a fita

Agora é hora de fixar alguns dos conceitos vistos no script anterior.

Variáveis

As variáveis são a base de qualquer script. É dentro delas que os dados obtidos durante a execução do script serão armazenados. Para definir uma variável, basta usar o sinal de igual "=" e para ver seu valor, usa-se o `echo`:

```
prompt$ VARIAVEL="um dois tres"
prompt$ echo $VARIAVEL
um dois tres
prompt$ echo $VARIAVEL $VARIAVEL
um dois tres um dois tres
prompt$
```

Não pode haver espaços ao redor do igual.

Ainda é possível armazenar a saída de um comando dentro de uma variável. Em vez de aspas, o comando deve ser colocado entre `$(...)`, veja:

```
prompt$ HOJE=$(date)
prompt$ echo "Hoje é: $HOJE"
Hoje é: Sáb Abr 24 18:40:00 BRT 2004
prompt$ unset HOJE
prompt$ echo $HOJE
prompt$
```

E, finalmente, o comando unset apaga uma variável.

Para ver quais as variáveis que o shell já define por padrão, use o comando `env`.

Detalhes sobre os comandos

Diferente de outras linguagens de programação, o shell não usa os

parênteses para separar o comando de seus argumentos, mas, sim o espaço em branco. O formato de um comando é sempre:

```
COMANDO  OPÇÕES  PARÂMETROS
```

O comando `cat` mostra o conteúdo de um arquivo. O comando `cat -n sistema` mostra o nosso script, com as linhas numeradas. O `-n` é a opção para o comando, que o instrui a numerar linhas, e `sistema` é o último argumento, o nome do arquivo.

O `read` é um comando do próprio shell, já o `date` é um executável do sistema. Dentro de um script, não faz diferença usar um ou outro, pois o shell sabe como executar ambos. Assim, toda a gama de comandos disponíveis no Unix/Linux pode ser usada em scripts!

Há vários comandos que foram feitos para serem usados com o shell, são como ferramentas. Alguns deles:

Comando	Função	Opções úteis
`cat`	Mostra arquivo	`-n`, `-s`
`cut`	Extrai campo	`-d -f`, `-c`
`date`	Mostra data	`-d`, `+'...'`
`find`	Encontra arquivos	`-name`, `-iname`, `-type f`, `-exec`
`grep`	Encontra texto	`-i`, `-v`, `-r`, `-qs`, `-w -x`
`head`	Mostra início	`-n`, `-c`
`printf`	Mostra texto	nenhuma
`rev`	Inverte texto	nenhuma
`sed`	Edita texto	`-n`, `s/isso/aquilo/`, `d`
`seq`	Conta números	`-s`, `-f`
`sort`	Ordena texto	`-n`, `-f`, `-r`, `-k -t`, `-o`
`tail`	Mostra final	`-n`, `-c`, `-f`
`tr`	Transforma texto	`-d`, `-s`, `A-Z a-z`
`uniq`	Remove duplicatas	`-i`, `-d`, `-u`
`wc`	Conta letras	`-c`, `-w`, `-l`, `-L`

Use `man` *`comando`* ou *`comando`* `--help` para obter mais informações sobre cada um deles.

E o melhor, em shell é possível combinar comandos, aplicando-os em

sequência para formar um comando completo. Usando o pipe | é possível canalizar a saída de um comando diretamente para a entrada de outro, fazendo uma cadeia de comandos. Exemplo:

```
prompt$ cat /etc/passwd | grep root | cut -c1-10
root:x:0:0
operator:x
prompt$
```

O `cat` mostra o arquivo todo, o `grep` pega essa saída e extrai apenas as linhas que contêm a palavra root e o `cut`, por sua vez, somente nessas linhas que o `grep` achou, extrai os 10 primeiros caracteres. Isso funciona como uma estação de tratamento de água, onde ela entra suja, vai passando por vários filtros que vão tirando as impurezas e sai limpa no final.

E, por fim, também é possível redirecionar a saída de um comando para um arquivo em vez da tela, usando o operador >. Para guardar a saída do comando anterior no arquivo `saida`, basta fazer:

```
prompt$ cat /etc/passwd | grep root | cut -c1-10 > saida
prompt$ cat saida
root:x:0:0
operator:x
prompt$
```

Cuidado, shell é tão legal que vicia!

O comando test

O canivete suíço dos comandos do shell é o `test`, que consegue fazer vários tipos de testes em números, textos e arquivos. Ele possui várias opções para indicar que tipo de teste será feito, algumas delas:

Testes em variáveis	
Opção	**Descrição**
`-lt`	Número é menor que (LessThan).
`-gt`	Número é maior que (GreaterThan).
`-le`	Número é menor igual (LessEqual).
`-ge`	Número é maior igual (GreaterEqual).

`-eq`	Número é igual (EQual).
`-ne`	Número é diferente (NotEqual).
`=`	String é igual.
`!=`	String é diferente.
`-n`	String é não nula.
`-z`	String é nula.
Testes em arquivos	
Opção	**Descrição**
`-d`	É um diretório.
`-f`	É um arquivo normal.
`-r`	O arquivo tem permissão de leitura.
`-s`	O tamanho do arquivo é maior que zero.
`-w`	O arquivo tem permissão de escrita.
`-nt`	O arquivo é mais recente (NewerThan).
`-ot`	O arquivo é mais antigo (OlderThan).
`-ef`	O arquivo é o mesmo (EqualFile).
`-a`	E lógico (AND).
`-o`	OU lógico (OR).

Tarefa: script que testa arquivos

Tente fazer o script `testa-arquivos`, que pede ao usuário para digitar um arquivo e testa se este arquivo existe. Se sim, diz se é um arquivo ou um diretório. Exemplo de uso:

```
prompt$ testa-arquivos
Digite o arquivo: /naoexiste
O arquivo '/naoexiste' não foi encontrado
prompt$ testa-arquivos
Digite o arquivo: /tmp
/tmp é um diretório
prompt$ testa-arquivos
Digite o arquivo: /etc/passwd
/etc/passwd é um arquivo
prompt$
```

Conceitos mais avançados

Até agora vimos o básico, o necessário para se fazer um script de funcionalidade mínima. A seguir, conceitos novos que ampliarão as fronteiras de seus scripts!

Recebimento de opções e parâmetros

Assim como os comandos do sistema que possuem e opções e parâmetros, os scripts também podem ser preparados para receber dados via linha de comando.

Dentro do script, algumas variáveis especiais são definidas automaticamente, em especial, `$1` contém o primeiro argumento recebido na linha de comando, `$2` o segundo, e assim por diante. Veja o script `argumentos`:

```
#!/bin/sh
# argumentos - mostra o valor das variáveis especiais
echo "O nome deste script é: $0"
echo "Recebidos $# argumentos: $*"
echo "O primeiro argumento recebido foi: $1"
echo "O segundo argumento recebido foi: $2"
```

Ele serve para demonstrar o conteúdo de algumas variáveis especiais, acompanhe:

```
prompt$ ./argumentos um dois três
O nome deste script é: ./argumentos
Recebidos 3 argumentos: um dois três
O primeiro argumento recebido foi: um
O segundo argumento recebido foi: dois
prompt$
```

O acesso é direto, basta referenciar a variável que o valor já estará definido. Assim é possível criar scripts que tenham opções como `--help`, `--version` e outras.

Expressões aritméticas

O shell também sabe fazer contas. A construção usada para indicar uma

expressão aritmética é `$((...))`, com dois parênteses.

```
prompt$ echo $((2*3))
6
prompt$ echo $((2*3-2/2+3))
8
prompt$ NUM=44
prompt$ echo $((NUM*2))
88
prompt$ NUM=$((NUM+1))
prompt$ echo $NUM
45
prompt$
```

If, for e while

Assim como qualquer outra linguagem de programação, o shell também tem estruturas para se fazer condicionais e loop. As mais usadas são `if`, `for` e `while`.

if	for	while

`if COMANDO`
`then`
`  comandos`
`else`
`  comandos`
`fi`
	`for VAR in LISTA`
	`do`
	`  comandos`
	`done`
		`while COMANDO`
		`do`
		`  comandos`
		`done`

Diferente de outras linguagens, o `if` testa um comando e não uma condição. Porém, como já conhecemos qual o comando do shell que testa condições, é só usá-lo em conjunto com o `if`. Por exemplo, para saber se uma variável é maior ou menor do que 10 e mostrar uma mensagem na tela informando:

```
if test "$VARIAVEL" -gt 10
then
    echo "é maior que 10"
else
    echo "é menor que 10"
fi
```

Há um atalho para o `test` , que é o comando `[`. Ambos são exatamente o mesmo comando, porém usar o `[` deixa o `if` mais parecido com o

formato tradicional de outras linguagens:

```
if [ "$VARIAVEL" -gt 10 ]
then
    echo "é maior que 10"
else
    echo "é menor que 10"
fi
```

Se usar o [também é preciso fechá-lo com o] e sempre devem ter espaços ao redor. É recomendado evitar esta sintaxe para diminuir suas chances de erro.

Já o `while` é um laço que é executado enquanto um comando retorna OK. Novamente o `test` é bom de ser usado. Por exemplo, para segurar o processamento do script enquanto um arquivo de lock não é removido:

```
while test -f /tmp/lock
do
    echo "Script travado..."
    sleep 1
done
```

O `for` percorre uma lista de palavras, pegando uma por vez:

```
for numero in um dois três quatro cinco
do
    echo "Contando: $numero"
done
```

Uma ferramenta muito útil para usar com o `for` é o `seq`, que gera uma sequência numérica. Para fazer o loop andar 10 passos, pode-se fazer:

```
for passo in $(seq 10)
```

O mesmo pode ser feito com o `while`, usando um contador:

```
i=0
while test $i -le 10
do
    i=$((i+1))
    echo "Contando: $i"
done
```

E temos ainda o loop infinito, com condicional de saída usando o `break`:

```
while :
```

```
do
    if test -f /tmp/lock
    then
        echo "Aguardando liberação do lock..."
        sleep 1
    else
        break
    fi
done
```

Exercícios

A melhor parte finalmente chegou, agora é a sua vez de se divertir. Seguem alguns exercícios que podem ser resolvidos usando o que foi aprendido até aqui.

Alguns exigirão pesquisa e necessitarão de algumas ferramentas que foram apenas citadas, mas não aprendidas. O shelleiro também tem que aprender a se virar sozinho!

Exercício 1 – relacao.sh

Recebe dois números como parâmetro e mostra a relação entre eles. Exemplo:

```
prompt$ ./relacao.sh 3 5
3 é menor que 5
prompt$ ./relacao.sh 5 3
5 é maior que 3
prompt$ ./relacao.sh 5 5
5 é igual a 5
prompt$
```

Exercício 2 – zerador.sh

Recebe um número como parâmetro e o diminui até chegar a zero, mostrando na tela cada passo, em uma mesma linha. Exemplo:

```
prompt$ ./zerador.sh 5
5 4 3 2 1 0
prompt$ ./zerador.sh 10
10 9 8 7 6 5 4 3 2 1 0
```

```
prompt$
```

Exercício 3 – substring.sh

Recebe duas palavras como parâmetro e verifica se a primeira palavra está contida dentro da segunda. Só mostra mensagem informativa em caso de sucesso, do contrário não mostra nada. Exemplo:

```
prompt$ ./substring.sh ana banana
ana está contida em banana
prompt$ ./substring.sh banana maria
prompt$ ./substring.sh banana
prompt$ ./substring.sh
prompt$
```

Pesquise sobre o comando `grep`.

Exercício 4 – juntatudo.sh

Mostra na tela "grudados" todos os parâmetros recebidos na linha de comando, como uma única palavra. Exemplo:

```
prompt$ ./juntatudo.sh a b c d e f verde azul
abcdefverdeazul
prompt$
```

Pesquise sobre o comando `tr`.

Exercício 5 – users.sh

Do arquivo `/etc/passwd`, mostra o usuário e o nome completo de cada usuário do sistema (campos 1 e 5) separados por um TAB. Exemplo:

```
prompt$ ./users.sh
ftp     FTP User
nobody  Nobody
named   Domain name server
xfs     X Font Server
mysql   MySQL server
aurelio Aurelio Marinho Jargas
prompt$
```

Pesquise sobre o comando `cut`.

Exercício 6 – shells.sh

Do arquivo `/etc/passwd`, mostra todos os shells (último campo) que os usuários usam. Não mostrar linhas repetidas. Exemplo:

```
prompt$ ./shells.sh
/bin/bash
/bin/false
/bin/sync
/sbin/halt
/sbin/shutdown
prompt$
```

Pesquise sobre o comando `uniq`.

Exercício 7 - parametros.sh

Mostra na tela todos os parâmetros recebidos na linha de comando, contando-os. Exemplo:

```
prompt$ ./parametros.sh a b c d e f
Parâmetro 1: a
Parâmetro 2: b
Parâmetro 3: c
Parâmetro 4: d
Parâmetro 5: e
Parâmetro 6: f
prompt$
```

Pesquise sobre o comando `shift`.

Respostas dos exercícios

Resista à tentação de olhar as respostas antes de tentar fazer os exercícios! É na prática que se consolida os conhecimentos aprendidos, não pule este passo. Seja forte, insista, não desista!

testa-arquivos

```
#!/bin/bash

echo -n "Digite o arquivo: "
read ARQUIVO

test -d "$ARQUIVO" && echo "$ARQUIVO é um diretório"
test -f "$ARQUIVO" && echo "$ARQUIVO é um arquivo"
```

```
test -f "$ARQUIVO" -o -d "$ARQUIVO" || echo "O arquivo '$ARQUIVO'
  não foi encontrado"

echo
```

⌨ relacao.sh

```
#!/bin/bash
if test $1 -eq $2
then
    echo "$1 é igual a $2"
elif test $1 -lt $2
then
    echo "$1 é menor que $2"
else
    echo "$1 é maior que $2"
fi
```

⌨ zerador.sh

```
#!/bin/bash
i=$1
while test $i -ge 0
do
    echo -n "$i "
    i=$((i-1))
done
echo
```

⌨ substring.sh

```
#!/bin/bash
test $# -ne 2 && exit
echo $2 | grep -qs $1 && echo "$1 está contida em $2"
```

⌨ juntatudo.sh

```
#!/bin/bash
echo $* | tr -d ' '
```

⌨ users.sh

```
#!/bin/bash
cat /etc/passwd | cut -d : -f 1,5 | tr : '\t'
```

⌨ shells.sh

```
#!/bin/bash
cat /etc/passwd | cut -d : -f 7 | sort | uniq
```

⌨ parametros.sh

```
#!/bin/bash
i=0
while test "$1"
do
    i=$((i+1))
    echo "Parâmetro $i: $1"
    shift
done
```

Apêndice B

Shell no Linux, Mac e Windows

O programa que você fez no Linux vai funcionar no Windows? E no Mac? Como é a compatibilidade entre eles? Como instalar o Bash e as ferramentas no Windows? Como testar um programa no Mac? Como lidar com as quebras de linha do Windows? Onde está o terminal? E o dialog, como fazê-lo funcionar fora do Linux? E a acentuação no Cygwin? As respostas para estas e outras perguntas estão nos parágrafos seguintes. Aproveite!

Shell no Linux

Instalação

O shell Bash faz parte dos pacotes essenciais do Linux, então já vem instalado, você não precisa fazer nada.

Execução

Se você estiver no modo texto (interface somente com caracteres), basta entrar no sistema. Após digitar seu usuário e senha, você já verá o prompt do shell à espera de seus comandos.

Se você estiver no modo gráfico (interface com janelas e botões), procure por ícones ou programas com os seguintes nomes: Console, Konsole, Terminal, Xterm, rxvt.

Este programa pode estar dentro de grupos, como Acessórios ou Ferramentas do Sistema.

Compatibilidade

O Linux não é um Unix. Eles são parentes, mas o Linux vem com as ferramentas GNU, que trazem várias opções adicionais aos comandos básicos do sistema. Se por um lado estas opções novas (e poderosas) facilitam seu trabalho, por outro há uma quebra de compatibilidade com sistemas Unix, cujas ferramentas não dispõem destas opções. Evite utilizá-las.

Shell no Mac

Instalação

A partir do Mac OS X versão 10.3 (Panther), o Bash é o shell padrão de todos os usuários. Não é preciso instalar nada.

Execução

Para acessar o shell, no Finder clique em Aplicativos > Utilitários > Terminal.

Compatibilidade

O Mac OS X é um Unix, derivado do BSD. Isso significa que as suas ferramentas não possuem as opções adicionais dos aplicativos GNU, comuns no Linux. Uma exceção a esta regra é o grep, que no Mac é o GNU grep.

tac

No Mac (e no Unix em geral) não tem tac. Mas o seu comportamento está embutido na opção `-r` do comando `tail`, que também mostra um arquivo de trás para a frente (esta opção não existe no GNU tail). Então basta fazer um alias:

```
alias tac="tail -r"
```

O funcionamento do `tac` é tão simples que é possível simulá-lo com um único comando `sed`. Essa pode ser uma alternativa caso o sistema não possua nem `tac` nem `tail -r`:

```
# Implementação do comando tac no Mac OS X
tac ()
{
        sed '1!G;h;$!d'
}
```

seq

No Mac (e no Unix em geral) não tem `seq`. O comando similar é o `jot`, porém ele possui uma sintaxe diferente, veja:

```
$ jot 5    # mostre 5 números
1
2
3
4
5
$ jot 5 3    # mostre 5 números, inicie no 3
3
```

```
4
5
6
7
$ jot - 3 9 1    # mostre do 3 ao 9, andando de 1 em 1
3
4
5
6
7
8
9
$ jot - 3 9 2       # mostre do 3 ao 9, andando de 2 em 2
3
5
7
9
$ jot - 9 3 -2      # mostre do 9 ao 3, voltando de 2 em 2
9
7
5
3
$ jot -s : - 9 3 -2 # idem anterior, usando : de separador
9:7:5:3
$
```

A ordem dos parâmetros é diferente, e o primeiro argumento é considerado o número total de linhas a serem mostradas, que pode ser omitido colocando-se um hífen em seu lugar.

Mas no geral o funcionamento é o mesmo, inclusive com a opção `-s` para definir o separador. Basta fazer uma função que converta a sintaxe do `seq` para a do `jot` e podemos ter um `seq` no Mac:

```
# Implementação do comando seq no Mac OS X
seq ()
{
    local inicio=1 fim=1 passo=1 sep

    # Foi passado algum separador? -s
    if test "$1" = "-s"
```

```
    then
        sep="$2"
        shift 2
    fi
    # Guarde os números da sequência
    test $# -eq 1 && fim=$1
    test $# -eq 2 && inicio=$1 fim=$2
    test $# -eq 3 && inicio=$1 passo=$2 fim=$3
    # Mostre a sequência
    if test "$sep"
    then
        jot -s $sep - $inicio $fim $passo
    else
        jot - $inicio $fim $passo
    fi
}
```

dialog

No Mac não tem `dialog`. Mas você pode obtê-lo por meio de programas como o Fink, MacPorts e Rudix, que permitem instalar no Mac os programas do Linux. O Rudix, aliás, é um excelente projeto brasileiro cujo diferencial é disponibilizar os programas já prontos, sem necessitar compilação.

Um projeto que também vale a pena conhecer é o CocoaDialog, que usa as janelas padrão (gráficas) do Mac OS X para mostrar suas caixinhas. As opções são um pouco diferentes das utilizadas no `dialog`, mas o uso é muito similar, veja:

```
#!/bin/bash

CD="/Applications/CocoaDialog.app/Contents/MacOS/CocoaDialog"

resposta=$($CD ok-msgbox \
    --text "Título da janela" \
    --informative-text "Texto de dentro da janela" \
    --no-newline --float)
```

```
case "$resposta" in
    1) echo "O usuário apertou o botão OK";;
    2) echo "O usuário apertou o botão Cancel";;
esac
```

Endereços:

- Rudix – http://rudix.org
- Fink – http://finkproject.org
- MacPorts – http://www.macports.org
- CocoaDialog – http://cocoadialog.sourceforge.net

Shell no Windows

Instalação

Para poder usar o Bash e as ferramentas do shell (`grep`, `sed`, `cut` e amigos) no Windows, é preciso instalar o Cygwin. Ele traz um ambiente Linux para dentro de seu Windows. Estes são os passos de instalação:

- Baixe o `setup.exe` e execute-o.
- Escolha Install from Internet.
- Deixe sempre as opções padrão já selecionadas, não mude: `C:\cygwin`, All Users, UNIX.
- Faça a instalação mínima, apertando Avançar na tela da escolha de pacotes, sem selecionar nada.
- Depois que o Cygwin estiver instalado e funcionando, use o Setup novamente para instalar algum programa adicional, caso precise.

Em meu site há um guia completo que ensina a instalar e usar o Cygwin, recomendo a leitura em caso de dúvidas. Outra fonte de informações é a lista de discussão `cygwin-br`, onde você pode conversar com outros usuários do programa, em português.

Endereços:

- setup.exe – http://cygwin.com/setup.exe

- Guia completo de instalação – http://aurelio.net/cygwin/rdl
- Portal nacional do Cygwin – http://aurelio.net/cygwin
- Lista cygwin-br – http://br.groups.yahoo.com/group/cygwin-br/

Execução

Clique no ícone do Cygwin ou acesse o menu Iniciar > Programas > Cygwin > Cygwin Bash Shell.

Compatibilidade

Por ser um ambiente Linux, o Cygwin não é 100% compatível com o Unix. Evite usar as opções que são exclusivas das ferramentas GNU para que seu programa também possa funcionar em sistemas Unix.

Arquivos e diretórios

- O diretório raiz / do Cygwin é a pasta `C:\cygwin` do Windows.
- Use os diretórios `/cygdrive/c` e `/cygdrive/d` para acessar o C:, D: e outros drives.
- Não é preciso usar o `chmod`. Arquivos executáveis são automaticamente detectados pela extensão (`.sh`) ou pela primeira linha mágica `#!/bin/bash`.
- Dentro do Cygwin, todos os diretórios são acessados usando as barras /, como no Linux. Já no Windows, são usadas as barras invertidas \ para separar diretórios. Há um programa chamado `cygpath` que se encarrega de traduzir diretórios de um formato para o outro:

```
$ echo $PWD
/home/aurelio/livro/shell/fontes/
$ cygpath -w $PWD
C:\cygwin\home\aurelio\livro\shell\fontes
```

Editor de textos

Você pode usar qualquer editor de textos do Windows para fazer seus programas, desde o Bloco de Notas (Notepad) ao MS Word. Mas se no

Cygwin aparecerem alguns ^M ou as linhas ficarem grudadas, será preciso usar o comando dos2unix para converter as quebras de linha para o formato do Unix.

Para evitar problemas é recomendado instalar o editor de textos nano no Cygwin, usando o `setup.exe`. É um editor amigável com instruções na tela.

- Use `nano arquivo.sh` para editar um arquivo.
- Use Ctrl+O para salvar e Ctrl+X para sair.
- Crie o seguinte alias em seu `.bashrc` para usar o editor em português:

```
alias nano="LANG=pt_BR nano"
```

Acentuação

Para que os acentos funcionem, é preciso criar um arquivo chamado `.inputrc` dentro do seu diretório (`/home/usuário`) com o seguinte conteúdo:

```
set meta-flag on
set convert-meta off
set output-meta on
```

dialog

No Cygwin não tem `dialog`. Mas é possível baixá-lo do site http://codigolivre.org.br/projects/dialogcyg/. Descompacte o arquivo e execute o programa de instalação que há nele. Exemplo:

```
$ tar -xzf dialog-0.9b.tar.gz
$ cd dialog-0.9b
$ ./install.sh
```

Apêndice C

Análise das Funções ZZ

Se um exemplo vale mais do que mil man pages, então este apêndice sozinho vale mais do que dezenas de livros de shell :) Acompanhe uma análise detalhada do código das Funções ZZ, um programa maduro, focado em compatibilidade e facilidade de uso, que vem sendo lapidado há mais de oito anos! Conheça as técnicas utilizadas, desvendando os segredos de um programa avançado.

Funções ZZ é uma coleção de várias funções para Bash, reunidas em um único arquivo, formando uma biblioteca que pode ser incluída na shell do usuário. Além disso, se essa biblioteca for chamada na linha de comando, age como um programa normal.

Isso a torna uma biblioteca e um programa ao mesmo tempo, que conta com dezenas de miniaplicativos especialistas, prontos para serem usados diretamente na linha de comando.

- Quer saber a cotação atual do dólar? `zzdolar`
- Quer gerar uma senha aleatória de oito letras? `zzsenha 8`
- Quer ver se um CPF é válido? `zzcpf 123.456.789-00`
- Quer saber o resultado da Mega Sena? `zzloteria megasena`
- Quer consultar o Google na linha de comando? `zzgoogle shell avançado`
- Quer calcular...

E assim segue, são cerca de 70 funções diferentes, todas iniciadas por zz (daí o nome), demonstrando todo o poder que um programa em shell pode ter na linha de comando.

Este programa tem uma grande comunidade de usuários e nos seus mais de oito anos de vida já ganhou contribuições de dezenas de pessoas. Seu código pode ser considerado maduro, pois além da longevidade, também funciona em diversas máquinas e sistemas diferentes, cujas correções foram sendo adicionadas ao longo dos anos, ficando cada vez mais compatível e portável.

O que faremos a seguir é analisar trechos do código-fonte deste programa, aprendendo várias técnicas avançadas de uso do shell e suas ferramentas. Aprenderemos também como funcionam na prática alguns dos conceitos ensinados, como código limpo e comentado, chaves, opções de linha de comando, caracteres de controle, expressões regulares, obtenção de dados da Internet, compatibilidade e experiência do usuário. Enfim, é um prato cheio mesmo para os mais famintos :)

Saindo da teoria para encarar o mundo real, aqui vamos nós!

Baixe as funções em www.funcoeszz.net para poder ver o código em seu editor de textos preferido e ainda poder testar cada função na linha de comando.

Cabeçalho informativo

Como já aprendemos, o cabeçalho de um programa é importante para um primeiro contato do leitor com o código-fonte. Perceba como neste cabeçalho estão incluídas as informações de autoria, o site do programa e a sua licença de uso. Logo em seguida começa o histórico de mudanças, registrando o que mudou entre uma versão e outra. Entre parênteses estão os agradecimentos às pessoas que ajudaram.

```
#!/bin/bash
# funcoeszz
#
# INFORMAÇÕES: http://funcoeszz.net
# NASCIMENTO : 22 fevereiro 2000
# AUTORES    : Aurelio Marinho Jargas <verde (a) aurelio net>
#              Thobias Salazar Trevisan <thobias (a) thobias org>
# DESCRIÇÃO  : Funções de uso geral para o shell Bash, que buscam
  informações
#              em arquivos locais e fontes na Internet
# LICENÇA    : GPL
#
# REGISTRO DE MUDANÇAS:
# 20000222 ** 1ª versão
# 20000424 ++ cores, beep, limpalixo, rpmverdono
# 20000504 ++ calcula, jpgi, gifi, trocapalavra, ajuda, echozz,
  forzz
# 20000515 ++ dominiopais, trocaextensao, kill, <> $* > "$@"
#          -- jpgi, gifi: o identify já faz
# 20000517 <> trocapalavra -> basename no $T
# 20000601 -- dicbabel: agora com session &:(
# 20000612 ++ celulartim, trocaarquivo
... removidas centenas de linhas para poupar as árvores ;)
# 20061114 ++ zzfoneletra (valeu rodolfo)
#          <> dicabl: URL e filtro atualizados (valeu sartini)
#          <> dicasl: URL e filtro atualizados
#          <> dicportugues: filtro atualizado (BSD)
#          <> dolar: filtro atualizado (BSD)
```

```
#          <> dominiopais: URL e filtro atualizados
#          <> google: filtro atualizado
#          <> howto: filtro atualizado
#          <> ipinternet: filtro atualizado
#          <> linuxnews: S) filtro atualizado, O) filtro
 atualizado
#          <> locale: URL e filtros atualizados
#          <> loteria: URL e filtro atualizados (valeu casali)
#          <> nomefoto: não sobrescreve arquivo já existente
 (valeu nogaroto)
#          <> noticiaslinux: I) URL atualizada (valeu aires)
#          <> noticiaslinux: U) URL e filtro atualizados
#          <> noticiaslinux: V) filtro atualizado, N) filtro
 atualizado
#          <> pronuncia: agora usa o comando 'say' no mac
#          <> security: URL e filtro atualizados para Conectiva e
 Mandriva
#          <> sigla: filtro atualizado
# 20070717 ++ ZZERIP: regex para casar IPs
#          <> zzzz: adicionada URL do site das funções
#          <> pronuncia: URL atualizada (valeu douglas)
#          <> ipinternet: filtro atualizado (valeu .*)
#          <> dolar: URL e filtro atualizados
#          <> google: filtro atualizado
#          <> whoisbr: URL atualizada
#          <> noticiaslinux: I) removido pois é não é linux-only
#          <> noticiaslinux: Y) URL e filtro atualizados
#          -- dicabl: removida pois agora usam AJAX :(
#          <> dictodos: removido ABL
#          <> arrumanome: opção nova -n (não faz nada, igual
 zznomefoto)
#          ++ checagem de problemas relacionados ao UTF-8 (valeu
 piero)
```

Configuração facilitada

Outro componente importante do início do código é a configuração padrão, que pode ser alterada pelo usuário. Veja como os ###### delimitam visualmente a área de configuração, deixando claro para o usuário onde ele deve alterar. Os comentários são bem amigáveis e

ajudam a entender o que faz cada configuração.

```
#####################################################################
 ############
#
#                                  Configuração
#                                  ------------
#
#
### Configuração via variáveis de ambiente
#
# Algumas variáveis de ambiente podem ser usadas para alterar o
 comportamento
# padrão das funções. Basta defini-las em seu .bashrc ou na
 própria linha de
# comando antes de chamar as funções. São elas:
#
#      $ZZCOR    - Liga/Desliga as mensagens coloridas (1 e 0)
#      $ZZPATH   - Caminho completo para o arquivo das funções
#      $ZZEXTRA  - Caminho completo para o arquivo com funções
 adicionais
#      $ZZTMPDIR - Diretório para armazenar arquivos temporários
#
# Nota: Se você é paranóico com segurança, configure a ZZTMPDIR
 para
#       um diretório dentro do seu HOME.
#
### Configuração fixa neste arquivo (hardcoded)
#
# A configuração também pode ser feita diretamente neste arquivo,
 se você
# puder fazer alterações nele.
#
ZZCOR_DFT=1                        # colorir mensagens? 1 liga, 0
 desliga
ZZPATH_DFT="/usr/bin/funcoeszz"    # rota absoluta deste arquivo
ZZEXTRA_DFT="$HOME/.zzextra"       # rota absoluta do arquivo de
 extras
ZZTMPDIR_DFT="${TMPDIR:-/tmp}"     # diretório temporário
#
#
```

```
##################################################################
 ############
```

Processo de inicialização

Mais um bloco visualmente bem-identificado pelo espaçamento e linhas de comentários. Há o aviso para o usuário não alterar as variáveis, além de lembretes sobre o que representa seu conteúdo (os comentários no fim da linha). O trecho seguinte faz vários testes importantes, que são explicados de uma vez em um bloco de comentários.

```
#
#                                Inicialização
#                                -------------
#
#
# Variáveis auxiliares usadas pelas funções ZZ.
# Não altere nada aqui.
#
#
ZZWWWDUMP='lynx -dump      -nolist -crawl -width=300 -
 accept_all_cookies'
ZZWWWLIST='lynx -dump                     -width=300 -
 accept_all_cookies'
ZZWWWPOST='lynx -post-data -nolist -crawl -width=300 -
 accept_all_cookies'
ZZWWWHTML='lynx -source'
ZZCODIGOCOR='36;1'            # use zzcores para ver os códigos
ZZFAQ='http://funcoeszz.net/download.html#problemas'
ZZERDATA='[0-9][0-9]\/[0-9][0-9]\/[0-9]\{4\}'; # dd/mm/aaa ou
 mm/dd/aaaa
ZZERHORA='[012][0-9]:[0-9][0-9]'
ZZERIP='\([0-9]\{1,3\}\.\)\{3\}[0-9]\{1,3\}' # 0.0.0.0 >
 999.999.999.999
ZZSEDURL='s| |+|g;s|&|%26|g;s|@|%40|g'
#
### Truques para descobrir a localização deste arquivo no sistema
#
# Se a chamada foi pelo executável, o arquivo é o $0.
# Senão, tenta usar a variável de ambiente ZZPATH, definida pelo
```

```
  usuário.
# Caso não exista, usa o local padrão ZZPATH_DFT.
# Finalmente, força que ZZPATH seja uma rota absoluta.
#
[ "${0##*/}" = 'bash' -o "${0#-}" != "$0" ] || ZZPATH="$0"
[ "$ZZPATH" ] || ZZPATH=$ZZPATH_DFT
[ "$ZZPATH" ] || echo 'AVISO: $ZZPATH vazia. zzajuda e zzzz não
  funcionarão'
[ "${ZZPATH#/}" = "$ZZPATH" ] && ZZPATH="$PWD/${ZZPATH#./}"
[ "$ZZEXTRA" ] || ZZEXTRA=$ZZEXTRA_DFT
[ -f "$ZZEXTRA" ] || ZZEXTRA=
#
### Últimos ajustes
#
ZZCOR="${ZZCOR:-$ZZCOR_DFT}"
ZZTMP="${ZZTMPDIR:-$ZZTMPDIR_DFT}/zz"
unset ZZCOR_DFT ZZPATH_DFT ZZEXTRA_DFT ZZTMPDIR_DFT
#
#
################################################################
  ############
```

Perceba que todo o processo de configuração e inicialização apenas define variáveis. É o que vimos no tópico de programar usando chaves. Note no final do bloco como a expansão de variáveis do shell ajuda a definir valores padrão para variáveis vazias.

Terminado este processo de inicialização, agora começam as definições das funções.

zztool – Uma minibiblioteca

Dentro da biblioteca Funções ZZ existe uma minibiblioteca chamada `zztool`, com várias ferramentas auxiliares usadas por várias das funções. Este conceito é útil para centralizar e padronizar o funcionamento das funções. Por exemplo, várias funções precisam verificar se o argumento informado é um número, ou se o arquivo indicado existe, ou apagar brancos do início e final de uma string. Estas tarefas rotineiras estão centralizadas na `zztool`.

O `$1` recebido por esta função é o nome da ferramenta, por exemplo, `testa_numero`. O `$2` é o número que precisa ser verificado. A chamada fica: `zztool testa_numero $meu_numero`. Quem vai fazer esta chamada são outras funções, que precisam ter certeza que o usuário realmente informou um número. As ferramentas que fazem testes apenas retornam verdadeiro/falso e as que manipulam dados mandam o resultado para a saída padrão.

O lado negativo dessa técnica é que as funções que usam estas ferramentas têm como dependência a `zztool`, ou seja, não vão funcionar sem ela. Porém, as vantagens da padronização e centralização de código fazem valer a pena seu uso.

Esta função é praticamente um miniaplicativo com várias ferramentas úteis, que usam e abusam dos recursos do shell para fazer suas espertezas. Recomendo uma estudada com bastante atenção em cada pedaço deste código, que você vai aprender técnicas novas.

```
# ------------------------------------------------------------------
  ------------
# Miniferramentas para auxiliar as funções
# Uso: zztool ferramenta [argumentos]
# ------------------------------------------------------------------
  ------------
zztool ()
{
    case "$1" in
```

A ferramenta `uso` mostra na tela a mensagem de uso de uma função (nome em `$2`), mostrada geralmente quando o usuário esqueceu de algum argumento obrigatório ou usou uma opção inválida. Aqui foi usada uma esperteza. Já existe uma função chamada `zzzz`, que mostra a mensagem de ajuda de cada função. Dentro desta mensagem, lá está a linha que descreve a sintaxe de uso, então basta extraí-la com um `grep`. Simples.

```
        uso)
            # Extrai a mensagem de uso da função $2, usando seu --
  help
            zzzz -h $2 -h | grep Uso
        ;;
```

A ferramenta `eco` usa a técnica que já vimos no capítulo de chaves (flags) para decidir se mostra uma mensagem com cores ou não, dependendo do estado da chave `ZZCOR`. Os caracteres de controle são usados para colorir o texto.

```
        eco)
            shift
            # Mostra mensagem colorida caso $ZZCOR esteja ligada
            if [ "$ZZCOR" != '1' ]
            then
                echo -e "$*"
            else
                echo  -e "\033[${ZZCODIGOCOR}m$*"
                echo -ne "\033[m"
            fi
        ;;
```

A ferramenta `acha` destaca uma palavra em um texto, usando cores. Essa palavra pode ser uma expressão regular também. Mais uma vez a chave `ZZCOR` é consultada para saber se as cores devem ser usadas ou não. Veja na definição da variável `$esc` uma maneira portável de se usar o caractere Esc no `sed`, em vez de colocá-lo literalmente.

```
        acha)
            # Destaca o padrão $2 no texto via STDIN ou $3
            # O padrão pode ser uma regex no formato BRE
  (grep/sed)
            local esc=$(echo -ne '\033')
            local padrao=$(echo "$2" | sed 's,/,\\/,g') # escapa /
            shift 2
            zztool multi_stdin "$@" |
                if [ "$ZZCOR" != '1' ]
                then
                    cat -
                else
                     sed "s/$padrao/$esc[${ZZCODIGOCOR}m&$esc[m/g"
                fi
        ;;
```

A `grep_var` faz um `grep` dentro do conteúdo de uma variável. Poderia ter usado o `echo|grep`, mas apenas com a expansão de variáveis do shell é

possível fazer o mesmo, economizando recursos do sistema. O código, apesar de curto, não é trivial. Prepare-se. Da variável `$3`, é apagado a partir do início (`#`) qualquer coisa (`*`) seguida de `$2`. Lembre-se de que isso é glob, e não expressões regulares!

Isso quer dizer que se `$2` está contida dentro de `$3`, em qualquer posição, ela será apagada. Então o teste compara se este resultado é diferente de `$3` em seu estado original. Se for diferente é porque algo foi apagado, então `$2` realmente estava lá dentro. Se são iguais é porque nada foi alterado então `$3` não contém `$2`. Não entendeu? Leia de novo, e de novo, e de novo :)

```
grep_var)
    # $2 está presente em $3?
    test "${3#*$2}" != "$3"
;;
```

Similar à anterior, a `index_var` também abusa da expansão de variáveis. Ela retorna um número, indicando em qual posição o texto pesquisado aparece na variável. Caso não apareça, retorna zero. Funciona assim: o padrão de pesquisa e tudo o que vem depois dele (`$padrao*`) é removido da variável (`%%`). É então adicionado um ao tamanho da variável que restou, revelando o índice.

```
index_var)
    # $2 está em qual posição em $3?
    local padrao="$2"
    local texto="$3"
    if zztool grep_var "$padrao" "$texto"
    then
        texto="${texto%%$padrao*}"
        echo $((${#texto} + 1))
    else
        echo 0
    fi
;;
```

Aqui duas ferramentas simples que testam a existência e permissão de leitura de um arquivo. A vantagem é fazer o teste e já mostrar a mensagem genérica de erro, poupando linhas nas outras funções que usam esta

ferramenta. O retorno 1 deixa a função que chamou a `zztool` decidir o que fazer no caso de erro.

```
arquivo_vago)
    # Verifica se o nome de arquivo informado está vago
    if test -e "$2"
    then
        echo "Arquivo $2 já existe. Abortando."
        return 1
    fi
;;
arquivo_legivel)
    # Verifica se o arquivo existe e é legível
    if ! test -r "$2"
    then
        echo "Não consegui ler o arquivo $2"
        return 1
    fi
;;
```

As ferramentas `testa_*` fazem uso das expressões regulares para casar os padrões desejados para vários tipos de números. Perceba que foi usado o metacaractere `{1,}` em vez do `+`. Eles são similares, porém o mais não existe em algumas implementações de Unix.

```
testa_numero)
    # Testa se $2 é um número positivo
    echo "$2" | grep -qs '^[0-9]\{1,\}$'
;;
testa_numero_sinal)
    # Testa se $2 é um número (pode ter sinal: -2 +2)
    echo "$2" | grep -qs '^[+-]*[0-9]\{1,\}$'
;;
testa_binario)
    # Testa se $2 é um número binário
    echo "$2" | grep -qs '^[01]\{1,\}$'
;;
testa_ip)
    # Testa se $2 é um número IP (nnn.nnn.nnn.nnn)
    local nnn="\([0-9]\{1,2\}\|1[0-9][0-9]\|2[0-4][0-
```

```
9]\|25[0-5]\)" # 0-255
            echo "$2" | grep -qs "^$nnn\.$nnn\.$nnn\.$nnn$"
        ;;
```

A `multi_stdin` traz uma solução interessante para aceitar texto vindo tanto da entrada padrão quanto dos parâmetros posicionais (`$1`, `$2` etc.). O código é bem simples e fácil de entender: no final tudo é mandado para a saída-padrão.

```
        multi_stdin)
            # Mostra na tela os argumentos *ou* a STDIN, nesta
ordem
            # Útil para funções/comandos aceitarem dados das duas
formas:
            #     echo texto | funcao
            # ou
            #     funcao texto
            shift
            if [ "$1" ]
            then
                     echo "$*"
            else
                    cat -
            fi
        ;;
```

E finalmente chegamos na `trim`, que remove os brancos (espaços e TABs) do início e fim das linhas. A `multi_stdin` que acabamos de ver garante a flexibilidade na chamada dessa função, e um único `sed` com expressões regulares é responsável por arrancar os brancos.

```
        trim)
            shift
            zztool multi_stdin "$@" |
                 sed 's/^[[:blank:]]*// ; s/[[:blank:]]*$//'
        ;;
        esac
}
```

zzajuda – Reaproveitamento dos comentários

A função `zzajuda` gera uma listagem de todas as funções, mostrando a sintaxe de uso e alguns exemplos. O nome da função fica em destaque (colorido) para facilitar a leitura. O resultado estende-se por várias telas, então é paginado.

A grande sacada aqui foi utilizar os próprios comentários que vêm antes do nome de cada função para compor o texto de ajuda. Esta função sabe como extraí-los, pois todos têm o mesmo formato.

Então o próprio arquivo das funções é lido (`ZZPATH`) e formatado. O possível arquivo com funções definidas pelo usuário (`ZZEXTRA`) também entra na brincadeira. Perceba como dessa vez o `cat` foi usado para realmente con**cat**enar dois arquivos. Em seguida, comandos avançados com o `sed` encarregam-se de extrair o texto de ajuda de cada função.

```
# ------------------------------------------------------------------
  ------------
# Mostra uma tela de ajuda com explicação e sintaxe de todas as
  funções
# Uso: zzajuda
# ------------------------------------------------------------------
  ------------
zzajuda ()
{
    zzzz -h ajuda $1 && return
    # Salva a configuração original de cores
    local zzcor_orig=$ZZCOR
    # Desliga cores para os paginadores antigos
    if [ "$PAGER" = 'less' -o "$PAGER" = 'more' ]
    then
        ZZCOR=0
    fi
    # Mostra ajuda das funções padrão e das extras
    cat $ZZPATH $ZZEXTRA |
        # Magia negra para extrair somente os textos de descrição
        sed '
            1 {
                s/.*/** Ajuda das funções ZZ (tecla Q sai)/
                G
                p
```

```
            }
            /^# --*$/,/^# --*$/ {
                s/-\{20,\}/-------/
                s/^# //p
            }
            d' |
        uniq |
        sed "s/^-\{7\}/&&&&&&&&&&/" |
        zztool acha 'zz[a-z0-9]\{2,\}' |
        ${PAGER:-less -r}
```

Para deixar a saída mais fácil de ler, a `zztool acha` dá um destaque ao nome da função, deixando-no colorido caso a chave `ZZCOR` esteja ligada. Então a variável de ambiente `PAGER` é usada para paginar o resultado, tendo o `less -r` como valor padrão caso não esteja definida.

```
    # Restaura configuração de cores
    ZZCOR=$zzcor_orig
}
```

zzzz – Multiuso

A função `zzzz` é multiuso. Dependendo da opção que recebe em `$1`, tem um comportamento diferente. Se chamada sem argumentos, mostra várias informações sobre a biblioteca.

```
# ------------------------------------------------------------------
  ------------
# Mostra informações sobre as funções, como versão e localidade
# Com a opção --atualiza, baixa a versão mais nova das funções
# Com a opção --bashrc, "instala" as funções no ~/.bashrc
# Com a opção --tcshrc, "instala" as funções no ~/.tcshrc
# Uso: zzzz [--atualiza|--bashrc|--tcshrc]
# ------------------------------------------------------------------
  ------------
zzzz ()
{
    local nome_func arg_func padrao
    local info_instalado info_cor versao_local versao_remota
    local arquivo_aliases arquivo_zz extra
    local bashrc="$HOME/.bashrc"
```

```
    local tcshrc="$HOME/.tcshrc"
    local url_site='http://funcoeszz.net'
    local url_exe="$url_site/funcoeszz"
    local instal_msg="Instalacao das Funcoes ZZ
(http://funcoeszz.net)"

    # A versão atual das funções é extraída direto do Changelog
    if test -f "$ZZPATH"
    then
        versao_local=$(
            sed -n '/^$/q;s/^# 200\(.\)\(..\).*/\1.\2/p' $ZZPATH |
            sed '$!d;s/\.0/./'
        )
    fi
```

Mais uma esperteza: a versão atual da biblioteca (`$versao_local`) é obtida diretamente lá do cabeçalho, extraindo o número da entrada mais recente do registro de mudanças. Assim não é preciso fazer uma variável global da biblioteca somente para guardar a sua versão, além de forçar que sempre se registre as últimas mudanças.

Em seguida vem o `case` que decide qual ação deverá ser tomada. Veja o grande bloco de comentários tentando explicar de maneira amigável algo complicado. Isso facilita o entendimento do código tanto para os usuários quanto para os próprios programadores, pois eles também podem esquecer dos detalhes do funcionamento.

```
    case "$1" in
        # Atenção: Prepare-se para viajar um pouco que é meio
complicado :)
        #
        # Todas as funções possuem a opção -h e --help para
mostrar um
        # texto rápido de ajuda. Normalmente cada função teria que
        # implementar o código para verificar se recebeu uma
destas opções
        # e caso sim, mostrar o texto na tela. Para evitar a
repetição de
        # código, estas tarefas estão centralizadas aqui.
        #
        # Chamando a zzzz com a opção -h seguido do nome de uma
função e
```

```
      # seu primeiro parâmetro recebido, o teste é feito e o
texto é
      # mostrado caso necessário.
      #
      # Assim cada função só precisa colocar a seguinte linha no
início:
      #
      #     zzzz -h beep $1 && return
      #
      # Ao ser chamada, a zzzz vai mostrar a ajuda da função
zzbeep caso
      # o valor de $1 seja -h ou --help. Se no $1 estiver
qualquer outra
      # opção da zzbeep ou argumento, nada acontece.
      #
      # Com o "&& return" no final, a função zzbeep pode sair
imediatamente
      # caso a ajuda tenha sido mostrada (retorno zero), ou
continuar seu
      # processamento normal caso contrário (retorno um).
      #
      # Se a zzzz -h for chamada sem nenhum outro argumento, é
porque o
      # usuário quer ver a ajuda da própria zzzz.
      #
      # Nota: Ao invés de "beep" literal, poderíamos usar
$FUNCNAME, mas
      #       o Bash versão 1 não possui essa variável.
      -h | --help)

          nome_func=${2#zz}
          arg_func=$3
          # Nenhum argumento, mostre a ajuda da própria
zzzz
          if ! [ "$nome_func" ]
          then
              nome_func='zz'
              arg_func='-h'
          fi
          # Se o usuário informou a opção de ajuda, mostre o
```

```
texto
          if [ "$arg_func" = '-h' -o "$arg_func" = '--help'  ]
          then
              padrao="Uso: [^ ]*zz$nome_func \{0,1\}"
              # Um xunxo bonito: filtra a saída da zzajuda,
mostrando
              # apenas a função informada.
              zzajuda |
                  grep -C9 "^$padrao\b" |
                  sed -e ':a' -e '$bz' -e 'N;ba' -e ':z' \
                      -e "s/.*\n---*\(\n\)\
(.*$padrao\)/\1\2/;s/\(\n\)---.*/\1/"
              return 0
          else
              # Alarme falso, o argumento não é nem -h nem --
help
              return 1
          fi
      ;;
```

Aqui um detalhe quase imperceptível, mas muito importante. Nas versões anteriores era usada a opção `-z` para mostrar o texto de ajuda. Como o formato mudou, é aconselhável que a opção antiga ainda seja suportada. Basta converter do formato antigo para o novo, assim há a garantia que outras funções não atualizadas continuem funcionando corretamente.

```
      # Garantia de compatibilidade do -h com o formato antigo
(-z):
      # zzzz -z -h zzbeep
      -z)
          zzzz -h $3 $2
      ;;
```

A opção `--atualiza` busca na Internet qual a versão mais recente da biblioteca e compara com a versão local, se forem diferentes, baixa a versão nova. Como este procedimento pode demorar, o usuário é informado com mensagens, a cada passo executado.

```
      # Baixa a versão nova, caso diferente da local
      --atualiza)
```

```
            echo "Procurando a versão nova, aguarde."
            versao_remota=$(
                $ZZWWWDUMP "$url_site" |
                sed -n 's/.*versão atual \([0-9.]\{1,\}\).*/\1/p'
            )
            echo "versão local : $versao_local"
            echo "versão remota: $versao_remota"
            echo

            # Aborta caso não encontrou a versão nova
            [ "$versao_remota" ] || return

            # Compara e faz o download
            if [ "$versao_local" != "$versao_remota" ]
            then
                echo -n 'Baixando a versão nova... '
                $ZZWWWHTML "$url_exe" > "funcoeszz-$versao_remota"
                echo 'PRONTO!'
                echo "Arquivo 'funcoeszz-$versao_remota' baixado,
instale-o manualmente."
                echo "O caminho atual é $ZZPATH"
            else
                echo 'Você já está com a versão mais recente.'
            fi
        ;;

        # Instala as funções no arquivo .bashrc
        --bashrc)

            if ! grep -qs "^[^#]*${ZZPATH:-zzpath_vazia}"
"$bashrc"
            then
                (
                    echo
                    echo "# $instal_msg"
                    echo "source $ZZPATH"
                    echo "export ZZPATH=$ZZPATH"
                ) >> "$bashrc"

                echo 'Feito!'
                echo "As Funções ZZ foram instaladas no $bashrc"
```

```
            else
                echo "Nada a fazer. As Funções ZZ já estão no
$bashrc"
            fi
        ;;

        # Cria aliases para as funções no arquivo .tcshrc
        --tcshrc)
            arquivo_aliases="$HOME/.zzcshrc"

            # Chama o arquivo dos aliases no final do .tcshrc
            if ! grep -qs "^[^#]*$arquivo_aliases"  "$tcshrc"
            then
                (
                    echo
                    echo "# $instal_msg"
                    echo "source $arquivo_aliases"
                    echo "setenv ZZPATH $ZZPATH"
                ) >> "$tcshrc"
                echo 'Feito!'
                echo "As Funções ZZ foram instaladas no $tcshrc"
            else
                echo "Nada a fazer. As Funções ZZ já estão no
$tcshrc"
            fi

            # Cria o arquivo de aliases
            echo > $arquivo_aliases
            for func in $(ZZCOR=0 zzzz | sed '1,/^(( fu/d; /^(/d;
s/,//g')
            do
                echo "alias zz$func 'funcoeszz zz$func'" >>
"$arquivo_aliases"
            done
            echo
            echo "Aliases atualizados no $arquivo_aliases"
        ;;
```

O asterisco do `case` pega o caso de nenhum argumento ser passado na linha de comando, o que significa que o usuário quer ler informações

sobre a biblioteca. Para obter a lista completa de todas as funções, novamente o próprio arquivo é lido e mais um `sed` mágico encarrega-se de extrair apenas os nomes, que então são ordenados e formatados por outro `sed`.

```
        # Mostra informações sobre as funções
        *)
            # As funções estão configuradas para usar cores?
            [ "$ZZCOR" = '1' ] && info_cor='sim' || info_cor='não'

            # As funções estão instaladas no bashrc?
            if grep -qs "^[^#]*${ZZPATH:-zzpath_vazia}" "$bashrc"
            then
                info_instalado="$bashrc"
            else
                info_instalado='não instalado'
            fi

            # Informações, uma por linha
            zztool eco "( local)\c"
            echo " $ZZPATH"
            zztool eco "(versão)\c"
            echo " $versao_local"
            zztool eco "( cores)\c"
            echo " $info_cor"
            zztool eco "(   tmp)\c"
            echo " $ZZTMP"
            zztool eco "(bashrc)\c"
            echo " $info_instalado"
            zztool eco "(extras)\c"
            echo " ${ZZEXTRA:-nenhum}"
            zztool eco "(  site)\c"
            echo " $url_site"

            # Lista de todas as funções
            for arquivo_zz in "$ZZPATH" "$ZZEXTRA"
            do
                if [ "$arquivo_zz" -a -f "$arquivo_zz" ]
                then
```

```
                    echo
                    zztool eco "(( funções disponíveis
  ${extra:+EXTRA }))"
                    # Nota: zzzz --tcshrc procura por " fu"

                    # Sed mágico que extrai e formata os nomes de
  funções
                    # limitando as linhas em 50 colunas
                    sed -n '/^zz\([a-z0-9]\{1,\}\) *(.*/s//\1/p'
  "$arquivo_zz" |
                        sort |
                        sed -e ':a' -e '$b' -e 'N; s/\n/, /; /.\
  {50\}/{p;d;};ba'

                    # Flag tosca para identificar a segunda volta
  do loop
                    extra=1
                fi
            done
        ;;
    esac
}
```

zzminusculas, zzmaiusculas – Um único sed

Veja como converter entre maiúsculas e minúsculas usando o `sed`. O comando `y` faz uma tradução caractere a caractere, trocando um pelo outro. Poderia ter sido usado o `tr` também, o resultado seria o mesmo. O `sed`, porém, sabe como ler um arquivo diretamente sem precisar de `STDIN`, coisa que o `tr` não faz. Perceba que as duas funções são complementares, utilizando o mesmo grupo de caracteres, apenas trocando a ordem.

```
# ------------------------------------------------------------
  ------------
# Conversão de letras entre minúsculas e MAIÚSCULAS, inclusive
  acentuadas
# Uso: zzmaiusculas [arquivo]
#      zzminusculas [arquivo]
# Ex.: zzmaiusculas /etc/passwd
#      echo NÃO ESTOU GRITANDO | zzminusculas
```

```
# ------------------------------------------------------------
  ------------
zzminusculas ()
{
    zzzz -h minusculas $1 && return

    sed '
        y/ABCDEFGHIJKLMNOPQRSTUVWXYZ/abcdefghijklmnopqrstuvwxyz/
         y/ÀÁÂÃÄÅÈÉÊËÌÍÎÏÒÓÔÕÖÙÚÛÜÇÑ/àáâãäåèéêëìíîïòóôõöùúûüçñ/'
  "$@"
}
zzmaiusculas ()
{
    zzzz -h maiusculas $1 && return

    sed '
        y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
         y/àáâãäåèéêëìíîïòóôõöùúûüçñ/ÀÁÂÃÄÅÈÉÊËÌÍÎÏÒÓÔÕÖÙÚÛÜÇÑ/'
  "$@"
}
```

zzuniq – Filtros espertos

O comando `uniq` você já conhece, elimina as linhas repetidas de um arquivo. Mas tem um detalhe importante: as linhas devem ser consecutivas! Para apagar as linhas repetidas mesmo que não sejam consecutivas, entra em cena a `zzuniq`.

Não há algoritmos, apenas uma cadeia de filtros que faz a mágica. A esperteza foi usar o `cat -n` para numerar as linhas em primeiro lugar, assim sabemos qual a ordem original das linhas do arquivo. Depois é usado o `sort | uniq` com opções que ignoram a numeração e agem apenas no conteúdo das linhas. Como o `cat -n` usa um TAB como separador da numeração, não é preciso informar qual o delimitador para o `sort` nem o `uniq`, pois o TAB já é o padrão.

Neste ponto as linhas repetidas foram removidas, porém a ordem original do arquivo foi bagunçada. Entra o `sort -n` para ordenar pela numeração, restaurando as linhas em seus lugares originais e, finalmente,

um `cut` remove os números (novamente o TAB é padrão e não precisa do `-d`).

```
# ------------------------------------------------------------------
  ------------
# Retira as linhas repetidas (consecutivas ou não)
# Útil quando não se pode alterar a ordem original das linhas,
# então o tradicional sort|uniq falha.
# Uso: zzuniq [arquivo]
# Ex.: zzuniq /etc/inittab
#      cat /etc/inittab | zzuniq
# ------------------------------------------------------------------
  ------------
zzuniq ()
{
    zzzz -h uniq $1 && return
    # As linhas do arquivo são numeradas para guardar a ordem
  original
    cat -n "${1:--}" | # Numera as linhas do arquivo
        sort -k2 |     # Ordena ignorando a numeração
        uniq -f1 |     # Remove duplicadas, ignorando a numeração
        sort -n |      # Restaura a ordem original
        cut -f2-       # Remove a numeração

}
```

zzsenha – Firewall e $RANDOM

```
# ------------------------------------------------------------------
  ------------
# Gera uma senha aleatória de N caracteres formada por letras e
  números
# Obs.: A senha gerada não possui caracteres repetidos
# Uso: zzsenha [n]     (padrão n=6)
# Ex.: zzsenha
#      zzsenha 8
# ------------------------------------------------------------------
  ------------
zzsenha ()
{
```

```
    zzzz -h senha $1 && return
    local posicao letra
    local n=6
    local
alpha="abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234
56789"
    local maximo=${#alpha}
    # Guarda o número informado pelo usuário (se existente)
    [ "$1" ] && n=$1
```

No bloco seguinte temos o conceito de firewall. São duas verificações de segurança que, se não satisfeitas, a função para o processamento (`return`). Veja como a `zztool` é útil, testando se o argumento informado pelo usuário é um número. Caso não seja, a mensagem de uso da função é mostrada ao usuário (`zztool` novamente). O segundo teste é do tamanho máximo da senha. Se o número passar por estes dois obstáculos, o processamento continua e a senha será gerada.

```
    # Foi passado um número mesmo?
    if ! zztool testa_numero "$n"
    then
        zztool uso senha
        return
    fi
    # Já que não repete as letras, temos uma limitação de tamanho
    if [ $n -gt $maximo ]
    then
        echo "O tamanho máximo da senha é $maximo"
        return
    fi
```

O esquema de geração da senha já está explicado nos comentários, leia em caso de dúvida. Note que foi usado o `cut -c` para extrair a letra do `$alpha` em vez da expansão de variáveis do próprio Bash: `${alpha:$i:1}`. Este recurso não existia na versão 1 do Bash, por isso foi evitado. Veja também como gerar um número aleatório entre 1 e *N* usando a variável `$RANDOM` e a operação de módulo: `$((RANDOM % N + 1))`.

```
    # Esquema de geração da senha:
    # A cada volta é escolhido um número aleatório que indica uma
    # posição dentro do $alpha. A letra dessa posição é mostrada
```

```
  na
    # tela e removida do $alpha para não ser reutilizada.
    while [ $n -ne 0 ]
    do
        n=$((n-1))
        posicao=$((RANDOM % ${#alpha} + 1))
        letra=$(echo -n "$alpha" | cut -c$posicao)
        alpha=$(echo $alpha | tr -d $letra)
        echo -n $letra
    done
    echo
}
```

zztrocaarquivos – Manipulação de arquivos

Esse código é lindo: curto e fácil de entender. A `zztrocaarquivos` é uma função muito simples que faz uma tarefa igualmente simples: trocar dois arquivos de lugar. Apesar de simples, essa tarefa é chata de fazer "na mão", pois é necessário usar um terceiro arquivo para servir de depósito temporário.

Veja que a criação do arquivo temporário não é a mais segura do mundo, mas é portável e funciona. Se `$ZZTMP` estiver dentro do `HOME` do usuário, melhor ainda. Outra preocupação é manter as permissões originais dos arquivos, por isso foi usado o redirecionamento em vez do `mv`. Os arquivos não são apagados e recriados, apenas seu conteúdo é trocado. Veja também o uso das aspas em todas as variáveis, nunca se esqueça delas!

```
# ----------------------------------------------------------------
  -----
# Troca o conteúdo de dois arquivos, mantendo suas permissões
  originais
# Uso: zztrocaarquivos arquivo1 arquivo2
# Ex.: zztrocaarquivos /etc/fstab.bak /etc/fstab
# ----------------------------------------------------------------
  -----
zztrocaarquivos ()
{
    zzzz -h trocaarquivos $1 && return
```

```
    # Um terceiro arquivo é usado para fazer a troca
    local tmp="$ZZTMP.trocaarquivos.$$"
    # Verificação dos parâmetros
    [ "$#" -eq 2 ] || { zztool uso trocaarquivos; return; }
    # Verifica se os arquivos existem
    zztool arquivo_legivel "$1" || return
    zztool arquivo_legivel "$2" || return
    # A dança das cadeiras
    cat "$2"   > "$tmp"
    cat "$1"   > "$2"
    cat "$tmp" > "$1"

    # E foi
    rm "$tmp"
    echo "Feito: $1 <-> $2"
}
```

zzbyte – Complicada, porém comentada

Esta função é muito útil para todo tipo de conversões entre vários níveis de grandeza quando o assunto é bytes, megas, gigas e teras. Mas fazer os cálculos entre um e outro não é tão trivial e o código acaba ficando difícil de entender. Veja como neste caso os comentários guiam o leitor, informando o que faz cada trecho, facilitando muito o entendimento de como funciona.

```
# ------------------------------------------------------------
  ------
# Conversão entre grandezas de bytes (mega, giga, tera, etc)
# Uso: zzbyte N [unidade-entrada] [unidade-saida]  # BKMGTPEZY
# Ex.: zzbyte 2048               # Quanto é 2048 bytes?  -- 2K
#      zzbyte 2048 K             # Quanto é 2048KB?      -- 2M
#      zzbyte 7 K M              # Quantos megas em 7KB? -- 0.006M
#      zzbyte 7 G B              # Quantos bytes em 7GB? --
  7516192768B
# ------------------------------------------------------------
  -----
zzbyte ()
{
```

```
zzzz -h byte $1 && return
local i i_entrada i_saida diferenca operacao passo falta
local unidades="BKMGTPEZY" # kilo, mega, giga, etc
local n=$1
local entrada=${2:-B}
local saida=${3:-.}

# Sejamos amigáveis com o usuário permitindo minúsculas também
entrada=$(echo $entrada | zzmaiusculas)
saida=$(  echo $saida   | zzmaiusculas)
# Verificações básicas
if ! zztool testa_numero $n
then
    zztool uso byte
    return
fi
if ! zztool grep_var $entrada $unidades
then
    echo "Unidade inválida '$entrada'"
    return
fi
if ! zztool grep_var $saida .$unidades
then
    echo "Unidade inválida '$saida'"
    return
fi

# Extrai os números (índices) das unidades de entrada e saída
i_entrada=$(zztool index_var $entrada $unidades)
i_saida=$(  zztool index_var $saida   $unidades)

# Sem $3, a unidade de saída será otimizada
[ $i_saida -eq 0 ] && i_saida=15

# A diferença entre as unidades guiará os cálculos
diferenca=$((i_saida - i_entrada))
if [ $diferenca -lt 0 ]
then
     operacao='*'
     passo='-'
else
```

```
        operacao='/'
        passo='+'
    fi

    i=$i_entrada
    while [ $i -ne $i_saida ]
    do
        # Saída automática (sem $3)
        # Chegamos em um número menor que 1024, hora de sair
        [ $n -lt 1024 -a $i_saida -eq 15 ] && break

        # Não ultrapasse a unidade máxima (Yota)
        [ $i -eq ${#unidades} -a $passo = '+' ] && break

        # 0 < n < 1024 para unidade crescente, por exemplo: 1 B K
        # É hora de dividir com float e colocar zeros à esquerda
        if [ $n -gt 0 -a $n -lt 1024 -a $passo = '+' ]
        then
            # Quantos dígitos ainda faltam?
            falta=$(( (i_saida - i - 1) * 3))

            # Pulamos direto para a unidade final
            i=$i_saida
            # Cálculo preciso usando o bc (Retorna algo como .090)
            n=$(echo "scale=3; $n / 1024" | bc)
            [ $n = '0' ] && break # 1 / 1024 = 0
            # Completa os zeros que faltam
            [ $falta -gt 0 ] && n=$(printf "%0.${falta}f%s" 0
${n#.})
            # Coloca o zero na frente, caso necessário
            [ "${n#.}" != "$n" ] && n=0$n

            break
        fi
        # Terminadas as exceções, este é o processo normal
        # Aumenta/diminui a unidade e divide/multiplica por 1024
        eval "i=$((i $passo 1))"
        eval "n=$((n $operacao 1024))"
```

```
    done
    # Mostra o resultado
    echo $n$(echo $unidades | cut -c$i)
}
```

zzss – Caracteres de controle

Esta função é um descanso de tela (screen saver) para console, que mostra mensagens coloridas em posições aleatórias. O usuário pode passar um texto qualquer ou pode escolher um dos temas disponíveis, que mostram desenhos simplistas usando arte ASCII. A opção `--fundo` inverte as cores, pintando o fundo em vez da letra e a opção `--rapido`, bem, você sabe, mostra mais rápido :)

Este código é complexo, porém não é preciso entendê-lo totalmente para aprender alguns truques bacanas de dimensões da tela, texto colorido e posicionamento do cursor. Agora você vai ver como é usar os caracteres de controle na vida real.

```
# ------------------------------------------------------------
  -----
# Screen Saver para console, com cores e temas
# Temas: mosaico, espaco, olho, aviao, jacare, alien, rosa, peixe,
  siri
# Obs.: Aperte Ctrl+C para sair
# Uso: zzss [--rapido|--fundo] [--tema <tema>] [texto]
# Ex.: zzss
#      zzss fui ao banheiro
#      zzss --rapido /
#      zzss --fundo --tema peixe
# ------------------------------------------------------------
  -----
zzss ()
{
    zzzz -h ss $1 && return
    local mensagem tamanho_mensagem mensagem_colorida
    local cor_fixo cor_muda negrito codigo_cores fundo
    local linha coluna dimensoes
    local linhas=25
    local colunas=80
```

```
local tema='mosaico'
local pausa=1
```

Lembra que vimos que os arrays só foram implementados na versão 2 do Bash? Por isso, em vez de utilizá-los para guardar os temas, foi usada uma string normal, com um tema por linha. Depois é muito simples extrair o tema desejado simplesmente usando o `grep`.

```
local temas="
    mosaico    #
    espaco    .
    olho    OO
    aviao    --o-O-o--
    jacare    ==*-,,--,,--
       alien    /-=-\\
    rosa    --/--\\-<@
    peixe    >-)))-D
    siri    (_).-=''=-.(_)
"
```

O comando `stty` é um nobre desconhecido até que você precise saber as dimensões da tela do usuário. Chamado com o argumento `size`, ele mostra o número atual de linhas e colunas, separados por um espaço em branco. Com a expansão de variáveis do Bash fica fácil separar cada número em sua própria variável, poupando o uso do `cut`.

```
# Tenta obter as dimensões atuais da tela/janela
dimensoes=$(stty size 2>&-)
if [ "$dimensoes" ]
then
    linhas=${dimensoes% *}
    colunas=${dimensoes#* }
fi
```

Lembra do capítulo de chaves e do outro de opções de linha de comando? Aqui estão ambos os conceitos sendo usados para saber quais as vontades do usuário. Cada opção apenas define uma variável ou uma chave. O processamento só vai começar lá na frente.

```
# Opções de linha de comando
while [ $# -ge 1 ]
do
```

```
    case "$1" in
        --fundo)
            fundo=1
        ;;
        --rapido)
            unset pausa
        ;;
        --tema)
            [ "$2" ] || { zztool uso ss; return; }
            tema=$2
            shift
        ;;
        *)
            mensagem="$*"
            unset tema
            break
        ;;
    esac
    shift
done
# Extrai a mensagem (desenho) do tema escolhido
if [ "$tema" ]
then
    mensagem=$(
        echo "$temas" |
             grep -w "$tema" |
            zztool trim |
            cut -f2
    )
if ! [ "$mensagem" ]
    then
        echo "Tema desconhecido '$tema'"
        return
    fi
fi
```

E as chaves continuam entrando em ação! Agora é a fase da bateria de testes, ajustando os valores de algumas variáveis de acordo com as configurações atuais.

```
  # O 'mosaico' é um tema especial que precisa de ajustes
  if [ "$tema" = 'mosaico' ]
  then
      # Configurações para mostrar retângulos coloridos
frenéticos
      mensagem=' '
      fundo=1
      unset pausa
  fi
  # Define se a parte fixa do código de cores será fundo ou
frente
  if [ "$fundo" ]
  then
       cor_fixo='30;4'
  else
      cor_fixo='40;3'
  fi
```

Finalmente, neste ponto, todas as chaves estão definidas e o processamento vai começar.

```
  # Então vamos começar, primeiro limpando a tela
  clear

  # O 'trap' mapeia o Ctrl-C para sair do Screen Saver
  ( trap "clear;return" 2

  tamanho_mensagem=${#mensagem}

  while :
  do
```

Dentro do loop infinito, temos a condição de parada "trapeada" no Ctrl+C, que limpa a tela e sai da função. A técnica do `$RANDOM` com módulo é usada para ter um número aleatório de linha e de coluna, e um simples `echo` com os caracteres de controle que já conhecemos encarrega-se de fazer o cursor pular para posição calculada.

```
      # Posiciona o cursor em um ponto qualquer (aleatório) da
tela (X,Y)
      # Detalhe: A mensagem sempre cabe inteira na tela
($coluna)
```

```
        linha=$((RANDOM % linhas + 1))
        coluna=$((RANDOM % (colunas - tamanho_mensagem + 1) + 1))
        echo -ne "\033[$linha;${coluna}H"
```

O cursor já pulou para a posição correta, mas ainda falta escolher (aleatoriamente) qual cor será utilizada dessa vez. Ao acaso também é escolhido se a cor vai ser em negrito ou não. Estas variáveis são, então, usadas no `echo` adiante, caso a chave `ZZCOR` esteja ligada. Por último, veja como é fácil fazer um `sleep` de um segundo caso a chave `$pausa` esteja ligada, usando expansão de variáveis.

```
        # Escolhe uma cor aleatória para a mensagem (ou o fundo):
  1 - 7
        cor_muda=$((RANDOM % 7 + 1))
        # Usar negrito ou não também é escolhido ao acaso: 0 - 1
        negrito=$((RANDOM % 2))

        # Podemos usar cores ou não?
        if [ "$ZZCOR" = 1 ]
        then
            codigo_cores="$negrito;$cor_fixo$cor_muda"
            mensagem_colorida="\033[${codigo_cores}m$mensagem\033[
  m"
        else
            mensagem_colorida="$mensagem"
        fi
        # Mostra a mensagem/desenho na tela e (talvez) espera 1s
        echo -ne $mensagem_colorida
        ${pausa:+sleep 1}
    done )
}
```

zzhora – Festa dos builtins

A `zzhora` faz cálculos com horas no formato `HH:MM`. Simples, não? Nem tanto! Prepare-se para viajar. Esta função requer atenção dobrada para não perder os detalhes, pois várias técnicas interessantes foram usadas neste código.

Um detalhe curioso é que o único comando externo utilizado foi o `date`.

Fora ele, a função é feita inteiramente com comandos internos (builtin) do Bash. Cálculos com `$((...))` e expansões de variáveis fazem a festa!

```
# ------------------------------------------------------------
-
# Faz cálculos com horários
# A opção -r torna o cálculo relativo à primeira data, por
exemplo:
#   02:00 - 03:30 = -01:30 (sem -r) e 22:30 (com -r)
# Uso: zzhora [-r] hh:mm [+|- hh:mm]
# Ex.: zzhora 8:30 + 17:25        # preciso somar duas horas!
#      zzhora 12:00 - agora       # quando falta para o almoço?
#      zzhora -12:00 + -5:00      # horas negativas!!!
#      zzhora 1000                # quanto é 1000 minutos?
#      zzhora -r 5:30 - 8:00      # que horas ir dormir para
acordar às 5:30?
#      zzhora -r agora + 57:00    # e daqui 57 horas, será quando?
# ------------------------------------------------------------
-
```

Pelos exemplos do cabeçalho, você já deve ter percebido que esta função é bem poderosa, explorando vários aspectos do tema: calcular horas. A palavra `agora` pode ser usada para indicar a hora atual e é possível somar e subtrair, inclusive horas negativas. A opção `-r` dá a flexibilidade de mudar a interpretação do resultado, baseando os cálculos na primeira hora informada.

```
zzhora ()
{
    zzzz -h hora $1 && return
    local hhmm1 hhmm2 operacao
    local hh1 mm1 hh2 mm2 n1 n2 resultado negativo
    local horas minutos dias horas_do_dia hh mm hh_dia extra
    local relativo=0
    # Opções de linha de comando
    [ "$1" = '-r' ] && {
        relativo=1
        shift
    }

    # Verificação dos parâmetros
```

```
    [ "$1" ] || { zztool uso hora; return; }

    # Dados informados pelo usuário (com valores padrão)
    hhmm1="$1"
    operacao="${2:-+}"
    hhmm2="${3:-00}"
```

A seguir, uma verificação que merece uma explicação mais detalhada. Era necessário saber se `$operação` era igual a "+" ou "-". Um teste duplo ligado com o `-o` (`OR`) poderia ser usado, mas foi escolhido um caminho menos comum. A expansão de variável `#` apaga caracteres do início. Mas quais caracteres? O mais ou o menos, representados pelo glob `[+-]`. Se a expansão realmente removeu um destes caracteres e o conteúdo era somente um deles, a variável ficará vazia. O `test` (camuflado de colchetes) confere se a variável tem conteúdo. Se tiver, é porque existia ali outro caractere que não o mais ou o menos, e não foi apagado. É confuso de entender, mas não desista :)

```
    # Somente adição e subtração são permitidas
    if [ "${operacao#[+-]}" ]
    then
         echo "Operação Inválida: $operacao"
        return
    fi
```

Aqui a utilização do único comando externo, o `date`. Perceba como é fácil permitir a palavra-chave em português ou inglês. É um toque que não precisava, mas é tão rápido de adicionar que nem precisa pensar para fazer. O usuário (gringo) agradece.

```
    # Atalhos bacanas para a hora atual
    [ "$hhmm1" = 'agora' -o "$hhmm1" = 'now' ] && hhmm1=$(date
+%H:%M)
    [ "$hhmm2" = 'agora' -o "$hhmm2" = 'now' ] && hhmm2=$(date
+%H:%M)
```

Agora vem uma sequência de expansões de variáveis que deixaram o `sed` de lado, em um canto escuro. O asterisco é um glob, não confunda com expressões regulares. Estes comandos apagam tudo o que vem antes (ou depois) dos dois-pontos, inclusive. A última bateria de expansões lida com os zeros que sobram do lado esquerdo. É importante removê-los para

que não sejam confundidos com números em octal!

```
# Se as horas não foram informadas, coloca 00
[ "${hhmm1#*:}" = "$hhmm1" ] && hhmm1=00:$hhmm1
[ "${hhmm2#*:}" = "$hhmm2" ] && hhmm2=00:$hhmm2

# Extrai horas e minutos para variáveis separadas
hh1=${hhmm1%:*}
mm1=${hhmm1#*:}
hh2=${hhmm2%:*}
mm2=${hhmm2#*:}

# Retira o zero das horas e minutos menores que 10
hh1=${hh1#0}
mm1=${mm1#0}
hh2=${hh2#0}
mm2=${mm2#0}
```

E, finalmente, os cálculos começam. Dica de ouro para lidar com horas: sempre converta tudo para minutos, faça o que tem que fazer, e no final desconverta.

```
# Os cálculos são feitos utilizando apenas minutos.
# Então é preciso converter as horas:minutos para somente
minutos.
n1=$((hh1*60+mm1))
n2=$((hh2*60+mm2))

# Tudo certo, hora de fazer o cálculo
resultado=$(($n1 $operacao $n2))

# Resultado negativo, seta a flag e remove o sinal de menos "-
"
if [ $resultado -lt 0 ]
then
     negativo=-
    resultado=${resultado#-}
fi

# Agora é preciso converter o resultado para o formato hh:mm
horas=$((resultado/60))
```

```
    minutos=$((resultado%60))
    dias=$((horas/24))
    horas_do_dia=$((horas%24))

    # Restaura o zero dos minutos/horas menores que 10
    hh=$horas
    mm=$minutos
    hh_dia=$horas_do_dia
    [ $hh -le 9 ] && hh=0$hh
    [ $mm -le 9 ] && mm=0$mm
    [ $hh_dia -le 9 ] && hh_dia=0$hh_dia
```

Aqui já temos o resultado do cálculo, porém observe como a saída do programa é diferente, dependendo do modo de operação que o usuário escolheu. O `-r` complica um pouco as coisas, tendo que calcular de quantos dias atrás é a hora que vamos mostrar como resultado.

```
    # Decide como mostrar o resultado para o usuário.
    #
    # Relativo:
    #   $ zzhora -r 10:00 + 48:00
    #   10:00 (2 dias)
    #
    # Normal:
    #   $ zzhora 10:00 + 48:00
    #   58:00 (2d 10h 0m)
    #
    if [ $relativo -eq 1 ]
    then

        # Relativo

        # Somente em resultados negativos o relativo é útil.
        # Para valores positivos não é preciso fazer nada.
        if [ "$negativo" ]
        then
            # Para o resultado negativo é preciso refazer algumas
contas
            minutos=$(( (60-minutos) % 60))
            dias=$((horas/24 + (minutos>0) ))
```

```
            hh_dia=$(( (24 - horas_do_dia - (minutos>0)) % 24))
            mm=$minutos
            # Zeros para dias e minutos menores que 10
            [ $mm -le 9 ] && mm=0$mm
            [ $hh_dia -le 9 ] && hh_dia=0$hh_dia
        fi
```

Pense no usuário! Se é mais bacana ler "ontem" em vez de "-1", por que não dar este presente? É tão fácil de fazer. São algumas linhas a mais no programa, mas, para quem está usando, o impacto visual é outro. Seu programa deixa de intimidar para auxiliar. Poderia ter feito "anteontem" também, mas como o seu par "depois de amanhã" é muito comprido, fica sem.

```
        # "Hoje", "amanhã" e "ontem" são simpáticos no resultado
        case $negativo$dias in
            1)
                 extra='amanhã'
            ;;
            -1)
                 extra='ontem'
            ;;
            0|-0)
                 extra='hoje'
            ;;
            *)
                 extra="$negativo$dias dias"
            ;;
        esac
        echo "$hh_dia:$mm ($extra)"
    else

        # Normal

        echo "$negativo$hh:$mm (${dias}d ${horas_do_dia}h
  ${minutos}m)"
    fi
}
```

zzcpf – Cálculo do CPF

Esta função nasceu para apenas verificar se determinado número de CPF era válido, mas hoje também gera um CPF aleatório. É que como foi necessário implementar todo o cálculo para descobrir quais os dígitos verificadores, ficou fácil calcular estes dígitos para qualquer número, inclusive um gerado aleatoriamente.

O algoritmo está bem-comentado, leia com atenção que você vai entender como funciona. Em resumo: primeiro é extraída (ou gerada aleatoriamente) a base do CPF, que são os nove primeiros dígitos. Os dois últimos números, chamados de dígitos verificadores, são calculados usando estes nove da base.

As regras do cálculo dos dígitos verificadores estão bem-explicadas no bloco enorme de comentários. Não é tão trivial e é fácil confundir-se, por isso os comentários estão bem-detalhados, para ajudar os leitores e também os próprios programadores, que são muito esquecidos :)

```
# ------------------------------------------------------------
  --
# Gera um CPF válido aleatório ou valida um CPF informado
# Obs.: O CPF informado pode estar formatado (pontos e hífen) ou
  não
# Uso: zzcpf [cpf]
# Ex.: zzcpf 123.456.789-09          # valida o CPF
#      zzcpf 12345678909             # com ou sem formatadores
#      zzcpf                         # gera um CPF válido
# ------------------------------------------------------------
  --
zzcpf ()
{
    zzzz -h cpf $1 && return
    local i n somatoria digito1 digito2 cpf base
    # Remove pontuação do CPF informado, deixando apenas números
    cpf="$(echo $* | tr -d -c 0-9)"

    # Extrai os números da base do CPF:
    # Os 9 primeiros, sem os dois dígitos verificadores.
    # Esses dois dígitos serão calculados adiante.
```

```
    if [ "$cpf" ]
    then
        # Faltou ou sobrou algum número...
        if [ ${#cpf} -ne 11 ]
        then
            echo "CPF inválido (deve ter 11 dígitos)"
            return
        fi

        # Apaga os dois último dígitos
        base=${cpf%??}
    else
        # Não foi informado nenhum CPF, vamos gerar um escolhendo
        # nove dígitos aleatoriamente para formar a base
        while [ ${#cpf} -lt 9 ]
        do
            cpf="$cpf$((RANDOM % 9))"
        done
        base=$cpf
    fi
```

Opa, essa linha seguinte merece uma atenção especial. Os dígitos da base estão todos grudados, por exemplo, 123456789. O que o `sed` faz é separá-los, trocando cada caractere por ele mesmo (`&`) seguido de um espaço em branco, ficando 1 2 3 4 5 6 7 8 9. O `set - ` pega esta lista de números e grava como parâmetros posicionais (`$1`, `$2` etc.), tornando fácil sua manipulação, sem precisar de arrays.

```
    # Truque para cada dígito da base ser guardado em $1, $2, $3,
...
    set - $(echo $base | sed 's/./& /g')
    # Explicação do algoritmo de geração/validação do CPF:
    #
    # Os primeiros 9 dígitos são livres, você pode digitar
quaisquer
    # números, não há sequência. O que importa é que os dois
últimos
    # dígitos, chamados verificadores, estejam corretos.
    #
    # Estes dígitos são calculados em cima dos 9 primeiros,
```

```
seguindo
  # a seguinte fórmula:
  #
  # 1) Aplica a multiplicação de cada dígito na máscara de
números
  #    que é de 10 a 2 para o primeiro dígito e de 11 a 3 para o
segundo.
  # 2) Depois tira o módulo de 11 do somatório dos resultados.
  # 3) Diminui isso de 11 e se der 10 ou mais vira zero.
  # 4) Pronto, achou o primeiro dígito verificador.
  #
  # Máscara   : 10    9    8    7    6    5    4    3    2
  # CPF       :  2    2    5    4    3    7    1    0    1
  # Multiplica: 20 + 18 + 40 + 28 + 18 + 35 +  4 +  0 +  2 =
Somatória
  #
  # Para o segundo é praticamente igual, porém muda a máscara
(11 - 3)
  # e ao somatório é adicionado o dígito 1 multiplicado por 2.

  ### Cálculo do dígito verificador 1
  # Passo 1
  somatoria=0
  for i in 10 9 8 7 6 5 4 3 2 # máscara
  do
      # Cada um dos dígitos da base ($n) é multiplicado pelo
      # seu número correspondente da máscara ($i) e adicionado
      # na somatória.
      n=$1
      somatoria=$((somatoria + (i * n)))
      shift
  done
  # Passo 2
  digito1=$((11 - (somatoria % 11)))
  # Passo 3
  [ $digito1 -ge 10 ] && digito1=0

  ### Cálculo do dígito verificador 2
  # Tudo igual ao anterior, primeiro setando $1, $2, $3, etc e
  # depois fazendo os cálculos já explicados.
```

```
    #
    set - $(echo $base | sed 's/./& /g')
    # Passo 1
    somatoria=0
    for i in 11 10 9 8 7 6 5 4 3
    do
        n=$1
        somatoria=$((somatoria + (i * n)))
        shift
    done
    # Passo 1 e meio (o dobro do verificador 1 entra na somatória)
    somatoria=$((somatoria + digito1 * 2))
    # Passo 2
    digito2=$((11 - (somatoria % 11)))
    # Passo 3
    [ $digito2 -ge 10 ] && digito2=0

    # Mostra ou valida
    if [ ${#cpf} -eq 9 ]
    then
        # Esse CPF foi gerado aleatoriamente pela função.
        # Apenas adiciona os dígitos verificadores e mostra na
tela.
        echo $cpf$digito1$digito2 |
             sed 's/\(...\)\(...\)\(...\)/\1.\2.\3-/' #
nnn.nnn.nnn-nn
```

O `sed` da linha anterior é bem bacana, pois formata de uma só vez o CPF inteiro, colocando os pontos e o hífen. Os nove primeiros caracteres foram agrupados de três em três e depois referenciados como \1, \2 e \3.

```
    else
        # Esse CPF foi informado pelo usuário.
        # Compara os verificadores informados com os calculados.
        if [ "${cpf#?????????}" = "$digito1$digito2" ]
        then
            echo CPF válido
        else
            # Boa ação do dia: mostrar quais os verificadores
corretos
            echo "CPF inválido (-$digito1$digito2)"
```

Na linha anterior, mais uma vez pensou-se no usuário. Como já tivemos o trabalho de calcular os dígitos verificadores, por que não mostrá-los na tela caso o CPF informado pelo usuário seja inválido? Assim ele já fica sabendo qual o valor correto imediatamente, sem precisar ficar tentando até acertar. Pequenos detalhes que fazem uma grande diferença. Pense nisso!

```
        fi
    fi
}
```

zzcalculaip – Parece fácil...

O cálculo dos endereços de rede e de broadcast não parece tão complicado... Até que você realmente vá fazê-lo. Essa função lhe poupa este trabalho, fazendo todos os cálculos automaticamente.

Mas para que a manutenção deste código não seja um pesadelo, comentários foram colocados e os nomes das variáveis descrevem o seu conteúdo, facilitando muito o entendimento. Se você gosta de matemática e é administrador de redes, delicie-se com a leitura, acompanhando todos os passos necessários para chegar ao resultado.

```
# ------------------------------------------------------------
  ---
# Calcula o endereço da rede e o endereço de broadcast
# a partir de um ip e uma máscara de rede
# OBS: Se não for especificado a máscara, é assumido a
  255.255.255.0
# Uso: zzcalculaip ip [netmask]
# Ex.: zzcalculaip 127.0.0.1 24
#      zzcalculaip 10.0.0.0/8
#      zzcalculaip 192.168.10.0 255.255.255.240
#      zzcalculaip 10.10.10.0
# ------------------------------------------------------------
  ---
zzcalculaip ()
{
    zzzz -h calculaip $1 && return
    local endereco mascara rede broadcast
```

```
local mascara_binario mascara_decimal mascara_ip
local i ip1 ip2 ip3 ip4 nm1 nm2 nm3 nm4 componente
# Verificação dos parâmetros
[ $# -eq 0 -o $# -gt 2 ] && { zztool uso calculaip; return; }
# Obtém a máscara da rede (netmask)
if zztool grep_var / "$1"
then
    endereco=${1%/*}
    mascara="${1#*/}"
else
    endereco=$1
    mascara=${2:-24}
fi
# Verificações básicas
if ! zztool testa_ip $endereco
then
    echo "IP inválido: $endereco"
    return
fi
if ! (zztool testa_ip $mascara || (
      zztool testa_numero $mascara && test $mascara -le 32))
then
    echo "Máscara inválida: $mascara"
    return
fi
# Guarda os componentes da máscara em $1, $2, ...
# Ou é um ou quatro componentes: 24 ou 255.255.255.0
set - $(echo $mascara | tr . ' ')
# Máscara no formato NN
if [ $# -eq 1 ]
then
    # Converte de decimal para binário
    # Coloca N números 1 grudados '1111111' (N=$1)
    # e completa com zeros à direita até 32, com pontos:
    # $1=12 vira 11111111.11110000.00000000.00000000
    mascara=$(printf "%$1s" 1 | tr ' ' 1)
    mascara=$(
        printf "%-32s" $mascara |
        tr ' ' 0 |
```

```
            sed 's/./&./24 ; s/./&./16 ; s/./&./8'
        )
    fi

    # Conversão de decimal para binário nos componentes do IP e
netmask
    for i in 1 2 3 4
    do
        componente=$(echo $endereco | cut -d'.' -f $i)
        eval ip$i=$(printf "%08d" $(zzconverte db $componente))
        componente=$(echo $mascara | cut -d'.' -f $i)
        if [ "$2" ]
        then
            eval nm$i=$(printf "%08d" $(zzconverte db
$componente))
        else
            eval nm$i=$componente
        fi
    done

    # Uma verificação na máscara depois das conversões
    mascara_binario=$nm1$nm2$nm3$nm4
    if ! (zztool testa_binario $mascara_binario &&
          test ${#mascara_binario} -eq 32)
    then
        echo "Máscara inválida"
        return
    fi

    mascara_decimal=$(echo $mascara_binario | tr -d 0)
    mascara_decimal=${#mascara_decimal}
    mascara_ip="$((2#$nm1)).$((2#$nm2)).$((2#$nm3)).$((2#$nm4))"

    echo "End. IP  : $endereco"
    echo "Mascara  : $mascara_ip = $mascara_decimal"

    rede=$(( ((2#$ip1$ip2$ip3$ip4)) & ((2#$nm1$nm2$nm3$nm4)) ))
    i=$(echo $nm1$nm2$nm3$nm4 | tr 01 10)
    broadcast=$(($rede | ((2#$i)) ))

    # Cálculo do endereço de rede
    endereco=""
    for i in 1 2 3 4
```

```
    do
        ip1=$((rede & 255))
        rede=$((rede >> 8))
        endereco="$ip1.$endereco"
    done
    echo "Rede     : ${endereco%.} / $mascara_decimal"
    # Cálculo do endereço de broadcast
    endereco=""
    for i in 1 2 3 4
    do
        ip1=$((broadcast & 255))
        broadcast=$((broadcast >> 8))
        endereco="$ip1.$endereco"
    done
    echo "Broadcast: ${endereco%.}"
}
```

zzarrumanome – Função ou programa?

Esta é uma das funções preferidas dos usuários. Útil para arrumar nomes de arquivos de música (MP3), que geralmente vêm com muitos espaços em branco e acentuação. Apesar de apenas arrumar nomes, você vai ver que seu código é bem completo, tendo que lidar com recursividade de situações de erro, sempre evitando que o usuário perca algum arquivo no processo. Este é um código que vale a pena estudar cada detalhe, aproveite!

```
# ------------------------------------------------------------
  ---------
# Renomeia arquivos do diretório atual, arrumando nomes estranhos
# Obs.: Ele deixa tudo em minúsculas, retira acentuação e troca
  espaços em
#       branco, símbolos e pontuação pelo sublinhado _
# Opções: -n  apenas mostra o que será feito, não executa
#         -d  também renomeia diretórios
#         -r  funcionamento recursivo (entra nos diretórios)
# Uso: zzarrumanome [-n] [-d] [-r] arquivo(s)
# Ex.: zzarrumanome *
#      zzarrumanome -n -d -r .                  # tire o -n para
```

```
  renomear!
#       zzarrumanome "DOCUMENTO MALÃO!.DOC"       # fica
  documento_malao.doc
#       zzarrumanome "RAMONES - Don't Go.mp3"     # fica ramones-
  dont_go.mp3
# ------------------------------------------------------------------
  ---------
zzarrumanome ()
{
    zzzz -h arrumanome $1 && return
    local arquivo caminho antigo novo recursivo pastas nao i
    # Opções de linha de comando
    while [ "${1#-}" != "$1" ]
    do
        case "$1" in
            -d)    pastas=1    ;;
            -r)    recursivo=1 ;;
            -n)    nao="[-n] " ;;
            *) break        ;;
        esac
        shift
    done
```

Adivinha o que acabamos de ver? Chaves! Pois é, pois é, pois é...

```
    # Verificação dos parâmetros
    [ "$1" ] || { zztool uso arrumanome; return; }

    # Para cada arquivo que o usuário informou...
    for arquivo in "$@"
    do
        # Tira a barra no final do nome da pasta
        [ "$arquivo" != / ] && arquivo=${arquivo%/}

        # Ignora arquivos e pastas não existentes
        [ -f "$arquivo" -o -d "$arquivo" ] || continue
```

Enfim, a recursividade vai aparecer! Acompanhe nas próximas linhas: se o arquivo da vez for na verdade um diretório, a função chama ela mesma novamente, para que todos os arquivos desse diretório sejam arrumados também. Isso, é claro, se a chave `$recursivo` estiver ligada. Veja como a

expansão de variáveis é utilizada para transformar variáveis em opções de linha de comando na chamada da função. Este é o processo inverso do que estamos acostumados!

Outro detalhe importante, mas quase imperceptível, está no fim dessa mesma linha: `"$arquivo"/*`. Veja como as aspas estão ao redor apenas da variável que contém o nome do diretório, mas não do asterisco do glob! Isso tem que ser feito dessa maneira, pois, se o asterisco estiver dentro das aspas, o shell não o considera um glob, e a expansão para "todos os arquivos" não será feita. E as aspas são necessárias no caso de `$arquivo` conter espaços em branco.

```
        # Se for uma pasta...
        if test -d "$arquivo"
        then
            # Arruma arquivos de dentro dela (-r)
            [ "${recursivo:-0}" -eq 1 ] &&
                zzarrumanome -r ${pastas:+-d} ${nao:+-n}
"$arquivo"/*

            # Não renomeia nome da pasta (se não tiver -d)
            [ "${pastas:-0}" -ne 1 ] && continue
        fi
```

De brinde, para os corajosos leitores que conseguiram chegar até aqui (parabéns!), as versões Bash para os comandos `dirname` e `basename`, usando apenas a expansão de variáveis. Prefira esta alternativa para economizar recursos do sistema, evitando chamar os comandos externos.

```
        # A pasta vai ser a corrente ou o 'dirname' do arquivo (se
tiver)
        caminho='.'
        zztool grep_var / "$arquivo" && caminho="${arquivo%/*}"

        # $antigo é o arquivo sem path (basename)
        antigo="${arquivo##*/}"
```

No bloco seguinte está o coração da `zzarrumanome`: o `sed` mágico que remove todos os caracteres estranhos do nome do arquivo. Mas, antes, um detalhe bem especial para nós, programadores shell. Veja que a variável `$antigo` está sendo mostrada sem aspas, para que o shell faça a remoção

de todos os espaços e TABs consecutivos, facilitando o trabalho de limpeza. Essa é uma situação rara, na qual as aspas não devem ser usadas. Mas lembre-se de que isto é uma exceção, e você deve usar aspas sempre.

Em seguida, o nome do arquivo é passado para minúsculas com a ajuda da `zzminusculas` que já conhecemos. Então vem o `sed` com vários comandos que vão aos poucos limpando a sujeira. Leia os comentários para saber o que faz cada um desses comandos.

```
    # $novo é o nome arrumado com a magia negra no Sed
        novo=$(
            echo $antigo | # Sem aspas para aproveitar o 'squeeze'
            zzminusculas |
            sed -e "
                # Remove aspas
                s/[\"']//g

                # Hífens no início do nome são proibidos
                  s/^-/_/

                # Remove acentos
                y/àáâãäåèéêëìíîïòóôõöùúûü/aaaaaaeeeeiiiioooooouuuu/
                y/çñß¢Ð£Øø§µÝý¥¹²³/cnbcdloosuyyy123/

                # Qualquer caractere estranho vira sublinhado
                s/[^a-z0-9._-]/_/g

                # Remove sublinhados consecutivos
                s/__*/_/g

                # Remove sublinhados antes e depois de pontos e
hífens
                s/_\([.-]\)/\1/g
                s/\([.-]\)_/\1/g"
        )
```

O `if` seguinte faz uma verificação muito importante. Devido à mudança gradual de ISO-8859-1 para UTF-8 nos sistemas operacionais, é possível que os caracteres acentuados dentro do `sed` causem problemas. Neste caso, um `return` garante que o processamento vai parar aqui e nenhum

arquivo será renomeado. Isso evita a perda de arquivos, pois, em caso de erro, o nome novo é imprevisível.

```
        # Se der problema com UTF, é o y/// do Sed anterior quem
  estoura
        if [ $? -ne 0 ]
        then
            echo "Ops. Problemas com a codificação dos caracteres"
            return
        fi

        # Nada mudou, então o nome atual já certo
        [ "$antigo" = "$novo" ] && continue
```

O trecho a seguir é pequeno, mas muito útil. Caso já exista um arquivo com o nome novo, a função não vai sobrescrevê-lo. O que ela vai fazer é adicionar um número no final do nome do arquivo, para diferenciá-lo. Se por acaso também já existir um arquivo com o mesmo número no final, então este número vai sendo aumentado até que finalmente apareça um nome de arquivo vago. Assim, se já existir o arquivo `olga.txt`, será tentado `olga.txt.1`, `olga.txt.2`, `olga.txt.3`, ... até aparecer um nome vago.

```
        # Se já existir um arquivo/pasta com este nome, vai
        # colocando um número no final, até o nome ser único.
        if test -e "$caminho/$novo"
        then
            i=1
            while test -e "$caminho/$novo.$i"
            do
                i=$((i+1))
            done
            novo="$novo.$i"
        fi
        # Tudo certo, temos um nome novo e único

        # Mostra o que será feito
        echo "$nao$arquivo -> $caminho/$novo"
```

No fim, o estado chave `$nao` vai decidir se o arquivo será realmente renomeado. Se a chave estiver ligada, é porque o usuário passou a opção -

`n`. Neste caso, ele apenas quer ver como vão ficar os arquivos, para conferir, mas não quer que eles sejam de fato renomeados.

```
        # E faz
        [ "$nao" ] || mv -- "$arquivo" "$caminho/$novo"
    done
}
```

zzipinternet – Dados da Internet

Esta é a primeira função que veremos que busca os dados na Internet e os formata para mostrar ao usuário. Todas as funções seguintes possuem um comportamento similar. O lynx (`$ZZWWWDUMP`) é usado para baixar uma página da Internet e logo em seguida o `sed` é chamado para limpar o resultado, removendo todas as informações desnecessárias.

Esta função não recebe argumentos e tem uma única missão: mostrar qual o seu número IP, caso esteja conectado na Internet. Há um site chamado de *What's My IP* que, quando acessado, mostra o seu número IP. Como ele já faz a maior parte do trabalho, basta então extrair somente o número na página de resultado.

O primeiro `sed` está sendo usado como um `grep`, pois está fechando a saída normal (`-n`), mostrando apenas as linhas que casam com o padrão de pesquisa (`s///p`). Mas como o comando `s` foi usado, no mesmo passo o `sed` já aproveita e formatada tais linhas, removendo o lixo.

O padrão de pesquisa é qualquer coisa (`.*`) seguida de um número IP, seguida de qualquer coisa (`.*`). As "quaisquer coisas" são apagadas, ficando apenas o número. Aquela variável global `ZZERIP` guarda uma expressão regular que casa com um número IP. Como o número aparece duas vezes na página, um `sed 1q` é usado para limitar a saída a apenas uma única linha. Poderia ter sido usado um `head -n 1` aqui também, mas como já tínhamos usado o `sed` no comando anterior, preferimos não ter mais um comando como requisito para rodar esta função.

```
# ------------------------------------------------------------
  ---
# http://www.whatismyip.com
# Mostra o seu número IP (externo) na Internet
```

```
# Uso: zzipinternet
# Ex.: zzipinternet
# ----------------------------------------------------------------
  ---
zzipinternet ()
{
    zzzz -h ipinternet $1 && return
    local url='http://www.whatismyip.com'
    # Faz a consulta e filtra o resultado
    $ZZWWWDUMP "$url" |
        sed -n "s/.* \($ZZERIP\).*/IP: \1/p" |
        sed 1q
}
```

zzramones – Cache local

Esta função baixa um arquivo da Internet que contém todas as frases de todas as letras da banda punk Ramones, e escolhe uma linha ao acaso, mostrando ao usuário. Como são muitas linhas (mais de 3.000), e este arquivo não muda, é feito um esquema de cache local para evitar ter que baixar o arquivo todo a cada vez que o usuário chama a função.

Pode parecer complicado, mas a implementação é muito simples. Primeiro é definido um arquivo temporário `$cache` para guardar a cópia local do arquivo com as frases. Todo procedimento de extração da linha aleatória é feita neste arquivo local. Na primeira vez que esta função for chamada, o arquivo de cache não vai existir, então o arquivo da Internet é baixado. Nas próximas execuções é usado diretamente o arquivo local, sem precisar baixar nada, tendo um resultado instantâneo.

O local padrão da variável `ZZTMP` é o diretório temporário `/tmp`, que em alguns sistemas é limpo a cada reinicialização da máquina. Com isso, a cada reboot o arquivo de cache terá que ser baixado novamente. Mas como não é todo dia que se precisa reiniciar a máquina, isso não é um problema.

```
# ----------------------------------------------------------------
  ---
# http://aurelio.net/doc/misc/ramones.txt
# Procura frases de letras de músicas da banda Ramones
```

```
# Uso: zzramones [palavra]
# Ex.: zzramones punk
#      zzramones
# ------------------------------------------------------------
  ---
zzramones ()
{
    zzzz -h ramones $1 && return

    local url='http://aurelio.net/doc/misc/ramones.txt'
    local cache=$ZZTMP.ramones
    local padrao=$1

    # Se o cache está vazio, baixa listagem da Internet
    if ! test -s "$cache"
    then
        $ZZWWWDUMP "$url" > "$cache"
    fi
    # Mostra uma linha qualquer (com o padrão, se informado)
    zzlinha -t "${padrao:-.}" "$cache"
}
```

zzloteria – Cache temporário em variável

Esta função é a preferida daqueles que sempre fazem sua fezinha. Ela mostra os resultados das principais loterias do país, consultando diretamente o site da Caixa Econômica Federal. Mas o que é importante, mesmo para nós, é conhecer a técnica de guardar o conteúdo baixado dentro de uma variável, fazendo um cache temporário.

```
# ------------------------------------------------------------
  ---
# http://www1.caixa.gov.br/loterias
# Consulta os resultados da quina, megasena, duplasena, lotomania
  e lotofácil
# Uso: zzloteria [quina | megasena | duplasena | lotomania |
  lotofacil]
# Ex.: zzloteria
#      zzloteria quina megasena
# ------------------------------------------------------------
  ---
```

```
zzloteria ()
{
    zzzz -h loteria $1 && return
    local dump numero_concurso resultado acumulado tipo
    local url='http://www1.caixa.gov.br/loterias/loterias'
    local tipos="quina megasena duplasena lotomania lotofacil"

    # O padrão é mostrar todos os tipos, mas o usuário pode
  informar alguns
    [ "$1" ] && tipos=$*
    # Para cada tipo de loteria...
    for tipo in $tipos
    do
        zztool eco $tipo:

        # O resultado da pesquisa (código HTML) é guardado em
  $dump
        # Depois cada dado desejado é extraído localmente
        dump=$($ZZWWWHTML "$url/$tipo/${tipo}_pesquisa.asp")
```

Veja na linha anterior como todo o conteúdo do site foi guardado em uma única variável `$dump`, fazendo um cache temporário. A seguir, as variáveis `$numero_concurso` e `$acumulado` extraem os dados desejados desse cache, mostrando-o com o `echo` (não esquecer das aspas!) e usando expressões regulares avançadas no `sed` para filtrar. Isso poupou a criação de um arquivo temporário.

```
        numero_concurso=$(
            echo "$dump" |
            sed -n "s/^\([0-9][0-9]*\)|.*|\($ZZERDATA\)|.*/\1
  (\2)/p"
        )
        acumulado=$(
            echo "$dump" |
            sed -n '/ACUMUL/{
               s/.*ACUMUL.* \([0-9/][0-9/:
  R$.,Mi]*,00\)\..*/Acumulado para \1/
               s/\.\([0-9]\)00\.000,00/.\1 Mi/ ; p ;}'
        )
```

Mas ainda não acabou. O cache é mostrado novamente para podermos

extrair os números sorteados da loteria em questão. A Lotomania e a LotoFácil são mais complicados, então possuem filtros especiais. As outras loterias caem no asterisco do `case` e usam o filtro genérico. Genérico, mas não necessariamente simples :)

Uma técnica de portabilidade mora aqui também. Perceba que nos dois primeiros itens do `case`, o `sed` adiciona uma arroba `@` no texto, depois um `tr` a troca por uma quebra de linha. Foi feito assim porque o `sed` do Unix não reconhece o `\n` como quebra de linha, então a arroba é usada para representá-lo temporariamente.

```
    # Para obter os números do resultado é mais complicado, varia
        case "$tipo" in
            lotomania)
                resultado=$(
                    echo "$dump" |
                    sed -n 's/.*[>|]|\(\([0-9][0-9]|\)\
{20\}\).*/\1/p' |
                     sed 's/|$// ; s/|/@   /10 ; s/|/ - /g' |
                    tr @ '\n'
                )
            ;;
            lotofacil)
                resultado=$(
                    echo "$dump" |
                    sed -n 's/.*[>|]|\(\([0-9][0-9]|\)\
{15\}\).*/\1/p' |
                    sed 's/|$// ; s/|/@   /10 ; s/|/@   /5 ; s/|/
- /g' |
                    tr @ '\n'
                )
            ;;
            *)
                # Tratamento genérico para a maioria dos tipos
                resultado=$(
                    echo "$dump" |
                    sed -n 's,.*<ul>\(\(<li>[0-9][0-9]*</li>\)*\)
</ul>.*,\1,p' |
                    sed 's,</li><li>, - ,g ; s,</*li>,,g'
                )
```

```
            ;;
        esac
```

Todos os resultados das filtragens foram guardados em variáveis. Até agora nada foi mostrado na tela. A seguir vem o último trecho do programa, que analisa os dados encontrados e decide o que vai mostrar para o usuário.

```
    # Mostra o resultado na tela (caso encontrado algo)
        if [ "$resultado" ]
        then
            echo "   $resultado"
            echo "   Concurso $numero_concurso"
            [ "$acumulado" ] && echo "   $acumulado"
            echo
        fi
    done
}
```

zzgoogle – Quebras de linha no sed

```
# ------------------------------------------------------------------
  ---
# http://google.com
# Retorna apenas os títulos e links do resultado da pesquisa no
  Google
# Uso: zzgoogle [-n <número>] palavra(s)
# Ex.: zzgoogle receita de bolo de abacaxi
#      zzgoogle -n 5 ramones papel higiênico cachorro
# ------------------------------------------------------------------
  ---
zzgoogle ()
{
    zzzz -h google $1 && return
    local padrao
    local limite=10
    local url='http://www.google.com.br/search'
    # Opções de linha de comando
    if [ "$1" = '-n' ]
    then
        limite=$2
```

```
        shift 2
    fi
    # Verificação dos parâmetros
    [ "$1" ] || { zztool uso google; return; }
    # Prepara o texto a ser pesquisado
    padrao=$(echo "$*" | sed "$ZZSEDURL")
    [ "$padrao" ] || return 0
```

Na `zzloteria` vimos uma maneira portável de embutir quebras de linha com o `sed`, usando a arroba como caractere temporário, que depois o `tr` convertia para quebras de linha. Isso funciona porque, no caso da loteria, tínhamos certeza que não aparecia nenhuma arroba no texto original. Mas aqui na `zzgoogle`, que mostra o título do site, pode ter qualquer caractere, é difícil prever. Então temos que usar uma técnica diferente.

Fica feio e quebra todo o alinhamento do código, mas é possível escapar uma quebra de linha na substituição do `sed`, colocando uma contrabarra \ no final da linha. Isso funciona em qualquer `sed`, é a maneira portável de substituir algo por uma quebra de linha. Mas lembre-se de que não podem haver espaços entre a barra e o fim da linha!

```
    # Pesquisa, baixa os resultados e filtra
    $ZZWWWHTML "$url?q=$padrao&num=$limite&ie=ISO-8859-1&hl=pt-BR"
  |
        sed '/<div class=g>/!d ; s/class=g/&\
/g' |
        sed -n 's|^><a href="\([^"]*\)" class=l>\(.*\)</a>
  <table.*|\2\
\1\
|p' |
        sed 's/<[^>]*>//g'
}
```

zznoticiaslinux – Baixar notícias

Lembra do capítulo de extração de dados da Internet, onde baixamos as notícias do site BR-Linux? Aqui está ele, entre outros sites que esta função também busca as últimas manchetes.

```
# ------------------------------------------------------------
  ---
```

```
# http://... - vários
# Busca as últimas notícias sobre linux em páginas nacionais
# Obs.: Cada página tem uma letra identificadora que pode ser
  passada como
#       parâmetro, identificando quais páginas você quer
  pesquisar:
#
#         Y)ahoo Linux          B)r Linux
#         C)ipsga               N)otícias linux
#         V)iva o Linux         U)nder linux
#
# Uso: zznoticiaslinux [sites]
# Ex.: zznoticiaslinux
#      zznoticiaslinux yn
# ------------------------------------------------------------
  ---
zznoticiaslinux ()
{
    zzzz -h noticiaslinux $1 && return
    local url limite
    local n=5
    local sites='byvucin'
    limite="sed ${n}q"

    [ "$1" ] && sites="$1"
```

Os sites desejados são informados por letras que os identificam. Caso o usuário não informe nenhuma, então todos os sites são pesquisados. Veja nas linhas anteriores como é fácil implementar este comportamento, definindo todas as letras como valor padrão de `$sites`, sobrescrevendo com as (possíveis) letras do usuário.

A seguir, em vez de fazer um loop em `$sites` e ir retirando uma por uma as letras que já foram utilizadas, uma técnica mais simples foi usada. É utilizado um `if` para cada letra, que verifica se ela está dentro de `$sites` usando a `zztool grep_var`. Dentro de cada `if` estão registrados o endereço e os filtros necessários para baixar e extrair as notícias daquele site em especial. Assim fica tudo bem separado, e caso um site mude e seu filtro pare de funcionar, os outros sites não são afetados.

```
# Yahoo
if zztool grep_var y $sites
then
    url='http://br.news.yahoo.com/tecnologia/linux'
    echo
    zztool eco "* Yahoo Linux ($url):"
    $ZZWWWHTML "$url" |
        sed -n '
            /topheadline/ {
                n
                s,</a>.*,,
                s/<[^>]*>//gp
            }
            /clearfix/ {
                 n
                s/<[^>]*>//gp
            }' |
        sed 's/^[[:blank:]]*//' |
        $limite
fi

# Viva o Linux
if zztool grep_var v $sites
then
    url='http://www.vivaolinux.com.br'
    echo
    zztool eco "* Viva o Linux ($url):"
    $ZZWWWHTML "$url/index.rdf" |
        sed -n '1,/<item>/d;s@.*<title>\(.*\)</title>@\1@p' |
        $limite
fi

# Cipsga
if zztool grep_var c $sites
then
    url='http://www.cipsga.org.br'
    echo
    zztool eco "* CIPSGA ($url):"
    $ZZWWWHTML "$url" |
```

```
            sed '/^.*<tr><td bgcolor="88ccff">
  <b>/!d;s///;s@</b>.*@@' |
            $limite
    fi
    # Br Linux
    if zztool grep_var b $sites
    then
        url='http://br-linux.org/rss/brlinux-iso.xml'
        echo
        zztool eco "* BR Linux ($url):"
        $ZZWWWDUMP "$url" |
            sed -n '1,/<item>/d;s@.*<title>\(.*\)</title>@\1@p' |
            $limite
    fi

    # UnderLinux
    if zztool grep_var u $sites
    then
        url='http://under-linux.org/feed'
        echo
        zztool eco "* UnderLinux ($url):"
        $ZZWWWDUMP "$url" |
            sed -n '1,/<item>/d;s@.*<title>\(.*\)</title>@\1@p' |
            $limite
    fi

    # Notícias Linux
    if zztool grep_var n $sites
    then
        url='http://www.noticiaslinux.com.br'
        echo
        zztool eco "* Notícias Linux ($url):"
        $ZZWWWHTML "$url" |
            sed -n '/<[hH]3>/{s/<[^>]*>//g;s/^[[:blank:]]*//g;p;}'
  |
            $limite
    fi
}
```

zzdolar – Magia negra com sed

Por mais que se preze por código limpo e comentários esclarecedores, em certas (raras) situações é bem complicado praticá-los. Segue um exemplo de uma função aparentemente simples, que busca a cotação atual do dólar em um site e usa o `sed` para formatar o resultado. São apenas três comandos: `lynx | sed | sed`.

Mas, neste caso, são tantas as pequenas edições no texto baixado que os comentários apenas iriam poluir os comandos. E aqui não tem choro, se você não souber ler expressões regulares, não vão ser os comentários que farão alguma diferença na manutenção.

Mas isto é uma exceção e não deve ser tomado como exemplo. Esta função é considerada *write-only*, pois se o formato da página de consulta for mudado, provavelmente será mais fácil reescrevê-la do zero do que tentar achar qual ponto mudar, ali no meio de tantos comandinhos...

```
# ---------------------------------------------------------------
# http://br.invertia.com
# Busca a cotação do dia do dólar (comercial, paralelo e turismo)
# Obs.: As cotações são atualizadas de 10 em 10 minutos
# Uso: zzdolar
# ---------------------------------------------------------------
zzdolar ()
{
    zzzz -h dolar $1 && return
    $ZZWWWDUMP
  'http://br.invertia.com/mercados/divisas/tiposdolar.aspx' |
        sed '
            # Você acredita que essa sopa de letrinhas funciona?
            # Pois é, eu também não... Mas funciona :)
            s/^ *//
            /Data:/,/Turismo/!d
            /percent/d
            s/  */ /g
            s/.*Data: \(.*\)/\1 compra   venda   hora/
            s|^[1-9]/|0&|
            s@^\([0-9][0-9]\)/\([0-9]/\)@\1/0\2@
            s/^D.lar //
```

```
            s/- Corretora//
            s/ SP//g
            s/ [-+]\{0,1\}[0-9.,]\{1,\}  *%$//
            s/al /& /
            s/lo /&   /
            s/mo /&    /
            s/ \([0-9]\) / \1.000 /
            s/\.[0-9]\>/&0/g
            s/\.[0-9][0-9]\>/&0/g
            /^[^0-9]/s/[0-9] /&  /g' |
        sed '/^Compra/d'
}
```

Funções do usuário (extras)

Agora que todas as funções oficiais foram definidas, é hora de carregar as funções do usuário. Estas são funções desenvolvidas por terceiros, que também podem ser utilizadas aproveitando todo o ambiente das ZZ, com `$ZZWWWDUMP`, `zztool`, `zzzz` e tudo mais.

Estas funções adicionais estão em um outro arquivo, cujo caminho completo está guardado em `$ZZEXTRA`. Se esta variável estiver definida e o arquivo realmente existir, ele é incluído na shell atual (source), mesclando as funções em um único pacote.

Não se deixe enganar pelo tamanho do código. Essa linha faz muito!

```
# --------------------------------------------------
## Incluindo as funções extras
[ "$ZZEXTRA" -a -f "$ZZEXTRA" ] && source "$ZZEXTRA"
```

Chamada pelo executável

Uma característica muito versátil desta biblioteca de funções é que ela também pode ser executada diretamente na linha de comando. É o código seguinte quem faz esta mágica acontecer, acompanhe.

```
# ------------------------------------------------------
## Lidando com a chamada pelo executável
# Se há parâmetros, é porque o usuário está nos chamando
```

```
if [ "$1" ]
then
```

Quando a biblioteca é executada com o comando `source`, ou seja, incluída na shell atual, não há parâmetros na linha de comando, logo não há `$1`. É este teste quem informa se o arquivo foi chamado pelo `source` ou foi executado pelo usuário. E se foi executado, temos algum processamento a fazer.

O primeiro item do `case` trata das opções clássicas `-h` e `--help`. Um `cat` com redirecionamento de entrada (*here document*) deixa limpa a tela de ajuda no código-fonte, sem precisar se preocupar com vários echos ou aspas. Note que foi usado o operador `<<-`, é este hífen quem diz que os TABs do início das linhas seguintes devem ser ignorados. Assim o código pode continuar bem-alinhado, não influindo na formatação da mensagem na tela.

```
    case "$1" in

        # Mostra a tela de ajuda
        -h | --help)

            cat - <<-FIM
                Uso: funcoeszz <função> [<parâmetros>]
                     funcoeszz <função> --help
                Dica: Inclua as funções ZZ no seu login shell,
                      e depois chame-as diretamente pelo nome:
                    prompt$ funcoeszz zzzz --bashrc
                    prompt$ source ~/.bashrc
                    prompt$ zz<TAB><TAB>
                    Obs.: funcoeszz zzzz --tcshrc também funciona
                Lista das funções:
                    prompt$ funcoeszz zzzz
            FIM
        ;;
```

Mais uma opção clássica, desta vez é o `--version`. O echo `-n` mostra o texto sem quebrar a linha no final, e o número da versão é extraído automaticamente da saída da função `zzzz`. Mais uma vez o `sed` é usado como `grep` para pescar a linha desejada e então formatá-la, sobrando

apenas os números.

```
# Mostra a versão das funções
-v | --version)
    echo -n 'Funções ZZ v'
    zzzz | sed '/versã/!d ; s/.* //'
;;
```

Se não era `--help` nem `--version`, então é porque o usuário passou o nome de uma função como argumento, por exemplo: `./funcoeszz dolar`. O trecho de código seguinte encarrega-se de executar a `zzdolar` neste caso.

Primeiro, é guardado o nome completo da função em `$func`. Uma esperteza é usada para garantir a flexibilidade do usuário digitar ou não o prefixo `zz`. Assim tanto faz chamar `funcoeszz zzdolar` ou `funcoeszz dolar`. A expansão de variáveis retira o prefixo `zz` do início (#), caso ele exista, deixando apenas o nome da função (`dolar`). Depois o prefixo `zz` é sempre colocado no início para completar o nome da função.

O comando `type` é um builtin do Bash que mostra o código de uma função carregada em memória. Como já definimos todas as funções anteriormente, neste ponto já estão todas carregadas em memória. Se o comando `type` falhar é porque o usuário digitou um nome inválido de função, pois ela não foi encontrada.

Se o `type` funcionou, então basta chamar a função. O `shift` é usado para remover o nome da função (`$1`) da lista de parâmetros, restando apenas seus argumentos em `$@`. Por exemplo, se o usuário chamou: `funcoeszz hora 12:00 + 02:00`, depois do `shift` o `$@` vai conter `12:00 + 02:00`, que são os argumentos que a função (`$func`) espera receber.

```
# Chama a função informada, caso ela exista
*)
    func="zz${1#zz}" # o prefixo zz é opcional

    if type $func >&- 2>&- # a função existe?
    then
        shift
        $func "$@"
    else
        echo "Função inexistente '$func' (tente --help)"
```

```
            fi
        ;;
    esac
fi
```

E fim!

Apêndice D

Caixa de ferramentas

O programador shell deve ter uma visão geral de quais são as ferramentas disponíveis no sistema, para quando precisar, saber por onde começar a procurar. Além desse conhecimento geral, o programador deve ser especialista nas ferramentas básicas, aquelas que independentemente da área de atuação estão sempre presentes nos programas. Conheça estas ferramentas essenciais e suas principais opções.

Assim como um bom mecânico, o programador shell deve ter uma caixa de ferramentas recheada, que o acompanha em toda a jornada de trabalho. É muito importante conhecer os detalhes de cada ferramenta, sabendo como usar, quando usar e também quando evitar.

Como na oficina, a escolha da ferramenta certa pode ser a diferença entre um trabalho tranquilo e rápido, ou um festival de gambiarras que leva horas para ficar pronto e que no final nem o próprio autor vai ficar orgulhoso do serviço.

- Já tentou soltar uma porca com um alicate de bico?
- Ou apertar um parafuso com uma faca de cozinha? De ponta?
- Ou martelar um prego com uma pedra?
- Ou medir paredes com régua escolar de 30cm?
- Ou...

É possível? Sim. Mas só quem já enfrentou essas situações de precariedade sabe o sacrifício que é fazer algo simples quando não se tem a ferramenta correta em mãos.

Em shell é exatamente o mesmo. É primordial conhecer todas as ferramentas para saber como e quando utilizá-las. Só tem um detalhe. Em vez de uma caixa de ferramentas, o shell é como se fosse uma loja de materiais de construção, com todos os tipos de ferramentas, materiais e acessórios. É um mundo de comandos, opções e parâmetros. Para todo tipo de tarefa há um programa especialista, ou a opção obscura de um programa mais genérico.

Cabe a cada profissional avaliar quais são as suas necessidades e compor a sua própria caixa de ferramentas. De nada adianta uma talhadeira a um encanador, assim como fita veda-rosca não tem nada a ver com marcenaria. Tanto o marceneiro quanto o encanador têm suas caixas de ferramentas, mas o conteúdo de ambas é diferente. Da mesma maneira, os programadores shell também seguem rumos variados, especializando-se em áreas distintas e sendo conhecedores de ferramentas diferentes.

O programador shell deve sobretudo ter uma visão geral de quais são as

ferramentas disponíveis, para quando precisar, saber por onde começar a procurar. Além desse conhecimento geral, o programador deve ser especialista nas ferramentas básicas, aquelas que independente da área estão sempre presentes nos programas.

Será isso o que veremos a seguir: as ferramentas básicas e suas opções mais comuns. É o suprassumo do ambiente shell, os comandos e opções que você deve saber de cabeça e que formam uma base sólida para a resolução de problemas variados. Não adianta querer conhecer outras ferramentas sem ter o domínio absoluto destas. Então dedique-se no estudo.

Caixa de ferramentas básica

Comando	Função	Opções úteis
`cat`	Mostra arquivo	`-n`, `-s`
`cut`	Extrai campo	`-d -f`, `-c`
`date`	Mostra data	`-d`, `+'...'`
`diff`	Compara arquivos	`-u`, `-Nr`, `-i`, `-w`
`echo`	Mostra texto	`-e`, `-n`
`find`	Encontra arquivos	`-name`, `-iname`, `-type f`, `-exec`, `-or`
`fmt`	Formata parágrafo	`-w`, `-u`
`grep`	Encontra texto	`-i`, `-v`, `-r`, `-qs`, `-n`, `-l`, `-w -x`, `-A -B -C`
`head`	Mostra início	`-n`, `-c`
`od`	Mostra caracteres	`-a`, `-c`, `-o`, `-x`
`paste`	Paraleliza arquivos	`-d`, `-s`
`printf`	Mostra texto	nenhuma
`rev`	Inverte texto	nenhuma
`sed`	Edita texto	`-n`, `-f`, `s/isso/aquilo/`, `p`, `d`, `q`, `N`
`seq`	Conta números	`-s`, `-f`
`sort`	Ordena texto	`-n`, `-f`, `-r`, `-k -t`, `-o`
`tac`	Inverte arquivo	nenhuma
`tail`	Mostra final	`-n`, `-c`, `-f`
`tee`	Arquiva fluxo	`-a`
`tr`	Transforma texto	`-d`, `-s`, `A-Z a-z`

`uniq`	Remove duplicatas	`-i`, `-d`, `-u`
`wc`	Conta letras	`-c`, `-w`, `-l`, `-L`
`xargs`	Gerencia argumentos	`-n`, `-i`

Para ver outras opções destas ferramentas, use a opção `-h` ou `--help` e leia a man page (`man` *`nome-do-comando`*). Para conhecer quais são as outras ferramentas disponíveis em seu sistema, liste o conteúdo do `/bin` e `/usr/bin`, além de outros diretórios que estejam em sua variável `$PATH`.

Vários dos exemplos seguintes utilizam um mesmo arquivo de texto chamado de `numeros.txt` para demonstrar o funcionamento dos comandos. Este é seu conteúdo:

```
um
dois
três
quatro
cinco
```

cat

Mostra o conteúdo de um ou mais arquivos.

Opções:

Opção	Lembrete	Descrição
`-n`	Number	Numera as linhas (Formato: Espaços, Número, TAB, Linha).
`-s`	Squeeze	Remove as linhas em branco excedentes.

Exemplos:

```
$ echo -e "um\ndois\n\n\n\ntrês"
um
dois


três
$ echo -e "um\ndois\n\n\n\ntrês" | cat -n -
     1  um
     2  dois
     3
     4
     5
```

```
     6  três
$ echo -e "um\ndois\n\n\ntrês" | cat -n -s -
     1  um
     2  dois
     3
     4  três
$
```

Dicas:

- Útil para emendar dois arquivos:

```
$ cat arquivo1.txt arquivo2.txt > arquivao.txt
```

- Geralmente `cat` *`arquivo`* `|` *`comando`* pode ser escrito diretamente como *`comando arquivo`* ou *`comando`* `<` *`arquivo`*.
- Em algumas versões do `cat`, as opções `-n` e `-s` não estão disponíveis.

cut

Extrai campos ou trechos de uma linha. Possui um formato bem flexível de especificação de campos e caracteres.

Opções:

Opção	Lembrete	Descrição
`-d`	Delimiter	Escolhe o delimitador (o padrão é o TAB).
`-f`	Field	Mostra estes campos (veja tabela seguinte).
`-c`	Chars	Mostra estes caracteres (veja tabela seguinte).

`-f / -c`	Abrange	Significa
2,5	2 5	O segundo e o quinto.
2-5	2 3 4 5	Do segundo ao quinto.
2-	2 3 4 5 ...	Do segundo em diante.
-5	1 2 3 4 5	Até o quinto.
2,5-	2 5 6 7 ...	O segundo e do quinto em diante.
2,3,5-8	2 3 5 6 7 8	O segundo, o terceiro e do quinto ao oitavo.

Exemplos:

```
$ echo 'um:dois:três:quatro:cinco' | cut -d : -f 2,3,4
```

```
dois:três:quatro

$ echo 'um:dois:três:quatro:cinco' | cut -d : -f 2-4
dois:três:quatro

$ echo 'um:dois:três:quatro:cinco' | cut -d : -f 2-
dois:três:quatro:cinco
$ echo 'um:dois:três:quatro:cinco' | cut -c 1-10
um:dois:tr
$ echo 'um:dois:três:quatro:cinco' | cut -c 4-9
dois:t
```

Dicas:

- Útil para extrair campos de arquivos como o `/etc/passwd` e o `/etc/fstab`

```
$ cat /etc/fstab | tr -s ' ' | cut -d ' ' -f 2
swap
/
/mnt/cdrom
/mnt/floppy
/proc
/dev/pts
```

- Combine o `cut` com o `rev` para obter os últimos caracteres ou campos

```
$ echo 'um:dois:três:quatro:cinco' | rev | cut -d : -f 1 | rev
cinco
```

date

Mostra uma data. Possui várias opções para escolher o formato dessa data. Caso não informada a data desejada, usa a atual.

Opções:

Opção	Lembrete	Descrição
`-d`	Date	Especifica a data (Ex.: `tomorrow`, `2 days ago`, `5 weeks`).
`+%?`		Formato da data – veja tabela seguinte (Ex.: `%Y-%m-%d`).

Formato	Descrição do caractere de formatação
`%a`	Nome do dia da semana abreviado (Dom..Sáb).

%A	Nome do dia da semana (Domingo..Sábado).
%b	Nome do mês abreviado (Jan..Dez).
%B	Nome do mês (Janeiro..Dezembro).
%c	Data completa (Sat Nov 04 12:02:33 EST 1989).
%y	Ano (dois dígitos).
%Y	Ano (quatro dígitos).
%m	Mês (01..12).
%d	Dia (01..31).
%j	Dia do ano (001..366).
%H	Horas (00..23).
%M	Minutos (00..59).
%S	Segundos (00..60).
%s	Segundos desde 1º de Janeiro de 1970.
%%	Um % literal.
%t	Um TAB.
%n	Uma quebra de linha.

Exemplos:

```
$ date
Sun Jul 20 20:29:57  2003

$ date -d yesterday
Sat Jul 19 20:30:02  2003

$ date -d '4 ºweeks ago'
Sun Jun 22 20:30:09  2003

$ date -d '123 days' +'Daqui 123 dias será %d de %b.'
Daqui 123 dias será 20 de Nov.
```

Dicas:

■ O formato `%Y-%m-%d` é bom para nomear becapes diários:

```
$ date +'becape-%Y-%m-%d.tgz'
becape-2003-07-20.tgz
```

■ O formatador `%s` é útil para fazer contas com datas:

```
$ date +%s ; sleep 10 ; date +%s
1058733386
1058733396
$ expr 1058733396 - 1058733386
10
```

■ Em algumas versão do `date`, a opção `-d` não está disponível.

diff

Mostra as diferenças entre dois arquivos.

Opções:

Opção	Lembrete	Descrição
`-u`	Unified	Formato unificado (com contexto e os sinais de + e -).
`-C`	Context	Indica a quantidade de linhas usadas para o contexto.
`-r`	Recusive	Varre todo o diretório.
`-N`	New file	Considera arquivos não encontrados como vazios.
`-i`	Ignore case	Ignora a diferença entre maiúsculas e minúsculas.
`-w`	White space	Ignora a diferença de linhas e espaços em branco.

Exemplos:

```
$ echo -e 'um\ndois\ntrês\nquatro' > arq1
$ echo -e 'um\ntrês\nquatro\ncinco' > arq2
$ diff -u arq1 arq2
--- arq1        Sun Jul 20 20:56:28 2003
+++ arq2        Sun Jul 20 20:57:36 2003
@@ -1,4 +1,4 @@
 um
-dois
 três
 quatro
+cinco
$ echo -e 'um\ndoisss\ntrêsss\nquatro' > arq3
$ diff -u arq1 arq3
--- arq1        Sun Jul 20 20:56:28 2003
+++ arq3        Sun Jul 20 20:57:56 2003
```

```
@@ -1,4 +1,4 @@
 um
-dois
-três
+doisss
+trêsss
 quatro
$
```

Dicas:

- Use as opções `-r` e `-u` para comparar dois diretórios

```
$ diff -r -u dir1 dir2
```

- Passe um número para a `-C` para definir quantas linhas de contexto:

```
$ diff -u -C 0 arq1 arq2
--- arq1        Sun Jul 20 20:56:28 2003
+++ arq2        Sun Jul 20 20:57:36 2003
@@ -2 +1,0 @@
-dois
@@ -4,0 +4 @@
+cinco
```

- O resultado é um arquivo pronto para ser usado pelo comando `patch`:

```
$ diff -u arq2 arq3 > alteracoes.patch
$ patch arq2 < alteracoes.patch
patching file `arq2'
$ diff arq2 arq3
$
```

- Para ver a saída do `diff` com cores, use o `vi`:

```
$ diff -u arq1 arq2 | vi -
```

- Em algumas versões do `diff`, as opções `-i`, `-N` e `-u` não estão disponíveis.
- Em algumas versões do `diff`, a opção `-w` não está disponível. Use a opção `-b` que é parecida.

echo

Mostra um texto. Útil para ecoar mensagens na tela.

Opções:

Opção	Lembrete	Descrição
`-n`	Newline	Não quebra a linha no final.
`-e`	Escape	Interpreta os escapes especiais (ver tabela seguinte).

Escape	Lembrete	Descrição
`\a`	Alert	Alerta (bipe).
`\b`	Backspace	Caractere Backspace.
`\c`	EOS	Termina a string.
`\e`	Escape	Caractere Esc.
`\f`	Form feed	Alimentação.
`\n`	Newline	Linha nova.
`\r`	Return	Retorno de carro.
`\t`	Tab	Tabulação horizontal.
`\v`	Vtab	Tabulação vertical.
`\\`	Backslash	Barra invertida \ literal.
`\`*nnn*	Octal	Caractere cujo octal é *nnn*.
`\x`*nn*	Hexa	Caractere cujo hexadecimal é *nn*.

Exemplos:

```
$ echo -e '0\t1\t2\t3\t4\t5\t6\t7'
0       1       2       3       4       5       6       7

$ echo -e 'Bip!\a Bip!\a Bip!\a'
Bip! Bip! Bip!

$ echo -e '\x65\x63\x68\x6F\x21'
echo!
```

Dicas:

- Para maior controle de alinhamento e formatação, use o `printf`.
- Com um pouco de criatividade, usando os escapes `\b` e `\r` pode-se fazer pequenas animações no console, como uma barra de progresso

que mostra a porcentagem:

```
$ echo -n '[ 33%]'; sleep 1 ; echo -ne '\r[ 66' ; sleep 1 ;
echo -e '\b\b\b100'
[100%]
```

- Ou a famosa hélice ASCII:

```
while :; do for a in / - \\ \|; do echo -ne "\b$a"; done; done
```

- Em algumas versões do `echo`, a opção `-e` não está disponível. Em outras, o `-e` é implícito, pois os escapes são expandidos por padrão.

find

Encontra arquivos procurando pelo nome, data, tamanho e outras propriedades.

Opções:

Opção	Descrição
`-name`	Especifica o nome do arquivo (ou *parte* dele).
`-iname`	Ignora a diferença entre maiúsculas e minúsculas no nome.
`-type`	Especifica o tipo do arquivo (`f`=arquivo, `d`=diretório, `l`=link).
`-mtime`	Mostra os arquivos modificados há N dias.
`-size`	Mostra os arquivos que possuem o tamanho especificado.
`-user`	Mostra os arquivos de um usuário específico.
`-ls`	Mostra os arquivos no mesmo formato do comando `ls`.
`-printf`	Formatação avançada para mostrar os nomes dos arquivos.
`-exec`	Executa um comando com os arquivos encontrados.
`-ok`	Executa um comando com os arquivos encontrados, com confirmação.
`-and, -or`	E, OU lógico para as condições.
`-not`	Inverte a lógica da expressão.

Detalhes das opções `-exec` e `-ok`

A string `{}` representa o nome do arquivo encontrado

O comando deve ser passado sem aspas

O comando deve ser terminado por um ponto e vírgula escapado \;

Tem que ter um espaço antes do ponto e vírgula escapado

Mover os arquivos `.txt` para `.txt.old`: `find . -name '*.txt' -exec mv`

```
{} {}.old \;
```

Exemplos:

```
$ find /etc -iname 'x*'
/etc/X11
/etc/xinetd.d
/etc/log.d/conf/logfiles/xferlog.conf
/etc/log.d/scripts/logfiles/xferlog

$ find /etc -iname 'x*' -maxdepth 1
/etc/X11
/etc/xinetd.d

$ find /etc -type d -name 'cron*' -and -not -name 'cron.d*'
/etc/cron.hourly
/etc/cron.monthly
/etc/cron.weekly
```

Dicas:

- Releia os detalhes das opções `-exec` e `-ok`, você vai precisar.
- Para procurar mais de um tipo de arquivo:

```
$ find . -name '*.txt' -or -name '*.html'
```

- Para apagar todos os arquivos `.mp3` de um diretório, incluindo subdiretórios:

```
$ find . -iname '*.mp3' -exec rm -f {} \;
```

- Para encontrar todos os links simbólicos do diretório atual e mostrar para onde eles apontam:

```
$ find . -type l -ls | cut -d / -f 2-
```

- Em algumas versões do `find`, as opções `-iname`, `-ls` e `-printf` não estão disponíveis.
- Em algumas versões do `find`, as opções `-and` e `-or` são chamadas `-a` e `-o`, respectivamente.
- Em algumas versões do `find`, a opção `-not` não está disponível. Use a exclamação em seu lugar.

fmt

Formatador simples de parágrafos. Adiciona ou remove quebras de linha e espaços para que o texto não ultrapasse o número máximo de colunas.

Opções:

Opção	Lembrete	Descrição
`-w`	Width	Define o número máximo de colunas (o padrão é 75).
`-u`	Uniform	Remove espaços excedentes.

Exemplos:

```
$ cat fmt.txt
Hello!     This is     Linus Torvalds and I pronounce Linux as
Linux.
$ cat fmt.txt | fmt -w 35
Hello!     This is     Linus
Torvalds and I pronounce Linux
as Linux.

$ cat fmt.txt | fmt -w 35 -u
Hello!  This is Linus Torvalds
and I pronounce Linux as Linux.
```

Dicas:

- O `fmt` é similar ao comando `gq` do VI.
- Em algumas versões do `fmt`, a opção `-u` chama-se `-s`.
- O comando `fmt` não está disponível em algumas versões do Unix.

grep

Procura em arquivos ou textos por linhas que contêm determinado padrão de pesquisa. O padrão pode ser uma string ou uma expressão regular.

Opções:

Opção	Lembrete	Descrição

`-i`	Ignore case	Ignora a diferença entre maiúsculas e minúsculas.
`-v`	Invert	Mostra as linhas que não casam com o padrão.
`-r`	Recursive	Varre subdiretórios também.
`-q`	Quiet	Não mostra as linhas que encontrar (usar com o `test`).
`-s`	Silent	Não mostra os erros (usar com o `test`).
`-n`	Number	Mostra também o número da linha.
`-c`	Count	Conta o número de linhas encontradas.
`-l`	Filename	Mostra apenas o nome o arquivo que casou.
`-w`	Word	O padrão é uma palavra inteira, e não parte dela.
`-x`	Full line	O padrão é uma linha inteira, e não parte dela.
`-A`	After	Mostre *N* linhas de contexto depois do padrão.
`-B`	Before	Mostre *N* linhas de contexto antes do padrão.
`-C`	Context	Mostre *N* linhas de contexto antes e depois do padrão.

As identidades do grep	
`grep`	Procura por uma expressão regular básica.
`egrep` ou `grep -E`	Procura por uma expressão regular estendida.
`fgrep` ou `grep -F`	Procura por uma string.

Metacaracteres	
Expressão regular básica	`^ $ . * [ \? \+ | \( \) \{ \}`
Expressão regular estendida	`^ $ . * [ ? + | ( ) { }`

Exemplos:

```
$ seq 10 | grep -A1 -B3 "7"
4
5
6
7
8
$ cat grep.txt
1. Grep é g/re/p, que é Global Regular Expression Print.
2. Com o *grep na mão, nenhum dado está perdido.
3. Grep, Egrep e Fgrep. Uma família e tanto!
4. Não saia de casa sem o seu [fe]?grep.
5. Grep me harder!
$ cat grep.txt | fgrep grep
```

```
2. Com o *grep na mão, nenhum dado está perdido.
3. Grep, Egrep e Fgrep. Uma família e tanto!
4. Não saia de casa sem o seu [fe]?grep.
$ cat grep.txt | fgrep -w grep
2. Com o *grep na mão, nenhum dado está perdido.
4. Não saia de casa sem o seu [fe]?grep.
$ cat grep.txt | egrep -i '(grep.+){3}'
3. Grep, Egrep e Fgrep. Uma família e tanto!
$ cat grep.txt | grep -i '\(grep.\+\)\{3\}'
3. Grep, Egrep e Fgrep. Uma família e tanto!
$
```

Dicas:

- Sempre que não for pesquisar uma expressão regular, use o `fgrep`, pois ele é mais rápido.
- A opção `-l` é útil para corrigir um mesmo erro em vários arquivos:

```
vi $(grep -l 'emossão' *.txt)
```

- Note que o `-c` conta as linhas, e não as palavras:

```
$ echo 'grep, grep, grep, grep' | grep -c grep
1
```

- As opções `-q` e `-s` são usadas em testes e condicionais:

```
$ if grep -qs 'grep' grep.txt ; then echo achei ; fi
achei

$ grep -qs 'grep' grep.txt && echo achei
achei
```

- Em algumas versões do `grep`, as opções `-A`, `-B`, `-C`, `-r` e `-w` não estão disponíveis.

head

Mostra o início de um texto. A quantidade a ser mostrada pode ser expressa em linhas ou caracteres. Seu irmão é o `tail`.

Opções:

Opção	Lembrete	Descrição

`-n`	Lines	Mostra as *N* primeiras linhas (o padrão é 10).
`-c`	Char	Mostra os *N* primeiros caracteres (incluindo `\n`).

Exemplos:

```
$ head -n 3 numeros.txt
um
dois
três
$ head -c 5 numeros.txt
um
do
$ head -c 5 numeros.txt | od -c        # o \n também conta!
0000000   u   m  \n   d   o
0000005
```

Dicas:

- A letra *n* pode ser omitida da opção, especificando-se diretamente o número de linhas desejadas (mas evite fazer isso):

```
head -3 numeros.txt
```

- Em algumas versões do `head`, a opção `-c` não está disponível.

od

O od é o Octal Dump, que serve para mostrar o código de cada caractere de um arquivo ou texto. Além de octal, também mostra os códigos em hexadecimal e ASCII.

Opções:

Opção	Lembrete	Descrição
`-a`	Name	Mostra os nomes dos caracteres.
`-c`	ASCII	Mostra os caracteres ASCII.
`-o`	Octal	Mostra os códigos em octal.
`-x`	Hexa	Mostra os códigos em hexadecimal.

Exemplos:

```
$ echo od, que legal | od -h
```

```
0000000 646f 202c 7571 2065 656c 6167 0a6c
0000016

$ echo od, que legal | od -c
0000000   o   d   ,       q   u   e       l   e   g   a   l  \
n
0000016

$ echo -ne '\a\b\f\n\r\t\v' | od -ac
0000000 bel  bs  ff  nl  cr  ht  vt
         \a  \b  \f  \n  \r  \t  \v
0000007
```

Dicas:

- O `od` é especialmente útil para visualizar espaços em branco. Por exemplo, quantos espaços existem entre essas duas palavras?

```
quantos          brancos?
```

- Ao usar o `od`, vemos que há algo mais. São três espaços, um TAB e outro espaço:

```
$ echo "quantos          brancos?" | od -a
0000000   q   u   a   n   t   o   s  sp  sp  sp  ht  sp   b
r   a   n
0000020   c   o   s   ?  nl
0000025
```

- Em algumas versões do `od`, as opções de formatação estão todas embutidas na opção `-t`.

paste

Junta linhas de vários arquivos em um só. Ele mostra as linhas lado a lado, uma por uma, como se colocasse os arquivos em paralelo. Cada arquivo vira uma coluna do resultado. Com a opção `-s` o resultado é invertido e cada arquivo vira uma linha.

Opções:

Opção	Lembrete	Descrição
`-d`	Delimiter	Escolhe o delimitador (o padrão é o TAB).

`-s`	Serial	Transforma todas as linhas em apenas uma.

Exemplos:

```
$ cat numeros.txt
um
dois
três
quatro
cinco
$ cat numbers.txt
one
two
three
four
five
$ paste numeros.txt numbers.txt
um      one
dois    two
três    three
quatro  four
cinco   five
$ paste -s numeros.txt numbers.txt
um      dois    três    quatro  cinco
one     two     three   four    five
$ paste -d : numeros.txt numbers.txt
um:one
dois:two
três:three
quatro:four
cinco:five
$
```

Dicas:

- Mais de dois arquivos podem ser passados, inclusive o mesmo arquivo mais de uma vez:

```
$ paste numeros.txt numbers.txt numeros.txt numbers.txt
numeros.txt
um      one     um      one     um
```

```
dois    two     dois    two     dois
três    three   três    three   três
quatro  four    quatro  four    quatro
cinco   five    cinco   five    cinco
```

- Com apenas um arquivo, a opção `-s` é útil para transformar linhas em colunas, com o delimitador a escolher:

```
$ paste -s -d : numeros.txt
um:dois:três:quatro:cinco
```

- Se tiver uma lista de números que precisar somar, use o `paste` para inserir os sinais:

```
$ seq 5 2 15
5
7
9
11
13
15
$ seq 5 2 15 | paste -s -d+ -
5+7+9+11+13+15
$ seq 5 2 15 | paste -s -d+ - | bc
60
$
```

printf

Mostra um texto usando vários formatadores especiais. Todos os escapes usados pelo comando `echo` (como `\t` e `\n`) também são válidos no `printf`.

Opções:

Formato	Lembrete	Descrição
%d	Decimal	Número decimal.
%o	Octal	Número octal.
%x	Hexa	Número hexadecimal (a-f).
%X	Hexa	Número hexadecimal (A-F).
%f	Float	Número com ponto flutuante.

%e		Número em notação científica (e+1).
%E		Número em notação científica (E+1).
%s	String	String.

Exemplos

```
$ printf '%o \n' 10
12
$ printf '%X \n' 10
A
$ printf '%05d \n' 10
00010
$ printf '%f \n' 10
10.000000
$ printf '%.3f \n' 10
10.000
$ printf '%.3e \n' 10
1.000e+01
$ printf '(%15s) \n' direita
(        direita)
$ printf '(%-15s) \n' esquerda
(esquerda       )
```

Dicas:

- Diferente do `echo`, o `printf` não quebra a linha no final da string. Deve-se colocar um `\n` no final para obter esse efeito.
- As opções de alinhamento do `%s` são muito boas para compor relatórios:

```
$ z=15; printf "%${z}s:\n%${z}s:\n%${z}s:\n" 'Nome Completo'
Idade Sexo
    Nome Completo:
            Idade:
             Sexo:
```

rev

Inverte a ordem dos caracteres da linha, colocando-os de trás para frente. Funciona como um espelho onde "texto" vira "otxet". Seu primo que inverte linhas é o `tac`.

Exemplos:

```
$ echo O rev é muito bacana. | rev
.anacab otium é ver O

$ echo O rev é muito bacana. | rev | rev
O rev é muito bacana.
```

Dicas:

- Usando o `cut` sozinho, não há como especificar o último campo, ou o último caractere. Com o `rev`, isso é possível:

```
$ echo 123456789 | rev | cut -c 1
9
```

- Similarmente, para pegar os cinco últimos caracteres:

```
$ echo 123456789 | rev | cut -c 1-5 | rev
56789
```

- O comando `rev` não está disponível em algumas versões do Unix.

sed

Editor de textos não interativo e programável. Executa uma série de comandos de edição em um texto. Seu uso clássico é trocar uma string por outra, assim:
`s/isso/aquilo/`. Esse comando significa: troque "isso" por "aquilo".

Opções:

Opção	Lembrete	Descrição
`-n`	Not print	Só mostra a linha caso usado o comando `p`.
`-e`	Expression	Especifica os comandos de edição.
`-f`	File	Lê os comandos de edição de um arquivo.

Comando	Lembrete	Ação
`s///`	Substitute	Troca um texto por outro.

`p`	Print	Mostra a linha na saída.
`l`	List	Mostra a linha na saída, com `\t`, `\a`, ...
`d`	Delete	Apaga a linha.
`q`	Quit	Sai do `sed`.
`r`	Read	Lê o conteúdo de um arquivo.
`N`	Next line	Junta a próxima linha com a atual.

Endereço	Abrange...
`1`	A primeira linha.
`1,5`	Da primeira linha até a quinta.
`5,$`	Da quinta linha até a última.
`/sed/`	A(s) linha(s) que contém a palavra "sed".
`5,/sed/`	Da quinta linha até a linha que contém "sed".
`/sed/,/grep/`	Da linha que contém "sed" até a que contém "grep".
`1,5!`	Todas as linhas, exceto da primeira a quinta.
`/sed/!`	A(s) linha(s) que não contém a palavra "sed".

s///	Exemplo	Descrição
`g`	`s/a/b/g`	Modificador Global, para trocar todas as ocorrências.
`p`	`s/a/b/gp`	Modificador Print, para mostrar o texto substituído.
`&`	`s/./& /`	Expande para todo o trecho casado na primeira parte.
`\1`	`s/\(.\)/\1 /`	Expande para o conteúdo do primeiro grupo marcado com `\(...\)`.

Exemplos:

```
$ cat numeros.txt | sed '1,3 d'
quatro
cinco
$ cat numeros.txt | sed '/o$/ d'
um
dois
três

$ cat numeros.txt | sed '/dois/ q'
um
dois

$ echo um dois três quatro cinco | sed 's/ //g'
```

```
umdoistrêsquatrocinco

$ cat numeros.txt | sed 's/tr/_TR_/ ; s/^/- /'
- um
- dois
- _TR_ês
- qua_TR_o
- cinco
```

Dicas:

- O `sed` usa os mesmos metacaracteres básicos do `grep`.
- O comando `l` é útil para decifrar espaços em branco:

```
$ echo -e '\t\t\a  ' | sed -n l
\t\t\a  $
```

- Pode-se usar outro delimitador no comando `s///` para evitar confusão:

```
$ echo /usr/bin/sed | sed 's,/usr/bin,/tmp,'
/tmp/sed
```

- Algumas versões do `sed` reconhecem os mesmos escapes do comando `echo`:

```
$ echo um dois três quatro cinco | sed 's/ /\t/g'
um      dois    três    quatro  cinco
```

- O comando `p` pode ser usado para mostrar a linha antes de editá-la:

```
$ cat numeros.txt | sed 'p ; s/./& /g ; s/^/---> /'
um
---> u m
dois
---> d o i s
três
---> t r ê s
quatro
---> q u a t r o
cinco
---> c i n c o
```

seq

Mostra uma sequência numérica na tela, um número por linha. A contagem pode ser crescente ou decrescente.

Opções:

Opção	Lembrete	Descrição
`-s`	Separator	Define o separador (o padrão é `\n`).
`-f`	Format	Define o formato do número (o padrão é `%g`).

Exemplos:

```
$ seq 4             # Conte até 4
1
2
3
4

$ seq 2 5           # Conte de 2 a 5
2
3
4
5
$ seq 0 5 20        # Conte de zero a 20, andando de 5 em 5
(passo=5)
0
5
10
15
20
$ seq 4 -1 0        # Conte de 4 até zero, com o passo de -1
4
3
2
1
0
$ seq 4 -2 -4       # Conte de 4 até -4, com o passo de -2
4
2
0
```

```
-2
-4
$ seq -s , 20
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20

$ seq -s '...' -f '%04g' 7
0001...0002...0003...0004...0005...0006...0007
```

Dicas:

- O `seq` é útil para usar junto com o `for`:

```
for i in $(seq 10); do echo Contando: $i; done
```

- O comando `seq` não está disponível no Unix. Um outro comando similar chama-se `jot`. Veja mais detalhes em Shell no Linux, Mac e Windows, página 361.

sort

Ordena as linhas de um texto, alfabética ou numericamente.

Opções:

Opção	Lembrete	Descrição
`-n`	Numeric	Ordena numericamente (o padrão é alfabeticamente).
`-r`	Reverse	Reverte a ordenação (de Z para A, de 9 para 0).
`-f`	Ignore case	Ignora a diferença entre maiúsculas e minúsculas.
`-k`	Key	Ordena pela coluna *`N`* (a primeira é 1).
`-t`	Separator	Escolhe o separador para o `-k` (o padrão é o TAB).
`-o`	Output	Grava a saída no arquivo especificado.

Exemplos:

```
$ sort numeros.txt
cinco
dois
quatro
três
um
```

```
$ cat -n numeros.txt | sort -k 2
```

```
     5  cinco
     2  dois
     4  quatro
     3  três
     1  um
$ echo -e "1\n10\n100\n2\n20\n200" | sort
1
10
100
2
20
200
$ echo -e "1\n10\n100\n2\n20\n200" | sort -n
1
2
10
20
100
200
```

Dicas:

- Usando a opção `-o` é possível ordenar um arquivo e gravar o resultado nele mesmo:

```
$ sort arquivo.txt -o arquivo.txt
```

tac

Preste atenção no nome do comando: `tac` é o contrário de `cat`. Ele mostra o texto ao contrário, da última para a primeira linha. O que o `rev` faz com caracteres, o `tac` faz com linhas.

Exemplos:

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

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

Dicas:

- O `tac` é útil para se usar em conjunto com o `sed`, `grep`, `awk` ou outros processadores orientados à linha, quando se precisa procurar ou mostrar registros em ordem reversa.
- O comando `tac` não está disponível no Unix. Use o `tail -r` para obter o mesmo comportamento. Veja mais informações em Shell no Linux, Mac e Windows, página 361.

tail

Mostra o final de um texto. A quantidade a ser mostrada pode ser expressa em linhas ou caracteres. Seu irmão é o `head`.

Opções:

Opção	Lembrete	Descrição
`-n`	Lines	Mostra as *N* últimas linhas (o padrão é 10).
`-c`	Char	Mostra os *N* últimos caracteres (incluindo `\n`).
`-f`	Follow	Monitora o arquivo *ad infinitum*.

Exemplos:

```
$ tail -n 3 numeros.txt
três
quatro
cinco
$ tail -c 9 numeros.txt
ro
cinco
$ tail -c 9 numeros.txt | od -c        # o \n também conta!
0000000   r   o  \n   c   i   n   c   o  \n
```

```
0000011
```

Dicas:

- A letra *n* pode ser omitida da opção, especificando-se diretamente o número de linhas desejadas (mas evite fazer isso):

```
tail -3 numeros.txt
```

- A opção `-f` é muito útil para monitorar arquivos de log do sistema em tempo real.

```
tail -f /var/log/messages
```

tee

Salva o fluxo de dados de um pipe para um arquivo, sem interrompê-lo.

Opções:

Opção	Lembrete	Descrição
`-a`	Append	Anexa ao final do arquivo (o padrão é sobrescrever).

Exemplos:

```
$ echo "um texto qualquer" | tee tee.txt
um texto qualquer
$ cat tee.txt
um texto qualquer
$ echo "um outro texto" | tee -a tee.txt
um outro texto
$ cat tee.txt
um texto qualquer
um outro texto
```

Dicas:

- Útil para fazer depuração (debug), obtendo o estado de um fluxo em um comando muito extenso.
- Útil para mensagens de log, que vão para a tela e também para um arquivo.

tr

Transforma, espreme e apaga caracteres, funcionando como um filtro. Reconhece os mesmos escapes do comando `echo`.

Opções:

Opção	Lembrete	Descrição
`-s`	Squeeze	Espreme caracteres iguais consecutivos para apenas um.
`-d`	Delete	Apaga todos os caracteres listados.
`-c`	Complement	Inverte a lista de caracteres (`-c 0-9` é similar a `[^0-9]`).

Argumento	Engloba
abc	“a” e “b” e “c”
a7z	“a” e “7” e “z”
a-z	de “a” até “z”
0-7	de zero a sete

Exemplos:

```
$ echo O tr é muito versátil | tr ' ' '\t'
O       tr      é       muito   versátil
$ echo O tr é muito versátil | tr a-z A-Z
O TR é MUITO VERSáTIL
$ echo O tr é muito versátil | tr a-zéá A-ZÉÁ
O TR É MUITO VERSÁTIL
$ echo O tr é muito versátil | tr -d aeiouéá
O tr  mt vrstl
$ echo O tr é muito versátil | tr -d -c aeiouéá
éuioeái
$ echo "muitos      espaços" | tr -s ' '
muitos espaços
$ cat numeros.txt | tr '\n' ,
um,dois,três,quatro,cinco,
```

Dicas:

- Em sistemas com a localização corretamente configurada para o português, os seguintes parâmetros também englobam caracteres

acentuados:

```
$ echo O tr é muito versátil | tr [:lower:] [:upper:]
O TR É MUITO VERSÁTIL
```

- Em algumas versões do `tr`, a opção `-c` chama-se `-C` (maiúsculo).

uniq

Remove linhas repetidas consecutivas em um texto, deixando apenas uma.

Opções:

Opção	Lembrete	Descrição
`-i`	Ignore case	Ignora a diferença entre maiúsculas e minúsculas.
`-d`	Duplicate	Mostra apenas as linhas que são repetidas.
`-u`	Unique	Mostra apenas as linhas que não são repetidas.

Exemplos:

```
$ cat uniq.txt
um
um
um
dois
Dois
DOIS
três
um
$ cat uniq.txt | uniq
um
dois
Dois
DOIS
três
um
$ cat uniq.txt | uniq -i
um
dois
três
```

```
um
$ cat uniq.txt | uniq -i -d
um
dois
$ cat uniq.txt | uniq -i -u
três
um
```

Dicas:

- Para remover todas as linhas repetidas, mesmo que não sejam consecutivas, é preciso ordenar o texto primeiro:

```
$ sort -f uniq.txt | uniq -i
DOIS
três
um
```

- Para fazer a mesma operação anterior, mas mantendo a mesma ordem original das linhas, veja a função `zzuniq` em Análise das Funções ZZ, página 385.
- Em algumas versões do `uniq`, a opção `-i` não está disponível.

wc

Conta letras, palavras e linhas em um texto.

Opções:

Opção	Lembrete	Descrição
`-c`	Char	Conta o número de caracteres (bytes).
`-w`	Word	Conta o número de palavras.
`-l`	Line	Conta o número de linhas.
`-L`	Longest	Mostra o tamanho da linha mais comprida.

Exemplos:

```
$ paste numeros.txt numbers.txt
um      one
dois    two
três    three
```

```
quatro  four
cinco   five
$ paste numeros.txt numbers.txt | wc     # linhas, palavras e
caracteres
       5      10      50
$ wc -l numeros.txt numbers.txt
       5 numeros.txt
       5 numbers.txt
      10 total
```

Dicas:

- A opção `-L` é útil quando se precisa posicionar um texto na tela, pois saber o tamanho da linha mais comprida é essencial:

```
tamanho_max=$(wc -L < arquivo.txt)
```

- Há diferença entre passar o nome do arquivo para o `wc` ou mandar o texto via `STDIN`. Em programas, geralmente usa-se a segunda forma, para obter apenas o número, sem o nome do arquivo:

```
$ wc -l numeros.txt
5 numeros.txt
$ wc -l < numeros.txt
5
```

- Em algumas versões do `wc`, a opção `-L` não está disponível.

xargs

Gerenciador de argumentos da linha de comando. Executa comandos passando os argumentos recebidos via `STDIN`. O comando padrão é o `echo`.

Opções:

Opção	Lembrete	Descrição
`-n`	Number	Use *N* argumentos por linha de comando.
`-i`	Replace	Troca a string `{}` pelo argumento da vez.

Exemplos:

```
$ cat numeros.txt | xargs
```

```
um dois três quatro cinco
$ cat numeros.txt | xargs -n 2
um dois
três quatro
cinco
$ cat numeros.txt | xargs -i echo Contando: {}...
Contando: um...
Contando: dois...
Contando: três...
Contando: quatro...
Contando: cinco...
```

Dicas:

- Útil para aplicar um comando para uma lista de arquivos, por exemplo, para criar cópias de todos os arquivos texto do diretório atual:

```
ls -1 *.txt | xargs -i cp {} {}.bak
```

- O comando similar ao anterior sem o `xargs` requer o uso de loop:

```
for arq in *.txt ; do cp $arq $arq.bak ; done
```

- Útil para evitar o uso de subshell e tratar nomes de arquivos com espaços, onde:

```
rm $(grep -l 'string' *.txt)
```

fica:

```
grep -l 'string' *.txt | xargs rm
```

- Em algumas versões do `xargs`, a opção `-i` não está disponível. Use a `-I` (letra i maiúscula) em seu lugar, informando a string que será substituída:

```
cat numeros.txt | xargs -I "{}" echo Contando: {}...
```

Apêndice E

Canivete Suíço

O Canivete Suíço é um resumo das características do shell e dos comandos do sistema. São várias tabelas que listam operadores, variáveis, opções, parâmetros, conceitos e dicas. É uma quantidade imensa de informações em um formato fácil de consultar. Use como guia de referência rápida nos momentos de dúvida. Estude as tabelas para aprender mais sobre o shell.

Operadores

Operadores aritméticos	
+	Adição.
-	Subtração.
*	Multiplicação.
/	Divisão.
%	Módulo.
**	Exponenciação.
Operadores de atribuição	
=	Atribui valor a uma variável.
+=	Incrementa a variável por uma constante.
-=	Decrementa a variável por uma constante.
*=	Multiplica a variável por uma constante.
/=	Divide a variável por uma constante.
%=	Resto da divisão por uma constante.
++	Incrementa em 1 o valor da variável.
--	Decrementa em 1 o valor da variável.
Operadores relacionais	
==	Igual.
!=	Diferente.
>	Maior.
>=	Maior ou igual.
<	Menor.
<=	Menor ou igual.
Operadores lógicos	
&&	E lógico (AND).
||	OU lógico (OR).
Operadores de bit	
<<	Deslocamento à esquerda.
>>	Deslocamento à direita.
&	E de bit (AND).

|	OU de bit (OR).
^	OU exclusivo de bit (XOR).
~	Negação de bit.
!	NÃO de bit (NOT).
Operadores de bit (atribuição)	
<<=	Deslocamento à esquerda.
>>=	Deslocamento à direita.
&=	E de bit.
|=	OU de bit.
^=	OU exclusivo de bit.

Redirecionamento

Operador	Ação
<	Redireciona a entrada padrão (STDIN).
>	Redireciona a saída padrão (STDOUT).
2>	Redireciona a saída de erro (STDERR).
>>	Redireciona a saída padrão, anexando.
2>>	Redireciona a saída de erro, anexando.
|	Conecta a saída padrão com a entrada padrão de outro comando.
2>&1	Conecta a saída de erro na saída padrão.
>&2	Conecta a saída padrão na saída de erro.
>&-	Fecha a saída padrão.
2>&-	Fecha a saída de erro.
3<>*arq*	Conecta o descritor de arquivos 3 ao arquivo *arq*.
<<FIM	Alimenta a entrada padrão (*Here Document*).
<<-FIM	Alimenta a entrada padrão, cortando TABs.
<(*cmd*)	A saída do comando *cmd* é um arquivo: `diff <(cmd1) <(cmd2)`.
>(*cmd*)	A entrada do comando *cmd* é um arquivo: `tar cf >(bzip2 -c >file.tbz) $dir`.

Variáveis especiais

Parâmetros Posicionais	

`$0`	Parâmetro número 0 (nome do comando ou função).
`$1`	Parâmetro número 1 (da linha de comando ou função).
...	Parâmetro número N ...
`$9`	Parâmetro número 9 (da linha de comando ou função).
`${10}`	Parâmetro número 10 (da linha de comando ou função).
...	Parâmetro número NN ...
`$#`	Número total de parâmetros da linha de comando ou função.
`$*`	Todos os parâmetros, como uma string única.
`$@`	Todos os parâmetros, como várias strings protegidas.
Outras	
`$$`	Número PID do processo atual (do próprio script).
`$!`	Número PID do último job em segundo plano.
`$_`	Último argumento do último comando executado.
`$?`	Código de retorno do último comando executado.

Expansão de variáveis

Expansão Condicional	
`${var:-texto}`	Se *`var`* não está definida, retorna *`texto`*.
`${var:=texto}`	Se *`var`* não está definida, defina-a com *`texto`*.
`${var:?texto}`	Se *`var`* não está definida, retorna o erro *`texto`*.
`${var:+texto}`	Se *`var`* está definida, retorna *`texto`*, senão retorna o vazio.

Expansão de Strings	
`${var}`	É o mesmo que `$var`, porém não ambíguo.
`${#var}`	Retorna o tamanho da string.
`${!var}`	Executa o conteúdo de `$var` (igual `eval \$$var`).
`${!texto*}`	Retorna os nomes de variáveis começadas por *`texto`*.
`${var:N}`	Retorna o texto à partir da posição *`N`*.
`${var:N:tam}`	Retorna *`tam`* caracteres à partir da posição *`N`*.
`${var#texto}`	Corta *`texto`* do início da string.
`${var##texto}`	Corta *`texto`* do início da string (* guloso).
`${var%texto}`	Corta *`texto`* do final da string.
`${var%%texto}`	Corta *`texto`* do final da string (* guloso).

`${var/texto/novo}`	Substitui *texto* por *novo*, uma vez.
`${var//texto/novo}`	Substitui *texto* por *novo*, sempre.
`${var/#texto/novo}`	Se a string começar com *texto*, substitui *texto* por *novo*.
`${var/%texto/novo}`	Se a string terminar com *texto*, substitui *texto* por *novo*.

Blocos e agrupamentos

Sintaxe	Descrição	Exemplo
`"..."`	Protege uma string, mas reconhece $, \ e ` como especiais	`"abc"`
`'...'`	Protege uma string, nenhum caractere é especial	`'abc'`
`$'...'`	Protege uma string, mas interpreta `\n`, `\t`, `\a` etc.	`$'abc\n'`
`` `...` ``	Executa comandos em uma subshell, retornando o resultado	`` `ls` ``
`{...}`	Agrupa comandos em um bloco	`{ ls ; }`
`(...)`	Executa comandos em uma subshell	`( ls )`
`$(...)`	Executa comandos em uma subshell, retornando o resultado	`$( ls )`
`((...))`	Testa uma operação aritmética, retornando 0 ou 1	`((5 > 3))`
`$((...))`	Retorna o resultado de uma operação aritmética	`$((5+3))`
`[...]`	Testa uma expressão, retornando 0 ou 1 (alias do comando `test`)	`[ 5 -gt 3 ]`
`[[...]]`	Testa uma expressão, retornando 0 ou 1 (podendo usar && e ||)	`[[ 5 > 3 ]]`

Opções do comando test

Testes em arquivos	
`-b`	É um dispositivo de bloco.
`-c`	É um dispositivo de caractere.
`-d`	É um diretório.
`-e`	O arquivo existe.
`-f`	É um arquivo normal.
`-g`	O bit SGID está ativado.
`-G`	O grupo do arquivo é o do usuário atual.
`-k`	O sticky-bit está ativado.
`-L`	O arquivo é um link simbólico.
`-O`	O dono do arquivo é o usuário atual.
`-p`	O arquivo é um named pipe.

`-r`	O arquivo tem permissão de leitura.
`-s`	O tamanho do arquivo é maior que zero.
`-S`	O arquivo é um socket.
`-t`	O descritor de arquivos N é um terminal.
`-u`	O bit SUID está ativado.
`-w`	O arquivo tem permissão de escrita.
`-x`	O arquivo tem permissão de execução.
`-nt`	O arquivo é mais recente (NewerThan).
`-ot`	O arquivo é mais antigo (OlderThan).
`-ef`	O arquivo é o mesmo (EqualFile).
Comparação Numérica	
`-lt`	É menor que (LessThan).
`-gt`	É maior que (GreaterThan).
`-le`	É menor igual (LessEqual).
`-ge`	É maior igual (GreaterEqual).
`-eq`	É igual (EQual).
`-ne`	É diferente (NotEqual).
Comparação de Strings	
`=`	É igual.
`!=`	É diferente.
`-n`	É não nula.
`-z`	É nula.
Operadores Lógicos	
`!`	NÃO lógico (NOT).
`-a`	E lógico (AND).
`-o`	OU lógico (OR).

Escapes do prompt (PS1)

Escape	Lembrete	Expande para...
`\a`	Alerta	Alerta (bipe).
`\d`	Data	Data no formato “Dia-da-semana Mês Dia” (Sat Jan 15).
`\e`	Escape	Caractere Esc.

`\h`	Hostname	Nome da máquina sem o domínio (dhcp11).
`\H`	Hostname	Nome completo da máquina (dhcp11.empresa).
`\j`	Jobs	Número de jobs ativos.
`\l`	Tty	Nome do terminal corrente (ttyp1).
`\n`	Newline	Linha nova.
`\r`	Return	Retorno de carro.
`\s`	Shell	Nome do shell (`basename $0`).
`\t`	Time	Horário no formato 24 horas HH:MM:SS.
`\T`	Time	Horário no formato 12 horas HH:MM:SS.
`\@`	At	Horário no formato 12 horas HH:MM am/pm.
`\A`	At	Horário no formato 24 horas HH:MM.
`\u`	Usuário	Login do usuário corrente.
`\v`	Versão	Versão do Bash (2.00).
`\V`	Versão	Versão+subversão do Bash (2.00.0).
`\w`	Working Dir	Diretório corrente, caminho completo (`$PWD`).
`\W`	Working Dir	Diretório corrente, somente o último (`basename $PWD`).
`\!`	Histórico	Número do comando corrente no histórico.
`\#`	Número	Número do comando corrente.
`\$`	ID	Mostra # se for root, $ se for usuário normal.
`\`*nnn*	Octal	Caractere cujo octal é *nnn*.
`\\`	Backslash	Barra invertida \ literal.
`\[`	Escapes	Inicia uma sequência de escapes (como códigos de cores).
`\]`	Escapes	Termina uma sequência de escapes.

Escapes do comando echo

Escape	Lembrete	Descrição
`\a`	Alerta	Alerta (bipe).
`\b`	Backspace	Caractere Backspace.
`\c`	EOS	Termina a string.
`\e`	Escape	Caractere Esc.
`\f`	Form feed	Alimentação.
`\n`	Newline	Linha nova.

`\r`	Return	Retorno de carro.
`\t`	Tab	Tabulação horizontal.
`\v`	Vtab	Tabulação vertical.
`\\`	Backslash	Barra invertida \ literal.
`\`*`nnn`*	Octal	Caractere cujo octal é *nnn*.
`\x`*`nn`*	Hexa	Caractere cujo hexadecimal é *nn*.

Formatadores do comando date

Formato	Descrição
`%a`	Nome do dia da semana abreviado (Dom..Sáb).
`%A`	Nome do dia da semana (Domingo..Sábado).
`%b`	Nome do mês abreviado (Jan..Dez).
`%B`	Nome do mês (Janeiro..Dezembro).
`%c`	Data completa (Sat Nov 04 12:02:33 EST 1989).
`%y`	Ano (dois dígitos).
`%Y`	Ano (quatro dígitos).
`%m`	Mês (01..12).
`%d`	Dia (01..31).
`%j`	Dia do ano (001..366).
`%H`	Horas (00..23).
`%M`	Minutos (00..59).
`%S`	Segundos (00..60).
`%s`	Segundos desde 1º de Janeiro de 1970.
`%%`	Um % literal.
`%t`	Um TAB.
`%n`	Uma quebra de linha.

Formatadores do comando printf

Formato	Descrição
%d	Número decimal.
%o	Número octal.
%x	Número hexadecimal (a-f).

%X	Número hexadecimal (A-F).
%f	Número com ponto flutuante.
%e	Número em notação científica (e+1).
%E	Número em notação científica (E+1).
%s	String.

Letras do comando ls -l

Letra	Lembrete	Tipos de Arquivo (primeiro caractere)
-	-	Arquivo normal.
d	Directory	Diretório.
l	Link	Link simbólico.
b	Block	Dispositivo de blocos (HD).
c	Char	Dispositivo de caracteres (modem serial).
s	Socket	Socket mapeado em arquivo (comunicação de processos).
p	Pipe	FIFO ou Named Pipe (comunicação de processos).

Letra	Lembrete	Permissões do Arquivo (próximos 9 caracteres)
-	-	Permissão desativada.
r	Read	Acesso de leitura.
w	Write	Acesso de escrita.
x	eXecute	Acesso de execução (ou acesso ao diretório).
X	eXecute	Acesso ao diretório somente.
s	Set ID	Usuário/grupo para execução (SUID, SGID), permissão 'x' ativada.
S	Set ID	Usuário/grupo para execução (SUID, SGID), permissão 'x' desativada.
t	sTicky	Usuários só apagam seus próprios arquivos, permissão 'x' ativada.
T	sTicky	Usuários só apagam seus próprios arquivos, permissão 'x' desativada.

Curingas para nomes de arquivo (glob)

Curinga	Casa com...	Exemplo
*	Qualquer coisa	*.txt

`?`	Um caractere qualquer	`arquivo-??.zip`
`[...]`	Qualquer um dos caracteres listados	`[Aa]rquivo.txt`
`[^...]`	Qualquer um caractere, exceto os listados	`[^A-Z]*.txt`
`{...}`	Qualquer um dos textos separados por vírgula	`arquivo.{txt,html}`

Curingas para o comando case

Curinga	Casa com...	Exemplo
*	Qualquer coisa	`*.txt) echo ;;`
?	Um caractere qualquer	`arquivo-??.zip) echo ;;`
`[...]`	Qualquer um dos caracteres listados	`[0-9]) echo ;;`
`[^...]`	Qualquer um caractere, exceto os listados	`[^0-9]) echo ;;`
`...|...`	Qualquer um dos textos separados por |	`txt|html) echo ;;`

Metacaracteres nos aplicativos

Programa	Opcional	Mais	Chaves	Borda	Ou	Grupo
awk	?	+	-	-	|	()
ed	\?	\+	\{,\}	\b	|	\(\)
egrep	?	+	{,}	\b	|	()
emacs	?	+	-	\b	|	\(\)
expect	?	+	-	-	|	()
find	?	+	-	\b	|	\(\)
gawk	?	+	{,}	\<\>	|	()
grep	\?	\+	\{,\}	\b	|	\(\)
mawk	?	+	-	-	|	()
perl	?	+	{,}	\b	|	()
php	?	+	{,}	\b	|	()
python	?	+	{,}	\b	|	()
sed	\?	\+	\{,\}	\<\>	|	\(\)
vim	\=	\+	\{,}	\<\>	|	\(\)

Sinais para usar com trap/kill/killall

#	Linux	Cygwin	SystemV	AIX	HP-UX	Solaris	BSD/Mac

1	HUP	HUP	HUP	HUP	HUP	HUP	HUP
2	INT	INT	INT	INT	INT	INT	INT
3	QUIT	QUIT	QUIT	QUIT	QUIT	QUIT	QUIT
4	ILL	ILL	ILL	ILL	ILL	ILL	ILL
5	TRAP	TRAP	TRAP	TRAP	TRAP	TRAP	TRAP
6	ABRT	ABRT	IOT	LOST	ABRT	ABRT	ABRT
7	BUS	EMT	EMT	EMT	EMT	EMT	EMT
8	FPE	FPE	FPE	FPE	FPE	FPE	FPE
9	KILL	KILL	KILL	KILL	KILL	KILL	KILL
10	USR1	BUS	BUS	BUS	BUS	BUS	BUS
11	SEGV	SEGV	SEGV	SEGV	SEGV	SEGV	SEGV
12	USR2	SYS	SYS	SYS	SYS	SYS	SYS
13	PIPE	PIPE	PIPE	PIPE	PIPE	PIPE	PIPE
14	ALRM	ALRM	ALRM	ALRM	ALRM	ALRM	ALRM
15	TERM	TERM	TERM	TERM	TERM	TERM	TERM
16	-	URG	USR1	URG	USR1	USR1	URG
17	CHLD	STOP	USR2	STOP	USR2	USR2	STOP
18	CONT	TSTP	CHLD	TSTP	CHLD	CHLD	TSTP
19	STOP	CONT	PWR	CONT	PWR	PWR	CONT
20	TSTP	CHLD	WINCH	CHLD	VTALRM	WINCH	CHLD
21	TTIN	TTIN	URG	TTIN	PROF	URG	TTIN
22	TTOU	TTOU	IO	TTOU	IO	IO	TTOU
23	URG	IO	STOP	IO	WINCH	STOP	IO
24	XCPU	XCPU	TSTP	XCPU	STOP	TSTP	XCPU
25	XFSZ	XFSZ	CONT	XFSZ	TSTP	CONT	XFSZ
26	VTALRM	VTALRM	TTIN	-	CONT	TTIN	VTALRM
27	PROF	PROF	TTOU	MSG	TTIN	TTOU	PROF
28	WINCH	WINCH	VTALRM	WINCH	TTOU	VTALRM	WINCH
29	IO	LOST	PROF	PWR	URG	PROF	INFO
30	PWR	USR1	XCPU	USR1	LOST	XCPU	USR1
31	SYS	USR2	XFSZ	USR2	-	XFSZ	USR2
32	-	-	-	PROF	-	WAITING	-

33	-	-	-	DANGER	-	LWP	-
34	-	-	-	VTALRM	-	FREEZE	-
35	-	-	-	MIGRATE	-	THAW	-
36	-	-	-	PRE	-	CANCEL	-
37	-	-	-	-	-	LOST	-

Como obter a listagem: `trap -l`, `kill -l` ou `killall -l`

Veja também: `man 7 signal`

Códigos de retorno de comandos

Código	Significado	Exemplo
0	Nenhum erro, execução terminou OK	`echo`
1	A maioria dos erros comuns na execução	`echo $((1/0))`
2	Erro de uso em algum *builtin* do Shell	-
126	Comando não executável (sem permissão)	`touch a ; ./a`
127	Comando não encontrado (*command not found*)	`echooo`
128	O parâmetro para o `exit` não é um decimal	`exit 1.0`
128+*n*	128 + código do sinal que o matou	`kill -9 $PPID #exit 137`
130	O programa interrompido com o Ctrl+C (128 + 2)	-
255	Parâmetro para o `exit` não está entre 0 e 255	`exit -1`

Códigos de cores (ANSI)

Cor	Letra	Fundo
Preto	30	40
Vermelho	31	41
Verde	32	42
Amarelo	33	43
Azul	34	44
Rosa	35	45
Ciano	36	46
Branco	37	47

Atributo	Valor

Reset	0
Negrito	1
Sublinhado	4
Piscando	5
Reverso	7

Exemplos: ESC [<N>;<N> m

```
ESC[m texto normal (desliga cores)
ESC[1m negrito
ESC[33;1m amarelo
ESC[44;37m fundo azul, letra cinza
ESC[31;5m vermelho piscando
```

Na linha de comando

```
echo -e '\e[33;1m amarelo \e[m'
echo -e '\033[33;1m amarelo \033[m'
```

Metacaracteres das expressões regulares

Meta	Nome	Descrição
`.`	Ponto	Curinga de um caractere.
`[]`	Lista	Casa qualquer um dos caracteres listados.
`[^]`	Lista negada	Casa qualquer caractere, exceto os listados.
`?`	Opcional	A entidade anterior pode aparecer ou não (opcional).
`*`	Asterisco	A entidade anterior pode aparecer em qualquer quantidade.
`+`	Mais	A entidade anterior deve aparecer no mínimo uma vez.
`{,}`	Chaves	A entidade anterior deve aparecer na quantidade indicada.
`^`	Circunflexo	Casa o começo da linha.
`$`	Cifrão	Casa o fim da linha.
`\b`	Borda	Limita uma palavra (letras, números e sublinhado).
`\`	Escape	Escapa um meta, tirando seu poder.
`|`	Ou	Indica alternativas (usar com o grupo).
`()`	Grupo	Agrupa partes da expressão, é quantificável e multinível.
`\1`	Retrovisor	Recupera o conteúdo do grupo 1.
`\2`	Retrovisor	Recupera o conteúdo do grupo 2 (segue até o \9).
`.*`	Curinga	Casa qualquer coisa, é o tudo e o nada.

`??`	Opcional NG	Idem ao opcional comum, mas casa o mínimo possível.
`*?`	Asterisco NG	Idem ao asterisco comum, mas casa o mínimo possível.
`+?`	Mais NG	Idem ao mais comum, mas casa o mínimo possível.
`{}?`	Chaves NG	Idem às chaves comuns, mas casa o mínimo possível.

Atalhos da linha de comando (set -o emacs)

Atalho	Descrição	Tecla Similar
Ctrl+A	Move o cursor para o início da linha	Home
Ctrl+B	Move o cursor uma posição à esquerda	←
Ctrl+C	Envia sinal EOF() para o sistema	
Ctrl+D	Apaga um caractere à direita	Delete
Ctrl+E	Move o cursor para o fim da linha	End
Ctrl+F	Move o cursor uma posição à direita	→
Ctrl+H	Apaga um caractere à esquerda	Backspace
Ctrl+I	Completa arquivos e comandos	TAB
Ctrl+J	Quebra a linha	Enter
Ctrl+K	Recorta do cursor até o fim da linha	
Ctrl+L	Limpa a tela (igual ao comando clear)	
Ctrl+N	Próximo comando	
Ctrl+P	Comando anterior	
Ctrl+Q	Destrava a shell (veja Ctrl+S)	
Ctrl+R	Procura no histórico de comandos	
Ctrl+S	Trava a shell (veja Ctrl+Q)	
Ctrl+T	Troca dois caracteres de lugar	
Ctrl+U	Recorta a linha inteira	
Ctrl+V	Insere caractere literal	
Ctrl+W	Recorta a palavra à esquerda	
Ctrl+X	Move o cursor para o início/fim da linha (2x)	Home/End
Ctrl+Y	Cola o trecho recortado	

Caracteres ASCII imprimíveis (ISO-8859-1)

32		64	@	96	`	162	¢	194	Â	226	â
33	!	65	A	97	a	163	£	195	Ã	227	ã
34	"	66	B	98	b	164	¤	196	Ä	228	ä
35	#	67	C	99	c	165	¥	197	Å	229	å
36	$	68	D	100	d	166	¦	198	Æ	230	æ
37	%	69	E	101	e	167	§	199	Ç	231	ç
38	&	70	F	102	f	168	¨	200	È	232	è
39	'	71	G	103	g	169	©	201	É	233	é
40	(	72	H	104	h	170	ª	202	Ê	234	ê
41	)	73	I	105	i	171	«	203	Ë	235	ë
42	*	74	J	106	j	172	¬	204	Ì	236	ì
43	+	75	K	107	k	173		205	Í	237	í
44	,	76	L	108	l	174	®	206	Î	238	î
45	-	77	M	109	m	175	¯	207	Ï	239	ï
46	.	78	N	110	n	176	°	208	Đ	240	ð
47	/	79	O	111	o	177	±	209	Ñ	241	ñ
48	0	80	P	112	p	178	²	210	Ò	242	ò
49	1	81	Q	113	q	179	³	211	Ó	243	ó
50	2	82	R	114	r	180	´	212	Ô	244	ô
51	3	83	S	115	s	181	µ	213	Õ	245	õ
52	4	84	T	116	t	182	¶	214	Ö	246	ö
53	5	85	U	117	u	183	·	215	×	247	÷
54	6	86	V	118	v	184	¸	216	Ø	248	ø
55	7	87	W	119	w	185	¹	217	Ù	249	ù
56	8	88	X	120	x	186	º	218	Ú	250	ú
57	9	89	Y	121	y	187	»	219	Û	251	û
58	:	90	Z	122	z	188	¼	220	Ü	252	ü
59	;	91	[	123	{	189	½	221	Ý	253	ý
60	<	92	\	124	|	190	¾	222	Þ	254	þ
61	=	93	]	125	}	191	¿	223	ß	255	ÿ
62	>	94	^	126	~	192	À	224	à		
63	?	95	_	161	¡	193	Á	225	á		

If, For, Select, While, Until, Case

if

```
if COMANDO
then
    ...
elif COMANDO
then
    ...
else
    ...
fi
```

for, select

```
for VAR in LISTA
do
    ...
done
ou: for ((exp1;exp2;exp3))
```

while, until

```
while COMANDO
do
    ...
done
```

case

```
case $VAR in
    txt1) ... ;;
    txt2) ... ;;
    txtN) ... ;;
    *) ... ;;
esac
```

Códigos prontos

Condicionais com o IF

```
if [ -f "$arquivo" ]; then echo 'Arquivo encontrado'; fi
```

```
if [ ! -d "$dir" ]; then echo 'Diretório não encontrado'; fi
if [ $i -gt 5 ]; then echo 'Maior que 5'; else echo 'Menor que 5';
  fi
if [ $i -ge 5 -a $i -le 10 ]; then echo 'Entre 5 e 10, incluindo';
  fi
if [ $i -eq 5 ]; then echo '=5'; elif [ $i -gt 5 ]; then echo
  '>5'; else echo '<5'; fi
if [ "$USER" = 'root' ]; then echo 'Oi root'; fi
if grep -qs 'root' /etc/passwd; then echo 'Usuário encontrado'; fi
```

Condicionais com o E (&&) e OU (||)

```
[ -f "$arquivo" ] && echo 'Arquivo encontrado'
[ -d "$dir" ] || echo 'Diretório não encontrado'
grep -qs 'root' /etc/passwd && echo 'Usuário encontrado'
cd "$dir" && rm "$arquivo" && touch "$arquivo" && echo 'feito!'
[ "$1" ] && param=$1 || param='valor padrão'
[ "$1" ] && param=${1:-valor padrão}
[ "$1" ] || { echo "Uso: $0 parâmetro" ; exit 1 ; }
```

Adicionar 1 à variável $i

```
i=$(expr $i + 1)
i=$((i+1))
let i=i+1
let i+=1
let i++
```

Loop de 1 a 10

```
for i in 1 2 3 4 5 6 7 8 9 10; do echo $i; done
for i in $(seq 10); do echo $i; done
for ((i=1;i<=10;i++)); do echo $i; done
i=1 ; while [ $i -le 10 ]; do echo $i ; i=$((i+1)) ; done
i=1 ; until [ $i -gt 10 ]; do echo $i ; i=$((i+1)) ; done
```

Loop nas linhas de um arquivo ou saída de comando

```
cat /etc/passwd | while read LINHA; do echo "$LINHA"; done
grep 'root' /etc/passwd | while read LINHA; do echo "$LINHA"; done
while read LINHA; do echo "$LINHA"; done < /etc/passwd
while read LINHA; do echo "$LINHA"; done < <(grep 'root'
  /etc/passwd)
```

Curingas nos itens do comando case

```
case "$dir" in /home/*) echo 'dir dentro do /home';; esac
case "$user" in root|joao|maria) echo "Oi $user";; *) echo "Não te
  conheço";; esac
case "$var" in ?) echo '1 letra';; ??) echo '2 letras';; ??*) echo
  'mais de 2';; esac
case "$i" in [0-9]) echo '1 dígito';; [0-9][0-9]) echo '2
  dígitos';; esac
```

Caixas do Dialog

```
dialog --calendar 'abc' 0 0 31 12 1999
dialog --checklist 'abc' 0 0 0 item1 'desc1' on item2 'desc2' off
dialog --fselect /tmp 0 0
(echo 50; sleep 2; echo 100) | dialog --gauge 'abc' 8 40 0
dialog --infobox 'abc' 0 0
dialog --inputbox 'abc' 0 0
dialog --passwordbox 'abc' 0 0
dialog --menu 'abc' 0 0 0 item1 'desc1' item2 'desc2'
dialog --msgbox 'abc' 8 40
dialog --radiolist 'abc' 0 0 0 item1 'desc1' on item2 'desc2' off
dialog --tailbox /tmp/arquivo.txt 0 0
dialog --textbox /tmp/arquivo.txt 0 0
dialog --timebox 'abc' 0 0 23 59 00
dialog --yesno 'abc' 0 0
Dica1: dialog ... && echo 'Apertou OK/Yes' || echo 'Apertou
  Cancel/No'
Dica2: resposta=$(dialog --stdout --TIPODACAIXA 'abc' ...)
```

Mensagem final

Espero que você tenha aproveitado essa viagem ao interior da concha.

Espero que você tenha compreendido a importância de vários pontos que este livro prega: código legível, manutenção facilitada, compatibilidade com vários sistemas, refazer testes após cada alteração, programar pensando no usuário, ser flexível, mas não descuidar da segurança, ler a documentação.

Espero que a leitura tenha instigado sua mente, fazendo pipocar muitas ideias, despertando um desejo imenso de sair programando. Não deixe a empolgação esfriar. Se você está aí pensando em reformar um programa antigo ou criar algo novo, meu conselho é um só: **faça agora**.

Espero também que o tempo que você investiu no estudo deste livro seja plenamente recompensado. Que venham as ofertas de emprego, os aumentos salariais, as promoções de cargo, novos projetos, novas ideias, novos programas e tudo de positivo que o conhecimento pode lhe proporcionar. Sonhe, inove, realize!

De agora em diante, não diga que você faz scripts. Você aprendeu a fazer programas de verdade em shell. Valorize seus conhecimentos, pois você não é mais um scripteiro. Você agora sabe como fazer um trabalho profissional e merece ser reconhecido por isso. Em shell isso é um grande diferencial, seus programas brilharão em meio à massa de scripts malfeitos que infestam servidores e sistemas.

Nos próximos programas que você fizer, pratique os conceitos aprendidos:

– **Faça códigos limpos**. Organize as informações, alinhe os blocos, dê nomes descritivos, separe trechos distintos, explique os algoritmos com comentários esclarecedores. Faça um código bonito, uma obra de arte que foi lapidada ao extremo, onde não há mais arestas para arredondar. Seu código é seu currículo, ele diz muito sobre você e seu método de

programação. Quanto mais você investir em limpeza e legibilidade, mais profissional será considerado o seu trabalho.

– **Use chaves** para controlar as funcionalidades de seus programas. Crie tantas quantas forem necessárias. Uma vez acostumado com as chaves, você dificilmente programará sem elas. O código fica mais legível e os algoritmos mais simples e diretos. Escolha nomes descritivos para poder economizar tempo nos comentários.

– **Domine a arte da depuração** para ficar ágil no isolamento de problemas. Quanto mais acostumado a usar as técnicas de debug aqui ensinadas, mais rápido você encontrará a causa das falhas em seus programas. Não perca tempo tentando adivinhar o que aconteceu, é mais eficiente ligar o debug e ver na tela em que ponto do código o erro acontece. Não se prenda em possibilidades, vá direto ao fato.

– **Faça programas flexíveis**, que aceitem ter seu comportamento padrão facilmente modificado através de opções de linha de comando e arquivos de configuração. Todos ganham com esta comodidade. Os usuários terão um programa mais poderoso, que pode ser facilmente adaptado às suas necessidades. Você perderá menos tempo com atualizações de código, pois deixou tudo fácil para que o próprio usuário consiga personalizar o programa.

– **Pense no usuário**, sempre. Este é um investimento de retorno garantido. Sempre que puder, facilite a vida do usuário: resolva seus problemas, guie seus passos, dê dicas, deixe claras as informações. Não tenha preguiça de codificar a mais para que o usuário tenha menos trabalho. O tempo que você ganhará na diminuição de chamados de suporte vai compensar o esforço. Use os caracteres de controle para destacar mensagens importantes, puxando os olhos do usuário em direção ao que realmente importa. Faça programas realmente intuitivos com o dialog, apresentando menus amigáveis que guiarão o usuário nos passos necessários para completar as tarefas. Peça confirmação antes de executar tarefas perigosas, informe quando obtiver sucesso, informe que está fazendo algo para o usuário saber que o programa não está travado. Prefira a clareza de informações à economia de telas.

– **Use banco de dados** para guardar informações úteis. Pense em quais

tipos de dados que seu programa poderia se lembrar em uma próxima execução. Guarde informações que o usuário tenha digitado, localização de arquivos que foram abertos, lista de tarefas que foram executadas, horário de execução do programa e detalhes sobre o ambiente. Você já tem o gerenciador, então é fácil guardar e obter qualquer tipo de informação.

– **Desbrave a Internet**. Há um mundo de informações na rede e seu programa pode aproveitar-se delas. Baixe os dados automaticamente, manipule-os com as expressões regulares e apresente o resultado de maneira limpa e facilmente legível, eliminando a necessidade de acessar sites pesados. Rompa as barreiras e coloque seu programa rodando na Internet como CGI, sendo acessível globalmente, evitando dores de cabeça com instalação, pré-requisitos e distribuição. Sonhe alto, voe alto. Dê condições para que seus programas ganhem o mundo.

– **Domine as ferramentas** básicas do sistema e os comandos builtin do Bash. Porém, faça isso gradualmente. O ideal não é você saber de cabeça quais são todos os comandos e suas opções, mas sim saber onde procurar quando precisar. Acostume-se a ler a documentação disponível, as man pages, a tela de ajuda. Mas não precisa fazer isso por antecipação, descubra os comandos à medida que precisar deles. Assim você não perderá tempo estudando algo que pode ser que nunca vá utilizar. Não lote sua caixa com ferramentas que só irão juntar poeira.

– **Torne-se um guru em shell**. Estude com atenção o Capítulo 13 – Dicas preciosas, praticando em seus programas até que seja natural escrever códigos portáveis que funcionarão em uma grande quantidade de sistemas diferentes, porém sem prejudicar a eficiência da execução. Dê atenção especial ao apêndice que analisa as Funções ZZ e as várias técnicas empregadas nelas, pois são muitos anos de experiência registrados naquelas linhas. Baixe as funções, estude seu código e tente aplicar os mesmos conceitos em seus próprios programas. É preciso fazer uma grande quantidade de programas para atingir um nível elevado de experiência e conhecimento, então o maior dos conselhos é: programe incansavelmente!

Visite o site do livro em `www.shellscript.com.br`.

```
exit 0
```


5ª EDIÇÃO
Revisada e ampliada
EXPRESSÕES REGULARES
UMA ABORDAGEM DIVERTIDA
(bb|[^b]{2})
[ER]
novatec
Aurelio Marinho Jargas
www.aurelio.net

Expressões Regulares - 5ª edição

Jargas, Aurelio Marinho
9788575224755
248 páginas

Você procura uma sigla em um texto longo, mas não lembra direito quais eram as letras. Só lembra que era uma sigla de quatro letras. Simples, procure por [A-Z]{4}. Revisando aquela tese de mestrado, você percebe que digitou errado o nome daquele pesquisador alemão famoso. E foram várias vezes. Escreveu Miller, Mueller e Müler, quando na verdade era Müller. Que tal corrigir todos de uma vez? Fácil, use a expressão M(i|ue|ü)ll?er. Que tal encontrar todas as palavras repetidas repetidas em seu texto? Ou garantir que há um espaço em branco após todas as vírgulas e os pontos finais? Se você é programador, seria bom validar dados em um único passo, não? Endereço de e-mail, número IP, telefone, data, CEP, CPF... Chega de percorrer vetores e fazer checagens "na mão". Estes são exemplos de uso das Expressões Regulares, que servem para encontrar rapidamente trechos de texto informando apenas o seu formato. Ou ainda pesquisar textos com variações, erros ortográficos e muito mais! Visite o site do livro: www.piazinho.com.br

Uma introdução para você colocar a mão na massa

Git

Guia Prático

O'REILLY
novatec

Richard E. Silverman

Git

Silverman, Richard E.
9788575227411
208 páginas

Este guia prático é a sua companhia perfeita para o trabalho com o Git, o sistema distribuído de controle de versões. O livro fornece uma introdução concisa e de fácil leitura para os novos usuários do Git, assim como uma referência para os comandos e procedimentos comuns para as pessoas que já possuam experiência com o uso do Git. Escrito para a versão 1.8.2 do Git, este útil guia orientado a tarefas está organizado em torno das funções básicas de controle de versão que você precisará, como executar commits, resolver erros, realizar fusões (merge) e pesquisar pelo histórico. •Veja os estados anteriores do seu projeto •Aprenda o básico sobre a criação e alterações em um repositório •Crie branches para que muitas pessoas possam trabalhar em um mesmo projeto ao mesmo tempo •Execute a fusão de branches e concilie as alterações entre eles •Execute a clonagem de um repositório existente e compartilhe as alterações com comandos push e pull •Examine e altere o histórico de commits do seu repositório •Acesse repositórios remotos, usando diferentes protocolos de rede •Veja vários guias passo a passo para executar tarefas comuns

O'REILLY

Padrões para Kubernetes

Elementos reutilizáveis no design de aplicações nativas de nuvem

Bilgin Ibryam
Roland Huß

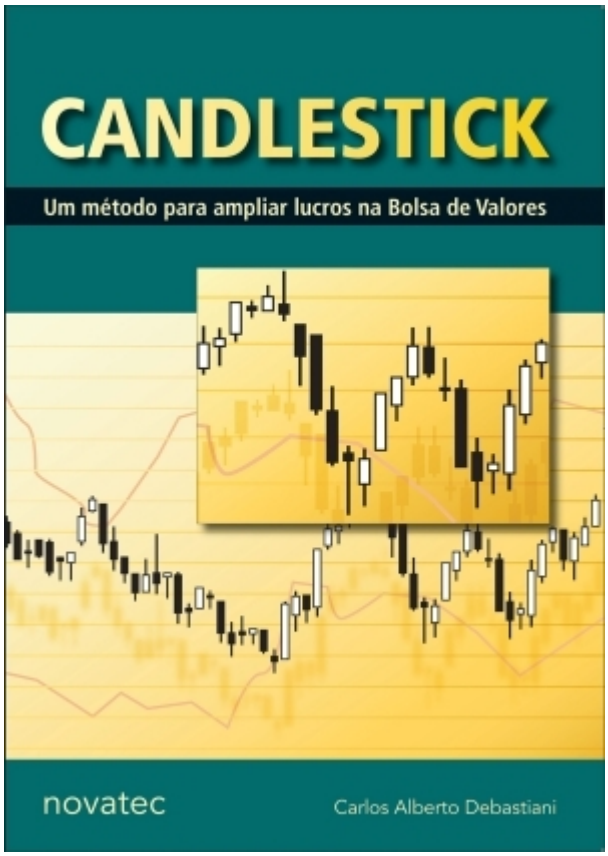
novatec

Padrões para Kubernetes

Ibryam, Bilgin
9788575228159
272 páginas

O modo como os desenvolvedores projetam, desenvolvem e executam software mudou significativamente com a evolução dos microsserviços e dos contêineres. Essas arquiteturas modernas oferecem novas primitivas distribuídas que exigem um conjunto diferente de práticas, distinto daquele com o qual muitos desenvolvedores, líderes técnicos e arquitetos estão acostumados. Este guia apresenta padrões comuns e reutilizáveis, além de princípios para o design e a implementação de aplicações nativas de nuvem no Kubernetes. Cada padrão inclui uma descrição do problema e uma solução específica no Kubernetes. Todos os padrões acompanham e são demonstrados por exemplos concretos de código. Este livro é ideal para desenvolvedores e arquitetos que já tenham familiaridade com os conceitos básicos do Kubernetes, e que queiram aprender a solucionar desafios comuns no ambiente nativo de nuvem, usando padrões de projeto de uso comprovado. Você conhecerá as seguintes classes de padrões: • Padrões básicos, que incluem princípios e práticas essenciais para desenvolver aplicações nativas de nuvem com base em contêineres. • Padrões comportamentais, que exploram conceitos mais específicos para administrar contêineres e interações com a

plataforma. • Padrões estruturais, que ajudam você a organizar contêineres em um Pod para tratar casos de uso específicos. • Padrões de configuração, que oferecem insights sobre como tratar as configurações das aplicações no Kubernetes. • Padrões avançados, que incluem assuntos mais complexos, como operadores e escalabilidade automática (autoscaling).

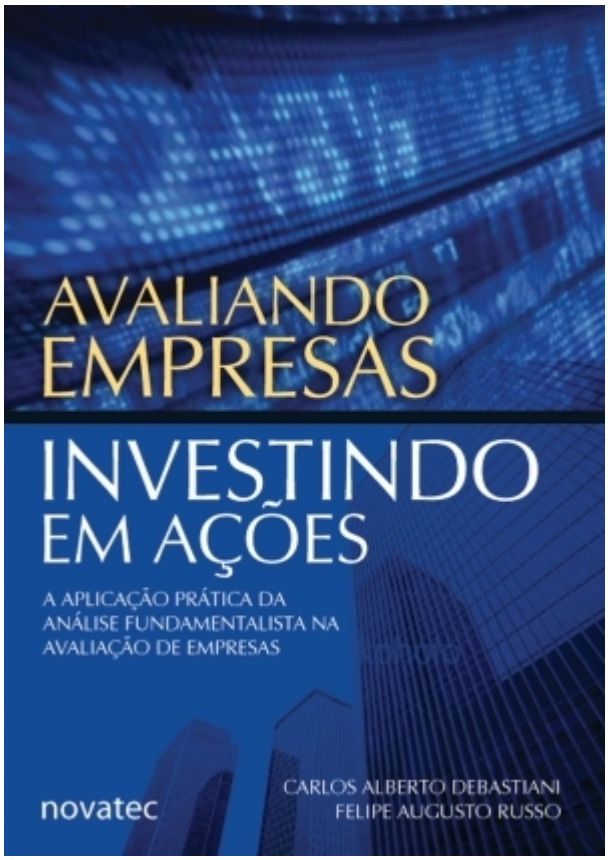

CANDLESTICK
Um método para ampliar lucros na Bolsa de Valores
novatec
Carlos Alberto Debastiani

Candlestick

Debastiani, Carlos Alberto
9788575225943
200 páginas

A análise dos gráficos de Candlestick é uma técnica amplamente utilizada pelos operadores de bolsas de valores no mundo inteiro. De origem japonesa, este refinado método avalia o comportamento do mercado, sendo muito eficaz na previsão de mudanças em tendências, o que permite desvendar fatores psicológicos por trás dos gráficos, incrementando a lucratividade dos investimentos. Candlestick – Um método para ampliar lucros na Bolsa de Valores é uma obra bem estruturada e totalmente ilustrada. A preocupação do autor em utilizar uma linguagem clara e acessível a torna leve e de fácil assimilação, mesmo para leigos. Cada padrão de análise abordado possui um modelo com sua figura clássica, facilitando a identificação. Depois das características, das peculiaridades e dos fatores psicológicos do padrão, é apresentado o gráfico de um caso real aplicado a uma ação negociada na Bovespa. Este livro possui, ainda, um índice resumido dos padrões para pesquisa rápida na utilização cotidiana.

Compre agora e leia


AVALIANDO
EMPRESAS
INVESTINDO
EM AÇÕES
A APLICAÇÃO PRÁTICA DA
ANÁLISE FUNDAMENTALISTA NA
AVALIAÇÃO DE EMPRESAS
novatec
CARLOS ALBERTO DEBASTIANI
FELIPE AUGUSTO RUSSO

Avaliando Empresas, Investindo em Ações

Debastiani, Carlos Alberto
9788575225974
224 páginas

Avaliando Empresas, Investindo em Ações é um livro destinado a investidores que desejam conhecer, em detalhes, os métodos de análise que integram a linha de trabalho da escola fundamentalista, trazendo ao leitor, em linguagem clara e acessível, o conhecimento profundo dos elementos necessários a uma análise criteriosa da saúde financeira das empresas, envolvendo indicadores de balanço e de mercado, análise de liquidez e dos riscos pertinentes a fatores setoriais e conjunturas econômicas nacional e internacional. Por meio de exemplos práticos e ilustrações, os autores exercitam os conceitos teóricos abordados, desde os fundamentos básicos da economia até a formulação de estratégias para investimentos de longo prazo.

Compre agora e leia