BLAU ARAUJO

PEQUENO MANUAL DO PROGRAMADOR GNU/BASH

PRIMEIRA EDIÇÃO

Blau Araujo

Pequeno Manual do Programador GNU/Bash

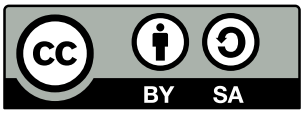
PRIMEIRA EDIÇÃO

Pequeno Manual do Programador GNU/Bash

PRIMEIRA EDIÇÃO

16 de novembro de 2020

ISBN: 978-65-00-12986-1



Um apelo sincero...

Este livro digital é uma produção independente feita apenas com vontade, esforço, dedicação e muito amor pela causa de transmitir um pouco de conhecimento e cultura de liberdade. O apoio financeiro foi mínimo e bem-vindo. O que jamais me faltou, porém, foi o incentivo de amigos e ex-alunos.

Por isso, eu faço um apelo:

Se você recebeu este livro por outros meios que não os canais que eu disponibilizei para venda, considere fazer uma doação de qualquer valor através de uma das formas de apoio abaixo.

PicPay
https://picpay.me/blauaraujo

PayPal
https://www.paypal.com/donate?hosted_button_id=T4EB496NUTKM6&source=url

PagSeguro
https://pag.ae/7Vu5Cocjo

Muito obrigado!
Blau Araujo

Agradecimentos...

Meu muito obrigado aos amigos que, de várias formas, me ajudaram e incentivaram a concluir esta primeira edição. São tantos que não sei se vou conseguir citar todos aqui, mas vou tentar...

Leandro Ramos
Paulo Kretcheu
Djalma Valois
Joseano Sousa
José Almeida
André Amaral
Wilson Faustino
César Faustino
Vicente Marçal
Qobi Ben Nun
Antônio R. Souza
Anderson Antônio
Dan Morris
Marcelo Amedi
Simplex
Daniel Lara
Adriano "Pandatitan"
Diego Sarzi
Sérgio Correia
Marcelo Gondim
Bruna Wojtenko
Adail Carvalho
Raul Goes
Talles Makoto
Keilon Araujo
Ayr Müller
André Paula
João Jota
Igor Muzetti
Fabrício Fernandes
Jeferson Stefani
Marcelo Alves
Flavius Lira
Leandro Vidal
Max Deivys
Olnei Araujo
Wanderson Carneiro
Áleson Medeiros
Fernando Eleutério
Marcello de Souza
Antônio Vitor Silva
Guilherme Xavier
Marcelo Pinheiro
Luiz Devillarte
André Moretto
Wallace Paisano
Hugo Xavier
David Clepf

E a todos os grandes amigos dos grupos do Telegram...

Todas as comunidades Debian brasileiras

GNU/Linux – Grupo de estudos

Comunidade Fedora Brasil

debxp/comunidade

Os Programadores

Pinguim Criativo

Curso GNU

Gnewlinux

BashBR

Dedicatória...

Este livro é dedicado à mulher que já me atura há 20 anos, sempre me impelindo a ir mais longe e a tentar novamente. Por uma dessas gratas coincidências, esta edição foi terminada justamente na data em que ela completou mais um ano de vida, o que me deixa duplamente feliz!

Muito obrigado, *Nena*, meu amor!

Índice Geral

Prefácio, por Paulo Kretcheu

Quando meu novo amigo Blau Araújo me pediu para escrever esse prefácio, de pronto aceitei.

Que coragem, hein?!

Que tarefa!

Que tarefa especial, honrosa e de responsabilidade.

Bem, vamos lá.

Blau Araújo, ou simplesmente Blau, é uma pessoa muito gentil, um professor excelente, superdidático e dedicado à arte.

Isso mesmo, ensinar é uma arte e é para poucos. Em suas aulas fica fácil perceber todos esses atributos e o conhecimento sobre os temas também deixa claro a dedicação nos estudos e a abertura e humildade para sempre aprender mais.

Blau é um profissional de muitas facetas: de músico a tradutor, de programador a professor. Tudo isso junto e misturado dão a ele a sensibilidade e competência necessárias para um professor e escritor que consegue catalisar o aprendizado de seus alunos e leitores.

No *Pequeno Manual do Programador GNU/Bash*, o texto, como você vai poder acompanhar em breve, flui com a mesma tranquilidade de suas aulas. Ele pega você pela mão e vai ajudando na sua caminhada para um conhecimento consistente. A leitura poderá fazer de você o especialista que sempre quis ser ou apenas um interessado e mais eficiente administrador de sistemas GNU/Linux ou qualquer outro *unix-like*. A cada capítulo, você vai poder ir aprofundando a compreensão sobre cada tema e se sentindo mais confiante e curioso para aprender ainda mais.

Espero que, como eu, você desfrute dessa obra de leitura tranquila e divertida.

Boa leitura, bons estudos, boa diversão.

Paulo Kretcheu
24 de outubro de 2020

Introdução e convenções

Se você achou que encontraria um manual com todas as informações e exemplos de uso de todos os comandos e recursos do Bash, você entendeu errado o nome deste livro. Este é um manual *do programador* que utiliza os conceitos da Filosofia Unix, conforme implementada pelo Projeto GNU, para programar em Bash. Pense comigo: qual o sentido escrever mais um livro falando dos mesmos comandos e utilitários do shell padrão do sistema operacional GNU quando já temos as obras de escritores e programadores geniais do calibre de um *Júlio Neves* ou de um *Aurélio Jargas*? Boa parte da minha motivação para esse empreendimento, aliás, vem justamente da paixão pelo shell que eu compartilho com esses mestres – e talvez esta seja a única coisa que nós tenhamos em comum.

Ironicamente, com eles, eu nunca aprendi a programar em shell! Para isso, bastariam os manuais, um terminal e muitos litros de café. No fim das contas, cada um ao seu jeito, o que eles realmente me ensinaram foi a moldar a minha forma de pensar, a transitar de forma segura entre o que eu já conhecia e, principalmente, a desenvolver uma verdadeira paixão por compartilhar o pouco que sei. Isso não se aprende em livros ou manuais *de programação*, apenas em manuais *de programadores*.

Com toda humildade de quem reconhece a sua própria estatura diante de gigantes, é isso que eu me proponho a fazer neste livro. Mais do que oferecer mais uma fonte de consulta para o seu arsenal de programador, o que eu realmente quero é convidá-lo para uma jornada de descobertas, onde cada novo conceito, cada comando, cada linha de código seja uma surpresa e um motivo a mais para você se apaixonar pela possibilidade de programar em Bash.

Sobre a organização dos tópicos

Quando comecei a ensinar linguagens de programação, umas das coisas que logo chamaram a minha atenção foi o fato de que não há como explicar um conceito básico sem recorrer a exemplos que envolvam os conceitos mais avançados. Um simples *"Olá, mundo"*, em qualquer linguagem, sempre fará referência a uma ou duas coisas que só poderão ser abordadas a fundo bem mais adiante numa sequência progressiva. Por este motivo, e também por acreditar que a melhor forma de ensinar uma nova linguagem é dando contexto ao que se pretende demonstrar, este livro foi pensado para ser lido da forma que você estabelecer como ideal para o seu momento.

A minha visão pessoal do que seria um *aprendizado progressivo* acabou restrita à organização dos capítulos no índice. Fora isso, tudo está entrelaçado e, salvo eventuais distrações, bem referenciado para que você possa encontrar as informações adicionais sobre o que estiver lendo. Mas, se não estiver disposto a ler tudo do início ao fim, tudo bem! Você pode pensar neste livro como se fossem três, cada um com seus próprios objetivos e abordagens:

> No *capítulo 1 – O que é um shell*, a ideia é apresentar a você o seu novo ambiente de desenvolvimento.
>
> No *capítulo 2 – O Bash como linguagem de programação*, além de abordarmos as peculiaridades do desenvolvimento em Bash, nós trabalharemos em um pequeno projeto: um *script para criar scripts*, e teremos inúmeras oportunidades para tomar contato, na prática, com quase tudo que precisamos saber para começarmos a programar em Bash.
>
> Do *capítulo 3* em diante, entra em cena o nosso lado mais "manual". A partir daqui, você poderá ler os tópicos em sequência ou, se achar melhor, pode utilizá-los como fonte de consulta quando precisar de alguma informação – você decide.

Instale o Bash

Felizmente, não há como aprender o Bash sem que ele esteja instalado e disponível para os nossos experimentos (seria muito chato se isso fosse possível). Se você usa o sistema operacional GNU/Linux, que é o foco deste livro, é bem provável que ele já seja o seu shell padrão. Mas, se não for, todas as distribuições oferecem formas simples de instalá-lo. De qualquer forma, se for o caso, os procedimentos podem variar muito e eu poderia atrapalhar mais do que ajudar. Então, a minha sugestão é que você consulte a documentação ou as comunidades de ajuda do seu sistema operacional.

Convenções tipográficas

No texto, tudo que aparecer em *itálico* representa algo que merece a sua atenção por algum motivo – podem ser termos ou expressões que serão explicados posteriormente, em outros idiomas, ou apenas eu tentando aguçar a sua curiosidade.

Comandos, *variáveis*, *palavras-chave*, *diretórios*, *nomes de arquivos* e *atalhos de teclado* serão formatados `desta forma` no texto – com caracteres mono espaçados e uma leve borda cinza.

Nos exemplos de *comandos executados no terminal*, haverá sempre um balão e a representação de um *prompt* (`:~$` ou algo parecido):

```
:~$ uma linha de comando
```

Se o *prompt* não aparecer, significa que o código do exemplo é parte de um *script* ou que o conteúdo do balão é apenas um *diagrama*:

```
uma linha de comando no script
```

Por último, fique atento às notas e observações que aparecerem nos balões destacados da forma abaixo:

> ***Importante!*** *Fique atento às notas e observações!*

A filosofia Unix

GNU não é Unix[1]*,* mas foi desenvolvido como um sistema operacional livre que, além das similaridades técnicas, adota alguns dos princípios mais importantes que a história da computação já produziu: a *Filosofia Unix*. Isso merece uma menção especial nesta introdução por um motivo muito simples: *o Bash foi projetado tendo em mente o uso e a criação de programas que atendessem aos princípios da Filosofia Unix*.

Em outras palavras, nós podemos criar sistemas complexos em Bash, mas nunca é uma boa ideia fazê-los monolíticos e cheios de recursos. O ideal é que a complexidade, se houver, seja o resultado de vários pequenos programas especializados trabalhando juntos e trocando dados entre si. Mais do que uma técnica de desenvolvimento, este é *o modelo mental* que faz toda diferença quando o nosso ambiente de desenvolvimento, a nossa coleção de bibliotecas e até a nossa IDE são o próprio sistema operacional!

O mais interessante, porém, é que você não precisa conhecer nada sobre a Filosofia Unix para começar a desenvolver em Bash utilizando seus princípios. É engraçado (e parece coisa de *Jedi*), mas, depois de um tempo, a gente "sente" o jeito do shell. Por exemplo, assim que você descobre que existe um comando `alias` e aprende a usá-lo, você quer fazer *aliases* para tudo! Aliás, a viagem fica ainda mais divertida quando você, já insatisfeito com o `alias`, aprende a criar *funções* – aí, não tem quem segure! Quando menos percebemos, todos os "comandos" que nós utilizamos no terminal são os *aliases,* as funções e os scripts que nós mesmos escrevemos. Neste ponto, o

1 **GNU** é um acrônimo recursivo para "GNU Não é Unix" (*GNU is Not Unix*, em inglês).

sistema operacional é nosso! A partir daí, é só conhecer um pouco mais sobre os recursos do Bash como shell e como linguagem. O principal, que é a forma de pensar, você já adquiriu sem perceber.

Mas, se o *bichinho do bashismo* ainda não mordeu você, é nisso que eu espero poder contribuir com este livro.

Bons estudos!
Blau Araujo

1 – O que é um shell

Mesmo que você nunca tenha criado nenhum script, mesmo que não saiba nada de programação, se você já usou um terminal de qualquer sistema operacional GNU/Linux, um terminal do seu macOS, ou até o console do Windows™, você já trabalhou com algum tipo de shell.

De forma simplificada, podemos entender o shell como uma camada que envolve o sistema operacional metaforicamente como uma casca ou uma concha (em inglês, *shell*[2]). Esta camada é responsável por fazer uma interface entre o usuário e as funcionalidades do núcleo do sistema – o *kernel*[3]. Embora seja uma ideia bastante difundida, o fato é que raramente um shell interage diretamente com o kernel, sendo bem mais comum que isso aconteça através de uma API[4].

Na sua função principal, a de interpretador de comandos, o shell recebe as instruções do usuário através de um dispositivo de entrada, avalia e processa essas instruções, repassa ao sistema operacional o que deve ser feito, retorna algum tipo de resultado para o usuário e aguarda o próximo comando.

Nem sempre o retorno vindo do sistema é algo que poderá ser visto no terminal. Às vezes, o comando do usuário implica apenas na execução de outro programa, por exemplo, e o único retorno que se pode esperar nesses

2 O termo **shell** foi primeiro utilizado no projeto MULTICS, cujo objetivo era criar um sistema operacional inovador. O projeto foi inciado como uma ação colaborativa entre o MIT, a General Electric e a Bell Labs. Por fim, esta última se afastou do grupo e acabou criando o UNIX.

3 O **kernel** é o componente de software central da maioria dos sistemas operacionais, e sua função é basicamente fazer uma ponte entre os demais programas e o processamento de suas instruções no nível do hardware.

4 **API** - Interface de Programação de Aplicação (do inglês, *Application Programming Interface*). É um conjunto de rotinas estabelecidas por um programa para que suas funcionalidades e serviços sejam utilizados por outro software.

casos é se foi possível executar o programa ou se ele terminou a sua execução com sucesso ou com erro.

1.1 – Iniciando e terminando sessões do shell

No fim das contas, apesar do papel especial que desempenha no sistema operacional, o shell não deixa de ser um programa como outro qualquer. Como qualquer programa, ele terá que ser iniciado de alguma forma e, em algum momento, precisará ser terminado.

1.1.1 – Iniciando o shell

Nos sistemas GNU/Linux, o shell parece poder ser iniciado de três formas:

- Ao abrirmos um terminal ou um console;
- Ao executarmos um script;
- Ou invocando diretamente seu executável.

Mas isso é apenas uma impressão causada pela experiência de utilização do sistema operacional. Na verdade, sempre que um shell é iniciado, significa que algo invocou o seu executável de alguma forma. Quando abrimos um terminal, por exemplo, a primeira coisa que vemos é o prompt do shell. Isso significa que, ao ser iniciado, o terminal invocou o executável de um shell.

> ***Pense nisso:*** *o emulador de terminal é um programa cuja única finalidade é executar um shell.*

Como existe quase que uma "crise de identidade" entre o terminal e o shell – o que pode ser confirmado por frases populares como: *"digite no terminal"*, ou *"execute no terminal"* –, isso pode ser mais difícil de notar observando apenas o

que acontece quando abrimos um terminal. Neste aspecto, os scripts podem nos dar uma visão bem melhor do que estamos tentando dizer.

1.1.2 – O que é um script

De forma bem simplificada, já que ainda estamos falando de shells em geral, um script é um arquivo contendo um ou mais comandos que nós queremos que o shell execute. Em outras palavras...

> ***Definição curta e limitada:*** *um script é um código composto apenas de instruções que o shell é capaz de interpretar e executar.*

Sendo assim, a menos que ele tenha permissão de execução (falaremos sobre isso mais adiante) e seja invocado diretamente a partir de um terminal, o executável do shell terá que ser invocado de alguma forma. Então, quando scripts e comandos precisam ser executados a partir de outros programas (ou quando queremos que eles sejam interpretados por um shell diferente do que está em uso no momento), o shell precisará ser invocado explicitamente e o nome do script ou a lista de comandos serão informados como parâmetros:

Exemplo 1.1 – Executando um script com um shell específico.

```
sh /caminho/nome-do-script

bash /caminho/nome-do-script
```

Exemplo 1.2 – Executando comandos com um shell específico.

```
sh -c 'lista de comandos'

bash -c 'lista de comandos'
```

No primeiro exemplo, nós estamos dizendo que o script deve ser interpretado e executado pelo shell cujo executável se chama `sh`. Já no segundo, o nome do executável é `bash`.

> ***Isso não é regra!*** *Outros shells podem ter outras formas de receber os nomes de scripts e as listas de comandos.*

Se quisermos que o script seja executado diretamente a partir da invocação do nome do seu arquivo, o shell a ser utilizado terá que ser informado logo na primeira linha do código:

Exemplo 1.3 – Linha do interpretador de comandos.

```
#!/bin/bash
```

Essa linha é chamada de *shebang*, *hashbang* ou, como eu prefiro, *linha do interpretador de comandos*. Como você pode notar, trata-se essencialmente da invocação do executável do shell.

> *Os caracteres `#!` no começo da primeira linha de um arquivo com permissão de execução são interpretados pelo kernel como uma instrução para executar o programa indicado na linha da hashbang utilizando o restante do conteúdo do arquivo como dados de entrada. Do ponto de vista do shell, por sua vez, o caractere `#` indica um "comentário", logo, tudo que vem depois dele é ignorado pelo interpretador.*

1.1.3 – Dando permissão de execução

Além da linha do interpretador, para ser executado a partir da invocação de seu nome, o arquivo do script precisa ter o atributo de permissão para execução, o que é feito com o comando `chmod`:

```
chmod +x nome-do-arquivo
```

1.1.4 – Invocando o shell na linha de comandos

Um shell é um programa como outro qualquer – no sentido de que ele pode ser executado a partir da invocação de seu nome. Mas, o que aconteceria se nós tentássemos invocar o executável de um shell diretamente na linha de comandos?

Experimente:

Exemplo 1.4 – Executando o shell 'sh' na linha de comandos.

```
:~$ sh
$
```

O que aconteceu no exemplo foi exatamente o que esperávamos: um outro shell foi iniciado (no caso, o shell `sh`), mas há algumas outras coisas acontecendo aqui que merecem a nossa atenção.

Para começar, o shell anterior não foi terminado. Ele ainda está em execução no mesmo terminal, só não está acessível enquanto durar a sessão do shell recém-iniciado. Então, neste momento, há dois shells diferentes em execução, o que pode ser verificado com o utilitário `ps`:

Exemplo 1.5 – Utilizando o utilitário 'ps' para listar os processos no terminal.

```
$ ps
  PID TTY          TIME CMD
 4525 pts/0    00:00:00 bash
 4533 pts/0    00:00:00 sh
 4676 pts/0    00:00:00 ps
```

> *Segundo o manual, quando invocado sem opções, o utilitário `ps` exibe uma lista com todos os processos ativos iniciados pelo usuário no terminal.*

Na primeira linha da lista exibida, nós podemos ver o processo iniciado há mais tempo, que é justamente o shell anterior (no nosso caso, o `bash`) ainda em execução, enquanto que a linha seguinte mostra o shell que estamos utilizando no momento (`sh`).

1.1.5 – Encerrando o shell

Para terminar o shell `sh` que iniciamos no *exemplo 1.4*, nós podemos executar o comando interno `exit`:

Exemplo 1.6 – O comando 'exit'.

```
$ exit
:~$
```

Como podemos perceber, o prompt do shell anterior (`bash`) foi restaurado, mas ainda podemos conferir se o shell `sh` foi realmente encerrado:

Exemplo 1.7 – Executando o utilitário 'ps' novamente.

```
:~$ ps
 PID TTY          TIME CMD
 4525 pts/0    00:00:00 bash
 5461 pts/0    00:00:00 ps
```

De fato, o shell `sh` não está mais na lista de processos ativos!

> ***Outra forma de terminar um shell:*** *quando o shell é iniciado a partir de um login, é possível terminá-lo e encerrar, ao mesmo tempo, toda a sessão do usuário com o comando* `logout`.

1.2 – O prompt de comandos

Quando fazemos o login no console, ou quando abrimos um terminal no ambiente gráfico, o shell é iniciado e nós somos apresentados a um conjunto de caracteres, símbolos e um cursor no local onde os comandos deverão ser digitados. Isso significa que o shell está pronto e esperando pelos nossos comandos.

É justamente desse estado de *prontidão* do shell que vem a expressão *prompt de comandos*, utilizada para designar o ponto de entrada dos comandos no terminal.

O que aparece no prompt depende do shell que está em execução e da forma como ele foi configurado para ser exibido. O shell `csh` (*C Shell*), por exemplo, geralmente exibirá um `%` em seu prompt, enquanto o Dash (*Debian Almquist Shell*), utilizado como shell `sh` no Debian GNU/Linux, exibirá um `$`.

Mas as utilidades do prompt não param por aí. Pode ser que outras informações importantes estejam configuradas para serem exibidas, como é o caso, por exemplo, de um login feito como usuário *root*.

1.2.1 – Usuário administrativo (root)

Em sistemas operacionais *unix-like*, o usuário *root* é uma conta especial que tem permissões para executar qualquer tipo de ação no sistema – inclusive, destruí-lo completamente! Então, nada mais natural do que recebermos algum tipo de indicação quando estivermos trabalhando logados como usuário *root*.

No prompt, esta indicação geralmente é feita com o símbolo `#` (cerquilha) no lugar do `$` ou do símbolo que for utilizado para indicar o login de um usuário comum.

> ***Importante!*** *Fique atento ao seu prompt e, a menos que seja orientado a fazer o contrário, jamais execute os nossos exemplos e experimentos como usuário root!*

1.2.2 – Mais informações no prompt

Além de informar se estamos logados como usuários comuns ou *root*, o prompt também pode ser configurado para informar o diretório em que estamos trabalhando. Se for este o caso, um til (`~`) representará a sua pasta pessoal de usuário, cujo caminho no sistema de arquivos é:

```
/home/seu_nome_de_usuário
```

1.2.3 – Um pequeno desvio – o tal do til (~)

O til (`~`) é uma das inúmeras *expansões do shell*, um dos assuntos mais importantes deste livro (*capítulo 4*) e para quem quer realmente entender como o shell funciona. Nas condições corretas, ele será substituído pelo caminho da sua pasta pessoal antes de um comando ser executado. Observe o exemplo:

Exemplo 1.8 – Expandindo o til.

```
:~$ echo ~
/home/blau
```

O `echo` é um comando interno (*builtin*) do Bash, que serve para exibir no terminal uma *string* (cadeia de caracteres) que for passada para ele como argumento. Mas, antes que o comando seja executado, o shell fará uma busca por símbolos especiais que indiquem que algo precisa ser “trocado”. No caso do exemplo, ele encontrará o til, e entenderá que aquele símbolo precisa ser

substituído pelo nome da pasta pessoal do usuário (`/home/blau`, no meu caso).

Deste modo, o comando `echo` sequer chega a "ver" o til digitado na linha de comandos. Para ele, o argumento passado é apenas a minha pasta pessoal, e é assim que funcionam todas as outras expansões.

1.2.4 – Voltando ao prompt

No prompt, o til também representa a nossa pasta de usuário, mas ali é só isso, não há uma expansão, ele só está encurtando o tamanho do prompt. Isso é necessário porque, a partir da pasta de usuário, nós podemos navegar por outras pastas ligadas a ela, e o prompt geralmente é configurado para exibir em que pasta estamos no momento.

Então, se eu entrar na pasta `Documentos` e, dentro dela, eu entrar na pasta `clientes`, o prompt será alterado para exibir todo o caminho a partir da minha pasta pessoal: `:~/Documentos/clientes$`.

1.2.5 – A variável PS1

O prompt é configurado a partir de uma *variável de ambiente* chamada `PS1` (*Prompt String 1*, em inglês), que é uma entre quatro variáveis que definem algum tipo de prompt. Mas você deve estar se perguntando: "*o que é uma variável de ambiente?*"

Nós veremos isso mais a fundo no *capítulo 3*, mas pense numa variável como uma gaveta onde você pode guardar algum tipo de informação. Sempre que precisar dessa informação, basta abrir a gaveta que ela estará lá. Quando trabalhamos com o shell, ou em qualquer linguagem de programação, nós podemos criar várias dessas gavetas para as mais diversas finalidades (seja no terminal ou em scripts), e o shell faz a mesma coisa em várias situações.

Especificamente na inicialização do shell (quando abrimos um terminal ou fazemos um login no console), algumas variáveis serão criadas para definir inúmeros aspectos do ambiente em que ele será executado – são as *variáveis de ambiente*. Uma dessas variáveis de ambiente é justamente a variável `PS1`, e a informação que ela armazena é uma *string* que representa o que será exibido no prompt de comando.

Na maioria das distribuições GNU/Linux, o prompt é configurado para exibir algo assim:

```
blau@enterprise:~$
```

1.2.6 – Caracteres de comando do prompt

Cada um dos componentes do prompt acima é definido na string armazenada na variável `PS1` a partir de um código especial iniciado com `\` (*barra invertida* ou *escape*) representado por um caractere que equivale a um comando específico a ser executado na expansão da string que formará o prompt.

Veja na tabela:

`\$`	Caractere do tipo de usuário: normal ($) ou root (#).
`\w`	Caminho (*working directory*).
`\h`	Nome da máquina na rede (*hostname*).
`\u`	Nome do usuário.

Como `@` e `:` são apenas caracteres separadores literais, uma configuração para resultar no prompt padrão mostrado acima (sem levar em conta detalhes como cores e estilos do texto) poderia ser feita assim:

Exemplo 1.9 – Atribuindo uma string à variável 'PS1'.

```
PS1='\u@\h:\w\$ '
```

> ***Atenção!*** *Observe que a string está entre aspas simples. Isso é importante para evitar que o shell aplique alguma das suas expansões antes da string ser atribuída a* `PS1`*. Mais sobre aspas no tópico 4.3 – Como funcionam as aspas.*

Se quisermos testar como fica a aparência da configuração acima, existem algumas opções. Uma delas é simplesmente executar a linha da configuração no terminal:

Exemplo 1.10 – Alterando o valor em 'PS1'.

```
:~$ PS1='\u@\h:\w\$ '
blau@enterprise:~$
```

O sinal de igual (`=`) é um dos operadores do shell (*capítulo 6*), e é responsável por realizar atribuições de valores a variáveis. Portanto, note que a linha da configuração do prompt é um comando do shell como outro qualquer.

No exemplo, quando o comando de atribuição foi executado, a alteração do prompt foi imediata, mas o seu efeito será limitado ao shell em execução no momento, ou seja, não afetará sessões do shell iniciadas posteriormente.

Para que a alteração fosse definitiva, seria preciso editar um dos arquivos de configuração do Bash, o que pode ser feito apenas para o usuário editando o arquivo `.bashrc`, um arquivo oculto (iniciado com ponto) que fica na nossa pasta pessoal. Se fosse este o caso, bastaria abrir o arquivo `.bashrc` e incluir a linha do comando de atribuição ao final do texto, mas a mudança só teria efeito a partir de duas condições: reiniciando o terminal ou executando:

Exemplo 1.11 – Aplicado alterações no arquivo '~/.bashrc'.

```
:~$ source ~/.bashrc
```

O comando `source` é outro comando interno (*builtin*) do Bash e serve para executar na sessão corrente do shell os comandos em um arquivo. Então, tudo que o comando do *exemplo 1.11* faz é executar o conteúdo do arquivo `.bashrc` na sessão atual do Bash, inclusive a definição do nosso prompt.

Aliás, se você executou o comando do *exemplo 1.10* apenas para ver como ficaria o seu prompt e quer uma forma de restaurar as configurações originais, basta fazer um outro `source` do `.bashrc`, como fizemos no *exemplo 1.11*.

> ***Dica:*** *nós podemos abreviar o comando* `source` *utilizando um ponto (*`.`*) em seu lugar:* `. ~/.bashrc`.

Uma coisa interessante sobre os caracteres de controle que utilizamos na configuração do prompt, é que eles só serão executados sob certas condições. Por isso, nós temos que recorrer a uma das expansões do shell se quisermos visualizar um prompt sem que isso afete a exibição atual. Antes, porém, precisamos descobrir como os valores das variáveis podem ser acessados.

1.2.7 – Expandindo valores de variáveis

Voltando à analogia das gavetas, nossas variáveis precisam de um puxador. No Bash, esse puxador é o símbolo `$` (não confunda com o `$` do prompt). Veja na tabela abaixo como ficam os nomes das variáveis nas situações de atribuição e de expansão de valores:

`nome`	No momento da atribuição ou da declaração de uma variável, nós utilizamos apenas o seu nome.
`$nome`	Para que seu valor seja expandido (e todo acesso ao valor de uma variável será uma *expansão*), nós escrevemos um `$` antes do nome.

`${nome}`	A rigor, esta é a forma completa da expansão do valor de uma variável, mas podemos dispensar as chaves quando trabalhamos com variáveis escalares (nomes que apontam para um único valor).

> ***Lembre-se disso:*** *no Bash, o acesso ao valor de uma variável sempre será feito através de uma expansão!*

Observe um exemplo de criação e expansão de uma variável:

Exemplo 1.12 – Expandindo valores em variáveis.

```
:~$ fruta=banana
:~$ echo $fruta
banana
```

A primeira linha já deve ser fácil de entender – nós executamos uma operação de atribuição do valor `banana` à variável `fruta`. O comando `echo` também é um velho conhecido, assim como o fato do shell realizar expansões antes de executar os comandos. Sendo assim, o comando `echo`, em vez de enxergar a expressão `$fruta`, verá apenas o conteúdo da variável e, portanto, exibirá `banana` no terminal.

Mas talvez você esteja se perguntando: *"o que isso tudo tem a ver com as configurações do prompt?"* – e a resposta é: tudo!

Por exemplo, se quisermos ver como fica a nossa configuração do prompt, basta atribuí-la a uma variável e utilizar a expansão da variável como atributo do comando `echo`.

Mas veja o que realmente aconteceria observando o exemplo abaixo:

Exemplo 1.13a – Exibindo a configuração do prompt.

```
:~$ p='\u@\h:\w\$ '
:~$ echo $p
\u@\h:\w\$
```

Observe que os caracteres de controle não foram executados. Eu também poderia ter feito assim e nada de especial seria exibido:

Exemplo 1.13b – Exibindo a configuração do prompt.

```
:~$ p='\u@\h:\w\$ '
:~$ echo ${p}
\u@\h:\w\$
```

Mas, o que eu ainda não contei é que nós podemos interferir na forma como as expansões são feitas pelo shell (mais sobre isso no *capítulo 4*) introduzindo outros símbolos junto ao nome da variável entre as chaves. Então, quando queremos que os caracteres de controle da string em `PS1` sejam executados, nós podemos incluir os caracteres `@P` logo depois do nome da variável:

Exemplo 1.13c – Exibindo o prompt a partir da configuração.

```
:~$ p='\u@\h:\w\$ '
:~$ echo ${p@P}
blau@enterprise:~$
:~$
```

Perceba que o meu prompt original não foi alterado e eu pude visualizar o resultado da nova configuração sem alterar nada!

> ***Lembra?*** *Eu disse que "expansões do shell" são um dos assuntos mais importantes deste livro!*

1.3 – Executando no console ou no terminal

Chegou a hora de falarmos da principal ferramenta de trabalho de um programador GNU/Bash: o terminal. Mas, como sempre, a nossa abordagem aqui também não será linear – afinal, o melhor da viagem é o caminho, e não a chegada ao destino.

1.3.1 – Um pouco de história

Quando os computadores ocupavam andares inteiros, a interface entre eles e os usuários era feita com equipamentos bem menores, quase do tamanho de armários, que geralmente eram dotados de um dispositivo para a exibição de mensagens e outro para a entrada de comandos. Estes conjuntos de entrada e saída de dados acoplados a um computador eram chamados, por analogia, de *consoles*.

Nessa mesma época, também havia versões menores desses consoles que permitiam o compartilhamento dos recursos do computador entre várias estações de trabalho em locais diferentes – eram os chamados *terminais*. Com os avanços tecnológicos, porém, e à medida que os equipamentos foram reduzindo de tamanho, os consoles e terminais foram deixando de ser equipamentos físicos e passaram a ser virtualizados em software. Isso deu origem à expressões como: *console virtual* (VC) e *terminal virtual* (VT). Hoje, portanto, quando dizemos "*console*" ou "*terminal*", estamos nos referindo à virtualização em software dos antigos terminais e consoles eletrônicos dos primórdios da computação.

1.3.2 – Terminal ou console?

Por padrão, todo sistema operacional GNU/Linux fornece um console chamado TTY, que tem seu nome herdado dos antigos consoles de teletipo (*TeleTYpe*, as antigas máquinas com teclados acoplados a impressoras). Já nos

ambientes gráficos que utilizamos, existem os programas que emulam os terminais oferecendo uma interface semelhante à dos consoles. Eles recebem o nome de PTS (*PseudoTerminal Secondary*), mas são mais conhecidos como *emuladores de terminais*.

> ***Nota:*** *a rigor, a palavra "terminal" é aplicável tanto para consoles TTY quanto para emuladores PTS. Para facilitar a nossa comunicação, porém, vamos convencionar que, a partir de agora, quando eu disser "console", eu estou me referindo especificamente ao TTY, enquanto "terminal" será o nosso termo para seu equivalente gráfico.*

Embora terminais e consoles tenham essencialmente as mesmas capacidades, é possível que, por ser executado em um ambiente gráfico, um emulador de terminal tenha recursos à disposição que faltariam a um console (e vice-versa). Então, pode acontecer de termos que verificar se um script está sendo executado no terminal ou no console.

Com o utilitário `tty` é possível verificar se o shell está sendo executado em um console ou em um terminal. Executado em um terminal, o resultado seria parecido com isso:

Exemplo 1.14 – Executando o utilitário 'tty' no terminal.

```
:~$ tty
/dev/pts/0
```

Aqui nós temos algumas informações bem interessantes para discutir, a começar pelo fato do utilitário `tty` ter retornado um caminho (no caso, `/dev/pts/0`). Isso aconteceu porque, primeiro, é exatamente para isso que serve o utilitário `tty`: exibir o nome do arquivo do terminal conectado à saída padrão, e segundo, do ponto de vista dos sistemas operacionais *unix-like*, tudo, inclusive o hardware, possui uma representação na forma de um arquivo.

> ***A propósito:*** *a pasta* `/dev` *tem seu nome derivado justamente do termo em inglês "device", que significa "dispositivo", e é o local onde podemos encontrar os arquivos correspondentes aos dispositivos de hardware no sistema.*

A outra informação relevante – talvez a mais importante, tendo em conta a finalidade do programa – é o fato do dispositivo ter sido nomeado como `pts`, o que deixa bastante claro que o nosso terminal é um *pseudo-terminal secundário*.

Se quisermos verificar a saída do utilitário `tty` no console, nós teremos que sair do modo gráfico, o que é feito pressionando as teclas `CTRL`+`ALT`+`Fn`, onde `Fn` geralmente é uma tecla de função entre `F1` e `F6`. O atalho para retornar ao modo gráfico, porém, dependerá de como ele foi iniciado. Se você utiliza algum tipo de *gerenciador gráfico de display* (como o *LightDM* ou o *GDM*, por exemplo), é bem provável que o atalho seja `CTRL`+`ALT`+`F7`. Caso não seja, o mais provável é que você tenha que utilizar o atalho `CTRL`+`ALT`+`F1`.

> ***Não tenha pressa de entender como isso funciona.*** *O nosso experimento com o programa* `tty` *no console nos dará uma ideia melhor de como funciona essa numeração dos atalhos para sair do modo gráfico e voltar.*

Assim que estiver no console, você verá uma mensagem pedindo seu nome de usuário e, logo em seguida, a sua senha. Se você introduzir corretamente seus dados, um *shell de login* (mais sobre isso no *tópico 1.7*) será carregado. Já com o prompt visível, nós iremos executar o `tty`. No meu caso, como eu utilizei o atalho `Ctrl`+`Alt`+`F2` para entrar no console, o resultado foi esse:

Exemplo 1.15 – Executando o utilitário 'tty' no console.

```
:~$ tty
/dev/tty2
```

Diferente do resultado no terminal, agora o dispositivo se chama `tty2`, e isso indica que eu estou utilizando o segundo console TTY dentre os disponíveis no meu sistema.

> *Entendeu de onde vem a numeração dos atalhos* `Ctrl`+`Alt`+`Fn`*?*

Eu iniciei o console `tty2` porque, como eu não utilizo gerenciadores de display, eu ativo o meu ambiente gráfico diretamente no console `tty1`, que é o primeiro console que me é disponibilizado logo após o início do sistema. Ainda no meu caso, o console `tty1` continuará "preso" na execução do meu ambiente gráfico enquanto eu não terminar a sua execução.

Mas o programa `tty` ainda tem outra forma de ser utilizado. Com a opção `-s` (*slient*, "silenciosa"), ele não retornará nada visível:

Exemplo 1.16 – Comando 'tty -s'.

```
:~$ tty -s
:~$
```

Isso pode parecer meio inútil, mas é só na aparência. Na verdade, mesmo que não apresentem nada visível como saída, todos os comandos e programas informam o seu *estado de saída*, ou seja, se eles terminaram a sua execução com *sucesso* ou com *erro*.

Para capturar o estado de saída de um comando, o Bash nos oferece uma variável interna chamada `?`. Nós já sabemos acessar os valores das variáveis, então vamos ver o que acontece com o comando `tty -s`:

Exemplo 1.17 – Capturando a saída do comando 'tty -s'.

```
:~$ tty -s
:~$ echo $?
0
```

Aqui, o valor retornado foi zero (`0`). Para o shell, um estado de saída zero indica que o comando (ou o programa) terminou com *sucesso*, enquanto qualquer estado de saída com valor diferente de zero representa um *erro*.

> ***Isso é importante!*** *Guarde bem essa informação, porque ela será a base de todos os conceitos lógicos da programação em Bash.*

Então, voltando ao estado de saída do comando `tty -s`, ele será zero (sucesso) sempre que for executado diretamente em um terminal ou em um console. Por outro lado, se não houver um terminal envolvido na sua execução (como no caso de um script chamado por um atalho de teclado no ambiente gráfico, por exemplo), o estado de saída será *erro*.

1.4 – Os vários shells

Dizem que *"tudo é prego para quem só conhece martelo"*, mas nós temos muito mais do que apenas um martelo à disposição! O shell do GNU/Linux (e de outros sistemas *unix-like*) é uma caixa de ferramentas completa e ampliável de várias formas, o que sempre me leva ao trocadilho: *"para muito ou para pouco, o shell é o limite"*. Mais do que uma brincadeira com a infinidade de soluções possíveis em shell, isso é um lembrete de que, se existe um limite, ele é determinado apenas pelo shell que estamos utilizando.

Neste sentido, não podemos nos esquecer de que o Bash não é o único shell utilizado por aí. Eventualmente, nossos scripts em Bash poderão parar nas mãos de alguém que utiliza algum desses outros shells por padrão no seu sistema. A rigor, isso não é um problema – o Bash pode ser instalado em qualquer sistema *unix-like*, e a decisão de rodar nossos scripts vem sempre com a responsabilidade prover os requisitos necessários para isso.

Seja como for, o ponto é que existem algumas convenções estabelecidas para garantir um mínimo de portabilidade de scripts escritos em shell, e este é o

caso do conjunto de normas de compatibilidade estabelecidas no padrão POSIX. Alguns shells foram desenvolvidos de forma a adequarem-se aos padrões POSIX, outros mantém configurações compatíveis, mas oferecem recursos que não atendem totalmente às normas. Há também os shells que foram escritos priorizando outros critérios, como a facilidade de uso, o desempenho, ou até os recursos que oferecem para a criação de programas. Aqui está uma lista de alguns desses shells:

Shell	Executável	Descrição
Bourne Shell	`sh`	Desenvolvido por Stephen Bourne, da AT&T, é o Shell padrão do UNIX 7 em substituição do Thompson Shell, cujo executável possuía o mesmo nome, `sh`.
Bourne-Again Shell	`bash`	GNU/Bash ou, como é mais conhecido, Bash, é um shell de comandos Unix e uma linguagem interpretada escrita inicialmente por Brian Fox para o Projeto GNU em substituição ao Bourne Shell. Quando Fox foi afastado da FSF em 1994, o desenvolvimento do Bash passou para Chet Ramey.
Almquist Shell	`ash, sh`	Shell Unix escrito por Kenneth Almquist no fim dos anos '80 como um clone da variante System V.4 do Bourne Shell. Substituiu o Bourne Shell original nas versões do Unix BSD lançadas no começo dos anos '90, razão pela qual algumas implementações ainda utilizam o nome de executável `sh`.
Debian Almquist	`dash, sh`	Em 1997, o Almquist Shell foi portado

Shell		do NetBSD para o Debian que, em 2002, lançou uma versão renomeada para Debian Almquist Shell, ou `dash`, priorizando a compatibilidade com os padrões POSIX e uma implementação bem mais enxuta em relação à original.
Korn Shell	`ksh`	Desenvolvido sobre o código do Bourne Shell por David Korn no começo dos anos '80. Inicialmente, era um projeto proprietário e mais tarde adotou uma licença compatível com as definições da Open Source Initiative (OSI).
Z Shell	`zsh`	Criado com a proposta de ampliar as funcionalidades do Bourne Shell, o Zsh traz diversos recursos presentes no Bash e no KornShell.

Para nós, obviamente, o que interessa é o Bash, que é o shell padrão do Projeto GNU e o mais presente nas distribuições GNU/Linux. Ele vem sofrendo ampliações e melhorias há mais de duas décadas, o que faz dele um shell sólido, poderoso e comprovadamente capaz de atender os requisitos de um shell confiável em qualquer tipo de ambiente de produção ou de uso pessoal.

1.4.1 – Sobre os padrões POSIX

Cada shell tem a sua forma própria de receber e interpretar os comandos dos usuários. Sendo assim, se estivermos escrevendo um script em Bash que precisará ser compatível com plataformas que utilizem outros shells, é bem provável que tenhamos que observar as definições da norma POSIX.

POSIX vem de *Portable Operating System Interface* (ou *"Interface Portável Entre Sistemas Operacionais"*, em português). Então, como o próprio nome diz, o objetivo das normas POSIX é garantir a portabilidade do código. Porém, a não ser em casos raros e muito específicos, portabilidade é uma preocupação secundária.

A verdade é que é muito mais comum ocorrerem problemas decorrentes de incompatibilidades entre versões do próprio Bash, códigos mal escritos, diferenças no conjunto de programas instalados por padrão no sistema base das distribuições, do que por motivos de incompatibilidade entre plataformas.

Eu sei que muitos discordarão do meu ponto de vista, mas não posso deixar de pensar na criação de programas em Bash como a criação de programas em qualquer outra linguagem. Quando eu escrevo um programa em Python, eu só me preocupo com a versão disponível na plataforma de execução, não com a incompatibilidade do Python com a linguagem C, por exemplo. Até porque, em última análise, escrever programas em Bash seguindo as normas POSIX é o mesmo que escrever em outra linguagem – ou seja, um projeto que exija um código compatível com as normas POSIX será escrito no shell POSIX, não em Bash.

Além disso, todo shell é apenas um programa que pode ser baixado e instalado de forma relativamente simples em qualquer plataforma *unix-like*. Então, da mesma forma que um programa em Python precisa de um interpretador Python e tantas dúzias de bibliotecas, se o script foi feito em Bash, basta instalar o Bash e as demais dependências – afinal, a simples presença do Bash no sistema não causará nenhuma interferência no shell em uso.

1.5 – Descobrindo o shell em execução

Uma forma bastante simples de descobrir qual shell está sendo executado no seu terminal é verificando o valor armazenado na variável interna `0` (sim, este é o nome da variável).

A variável `0` faz parte de uma série de variáveis internas do shell chamadas *parâmetros posicionais* (mais sobre isso nos tópicos *2.3.8* e *3.9*). De acordo com o contexto, ela poderá conter o nome do script ou o nome do executável do shell. Executando a expansão do seu valor na linha de comandos, nós veremos que ela contém o nome do executável do shell:

Exemplo 1.18 – Descobrindo o nome do executável do shell.

```
:~$ echo $0
bash
```

Mas nós podemos ir além!

Existe um comando interno do shell muito interessante chamado `type`, cuja função é exibir informações sobre comandos. Com a opção `-a`, ele retorna todos os locais contendo um executável com o nome que for passado como argumento. Então, além de descobrirmos o nome do shell em execução, nós ainda podemos ver onde o seu executável está armazenado no sistema.

Observe o exemplo:

Exemplo 1.19 – Descobrindo a localização do executável do shell.

```
:~$ type -a $0
bash é /usr/bin/bash
bash é /bin/bash
```

Note que o conteúdo da variável `0` foi utilizado como argumento comando `type -a`. Após a expansão de seu valor, que no caso era `bash`, o comando foi

executado retornando, portanto, as duas localizações do executável `bash` no meu sistema.

> ***Detalhe importante!*** *A linha do comando do exemplo 1.19 só funciona se o shell em execução* ***não for*** *um shell de login, mas nós falaremos sobre isso no tópico 1.7, ainda neste capítulo.*

1.5.1 – A variável de ambiente SHELL

Diferente das variáveis internas `?` e `0`, as variáveis de ambiente não são necessariamente controladas pelo shell. De modo geral, nós podemos definilas e modificá-las conforme a necessidade, e este é o caso da variável de ambiente `SHELL`. Suas configurações iniciais são feitas na configuração do próprio sistema operacional. Para ser mais exato, ela é definida no conjunto de configurações feitas no desenvolvimento de uma distribuição GNU/Linux (ou de qualquer outro sistema *unix-like*) e pode ser alterada a partir de certos procedimentos administrativos.

Para a finalidade deste tópico, porém, basta saber que nela está o nome do shell configurado como padrão para o usuário. Para ver o que ela armazena, basta executar:

Exemplo 1.20 – Expandindo o shell configurado como padrão para o usuário.

```
:~$ echo $SHELL
/bin/bash
```

Eventualmente, dependendo do seu sistema operacional, pode ser que a variável `SHELL` nem esteja definida. Nestes casos, o shell padrão certamente será obtido a partir da leitura de um arquivo que contém informações de todos os usuários e grupos registrados no sistema: o arquivo `/etc/passwd`.

1.5.2 – Lendo o conteúdo de /etc/passwd

A forma mais comum de lermos o conteúdo de arquivos é com a utilização do comando `cat`. Sua função é concatenar o conteúdo de arquivos e exibir o resultado no terminal – claro, se apenas um nome de arquivo for informado, somente o seu conteúdo será exibido:

Exemplo 1.21 – Exibindo o conteúdo de /etc/passwd com o comando 'cat'.

```
:~$ cat /etc/passwd
```

Nós podemos utilizar o comando `cat` para ler o conteúdo do arquivo `/etc/passwd`, o problema é que ele pode ser muito grande, tornando difícil a localização de qualquer informação.

Nesses casos, a melhor opção pode ser o comando `grep`, que também exibe o conteúdo de arquivos, mas permite a realização de uma filtragem segundo um padrão de busca. Descartando toda uma infinidade de possibilidades de uso do comando `grep`, nós podemos ver a linha relativa ao nosso usuário desta forma:

Exemplo 1.22 – Filtrando o conteúdo de /etc/passwd com o comando 'grep'.

```
:~$ grep blau /etc/passwd
blau:x:1000:1000:Blau Araujo,,,:/home/blau:/bin/bash
```

Se você observar bem, a linha resultante contém vários dados separados entre si por dois pontos (`:`), e a última informação exibida é justamente o nosso shell padrão: `/bin/bash`.

Apenas para apresentar mais uma alternativa, nós também podemos ver qual shell está em execução com o nosso já conhecido comando `ps`:

Exemplo 1.23 – Executando 'ps' para descobrir o shell em execução.

```
:~$ ps
 PID TTY          TIME CMD
 4525 pts/0    00:00:00 bash
 5461 pts/0    00:00:00 ps
```

1.6 – Alterando o shell

Agora, se por acaso você utilizou um dos métodos do tópico anterior e acabou descobrindo que o seu shell padrão não é o Bash, não precisa se preocupar, a troca do shell é um processo relativamente simples. Obviamente, antes mesmo de pensar em trocar de shell, você precisa providenciar a instalação do Bash segundo os procedimentos descritos na documentação do seu sistema operacional (nisso eu não tenho como ajudar, desculpe). Certificando-se de que o Bash já está instalado, você pode decidir executá-lo temporariamente ou torná-lo o seu shell padrão.

A primeira opção, executá-lo temporariamente, não é nenhuma novidade – nós já fizemos isso quando invocamos um shell pela linha de comandos, lá no começo deste capítulo. Também já vimos (no mesmo tópico) que a linha do interpretador de comandos, que é a primeira linha de um script executável, também fará a invocação do shell necessário para interpretar os comandos do código. Então, falta apenas vermos como tornar o Bash o nosso shell padrão, o que faremos com o comando `chsh`.

1.6.1 – O comando 'chsh'

O comando `chsh` (*Change Shell*) é um utilitário desenvolvido especificamente para alterar o shell definido como padrão para um usuário. Ele pode ser executado como usuário normal, afetando apenas a configuração de sua própria conta, ou como usuário administrativo (*root*), quando for necessário alterar o shell de qualquer outro usuário.

> ***Atenção!*** *De um modo ou de outro, uma senha será solicitada: a sua, se estiver alterando apenas o seu shell, ou a senha do usuário root, caso esteja mudando o shell de outro usuário.*

Supondo que você queira alterar apenas o seu shell, basta executar a linha do comando abaixo como usuário normal:

```
chsh -s /caminho/do/shell nome_do_usuario
```

Por exemplo:

Exemplo 1.24 – Alterando o seu shell padrão.

```
:~$ chsh -s /bin/bash blau
:~$ Senha:
```

Simples assim!

Você ainda pode conferir a alteração listando as configurações do seu usuário no arquivo `/etc/passwd`.

> *Repare que o último campo da linha é o seu shell padrão.*

Exemplo 1.25 – Conferindo a alteração do seu shell padrão.

```
:~$ grep blau /etc/passwd
blau:x:1000:1000:Blau Araujo,,,:/home/blau:/bin/bash
```

1.7 – O shell de login

Para ser mais exato, o que nós alteramos com o comando `chsh` é o chamado *shell de login*. Mas, de modo algum, isso significa que há dois shells diferentes.

Essa designação diz respeito apenas a como uma sessão do shell é iniciada no terminal: com ou sem a solicitação das credenciais do usuário.

A confusão que se faz sobre isso, provavelmente deve-se ao fato de que existem, isto sim, duas *interfaces* nos sistemas operacionais modernos: a interface de linha de comandos (CLI) e a interface gráfica com o usuário (GUI). Contudo, nós geralmente não percebemos que temos que nos identificar para ganhar acesso a qualquer uma delas.

Quando iniciamos o sistema e caímos no chamado "modo texto", ou quando utilizamos os atalhos `Ctrl`+`Alt`+`F1` a `F7` para acessar um console, o sistema executará um gerenciador de login para solicitar as nossas credenciais. Em seguida, um shell será carregado e, neste caso, ele será um *shell de login*. Quando isso acontece, o conteúdo da variável `0` será precedido por um traço (`-`):

Exemplo 1.26 – Expandindo a variável '0' em um shell de login.

```
:~$ echo $0
-bash
```

> *O mesmo aconteceria, por exemplo, se estivéssemos fazendo o login em uma sessão remota do shell através de uma conexão SSH.*

Por detrás dos bastidores das interfaces gráficas, também acontece um processo semelhante – nós também temos que apresentar as nossas credenciais, mesmo quando o nosso *gerenciador de display* (LightDM, GDM, etc) está configurado para não solicitar uma senha. Estando na GUI, portanto, nós já estamos identificados, o que torna desnecessária (e pouco prática) uma nova solicitação de login e senha cada vez que iniciamos uma sessão do shell a partir de um terminal gráfico.

Para indicar que não estamos em um shell de login, a variável `0` deixa de exibir o traço antes do nome do shell:

Exemplo 1.27 – Expandindo a variável '0' em um shell que não é de login.

```
:~$ echo $0
bash
```

É por causa do traço incluído no início do nome do shell de login que o comando do exemplo 1.19 (`type -a $0`) só funcionará se o shell não for de login.

1.8 – Modos de execução

O mais importante, porém, é que fique claro que há dois modos de iniciar o shell, e isso terá uma influência direta na forma como ele será operado.

1.8.1 – Modo interativo

Quando abrimos um terminal e começamos a digitar comandos, nós estamos utilizando o shell de forma *interativa*. Nele, existe uma interação entre nós e o shell: nós digitamos um comando no prompt, o shell processa o comando, nos dá uma resposta, e nós pensamos e decidimos o que fazer em seguida.

1.8.2 – Modo não-interativo

Mas existe uma outra forma de trabalhar que pode ser muito útil e prática, principalmente quando precisamos automatizar a execução dos comandos: o modo *não-interativo*, que é, essencialmente, o que fazemos com os scripts: nós escrevemos todos os comandos em um arquivo de texto (código fonte) e mandamos o shell executar tudo que está lá a partir da invocação do nome desse arquivo.

As principais características dos modos interativo e não-interativo, são:

- No modo interativo, os comandos do usuário são passados para o shell de forma direta através da *entrada padrão* (*stdin*) e todas as mensagens retornadas por um comando ou um programa são direcionadas para a *saída padrão* (*stdout*), ambas (*stdin* e *stdout*) conectadas a um terminal.
- O conteúdo da variável `PS1` é lido a fim de que um prompt seja exibido no modo interativo.
- O modo não-interativo sempre é executado em sua própria sessão do shell, mesmo quando o script é invocado na linha de comandos.
- Entre os caracteres relativos às opções de execução do shell, obtidos com a expansão da variável especial `-` (traço), o caractere `i` estará presente apenas no modo interativo).

Geralmente não há muito com que se preocupar quando os nossos scripts executáveis em Bash são devidamente iniciados com a linha do interpretador de comandos (*shebang*) ou são explicitamente invocados como argumentos do executável `bash`. Eventualmente, porém, nós teremos que conferir uma ou outra configuração do Bash antes de contarmos com certos recursos nos nossos scripts, mas são casos bem específicos.

> *De todo modo, pelo menos no que for abordado aqui neste livro, nós faremos questão de deixar claro quando isso for necessário.*

1.8.3 – Como detectar se estamos ou não no modo interativo

Na prática, quase sempre isto seria bastante óbvio: scripts e comandos passados como parâmetros do executável do shell são executados no modo não-interativo, enquanto os comandos dados diretamente na linha de comandos são executados no modo interativo. Mas, até como pretexto para novas descobertas, é interessante notar que podemos identificar se estamos ou não no modo interativo através da expansão da variável especial `-` (traço).

Observe o exemplo:

Exemplo 1.28 – Expandindo as configurações do Bash.

```
:~$ echo $-
himBHs
```

Os caracteres expandidos aqui correspondem a diversas configurações de execução do Bash, como podemos ver na tabela abaixo:

h	*hash*	Registra a localização (caminho) dos comandos executados.
i	*interactive*	Modo interativo.
m	*monitor*	O controle de *jobs* está habilitado.
B	*Brace*	O shell executa expansões de chaves.
H	*History*	Habilita o acesso ao histórico com a exclamação (`!`).
s	*stdin*[5]	Indica que os comandos serão lidos a partir da entrada padrão.

Porém, no modo não-interativo...

Exemplo 1.29 – Expandindo as configurações do Bash não-interativo.

```
:~$ bash -c 'echo $-'
hBc
```

A primeira coisa que você deve notar, é que o `i` não está mais entre os caracteres expandidos, e este é o principal indicador de que estamos no modo não-interativo. Por último, observe que agora existe um novo caractere: o caractere `c`. Isso aconteceu porque o comando `echo $-` foi passado como argumento do comando `bash -c`.

5 Eu utilizei ***stdin*** apenas como um mnemônico, já que não há menção ao motivo desta opção ser chamada de **-s** no manual do Bash.

> *Se o comando* `echo $-` *tivesse sido executado em um script, nós veríamos apenas os caracteres* `hB`.

1.9 – Obtendo ajuda

Como eu sempre digo: *um bom programador não é aquele que tem todas as respostas na cabeça (até porque isso não existe), mas aquele que sabe onde e como encontrar rapidamente todas as respostas*. Sendo assim, a principal arma no arsenal de um programador sempre serão manuais e outras formas de ajuda que a sua linguagem tenha para oferecer.

> *Claro, os buscadores da internet e fóruns também ajudam muito, mas eu os considero como ferramentas que podem nos dar mais pistas do que efetivamente soluções – no fim das contas, as únicas informações realmente confiáveis são aquelas encontradas nos manuais.*

Neste aspecto, o Bash possui uma farta documentação que pode ser consultada de várias formas:

- Comando `man`;
- Comando interno `help`;
- Site do manual na web.

1.9.1 – Manual online

Pessoalmente, eu gosto muito do manual online, especialmente a versão em uma única página em HTML, mas você pode escolher a versão que achar melhor. As opções estão no link abaixo:

https://www.gnu.org/software/bash/manual/

1.9.2 – Comando 'man'

A vantagem do comando `man` é que, além do manual do próprio Bash, com ele nós podemos ler a documentação de outros comandos do shell que, eventualmente, venham parar nos nossos scripts. Para ler o manual do Bash, o comando é:

Exemplo 1.30 – Consultando o manual do Bash.

```
:~$ man bash
```

1.9.3 – Comando interno 'help'

Mas, se o que você procura é alguma informação sobre os comados internos do Bash, é possível que o comando `man` seja insuficiente. Para estes casos, o Bash oferece o comando interno (*builtin*) `help`...

Exemplo 1.31 – O comando interno 'help'.

```
:~$ help [opções] [comando_builtin]
```

Sem o parâmetro opcional `comando_builtin`, o comando `help` exibirá uma lista com todos os comandos para os quais ele oferece ajuda (todos *builtin*).

Por padrão, ele exibirá as informações sobre o comando pesquisado na forma de uma "pseudo-manpage" (que apenas imita o estilo de uma página exibida pelo comando `man`). Este mesmo comportamento pode ser conseguido com a opção `-m`. Experimente:

Exemplo 1.32 – Exibindo a ajuda completa.

```
:~$ help -m help
```

Se o que você procura é apenas uma breve descrição de um comando *builtin*, basta utilizar a opção `-d`:

Exemplo 1.33 – Exibindo apenas a descrição do comando interno.

```
:~$ help -d help
help - Display information about builtin commands.
```

Mas, se o que você quer ver é a sintaxe do comando, utilize a opção `-s`:

Exemplo 1.34 – Exibindo apenas a sintaxe do comando interno.

```
:~$ help -s help
help: help [-dms] [PADRÃO …]
```

Aliás, repare que a sintaxe retornada pelo comando acima fala de um `PADRÃO`, porque o `help` tentará encontrar os tópicos de ajuda que correspondam aos primeiros caracteres passados como parâmetro.

Por exemplo:

Exemplo 1.35 – Exibindo ajuda de comandos que começam com 'comp'.

```
:~$ help -d comp
compgen - Display possible completions [...]
complete - Specify how arguments are to be [...]
compopt - Modify or display completion [...]
```

1.9.4 – Descobrindo se o comando é ou não é *builtin*

Talvez não seja a melhor forma, mas o comando `help` também pode nos ajudar a descobrir se um comando é ou não é *builtin*. A ideia é simples: se ele retornar um *erro*, significa que o nome informado *não é* um comando interno do Bash.

Exemplo 1.36 – Testando se o comando 'ls' é builtin.

```
:~$ help ls
bash: help: nenhum tópico de ajuda corresponde a `ls'. [...]
```

> ***Um detalhe importante:*** *supondo que ainda não sabemos se o que estamos testando é um comando ou um programa utilitário durante os nossos experimentos, vamos chamá-los todos apenas de "comandos".*

Perceba que aquilo que chamamos de "comando `ls`" (utilizado para listar arquivos e pastas) é um *utilitário* do sistema operacional GNU, mas não é um *builtin* do Bash. Por isso, o comando `help` retornou um erro.

Mas, lembre-se: o shell sempre retorna um valor correspondente ao *estado de saída* do comando executado (reveja o *exemplo 1.17*) – este valor será zero (`0`) no caso de *sucesso*, ou qualquer outro inteiro diferente de zero no caso de *erro*. Às vezes, essa informação pode ser mais valiosa do que a mensagem que vier a ser exibida.

1.9.5 – Redirecionando mensagens e capturando estados de saída

Em geral, mensagens de erro são muito úteis no modo interativo, mas não costumam ajudar muito nos nossos scripts (modo *não-interativo*). Nós já vimos que o estado de saída do último comando executado fica armazenado na variável especial `?`, mas ainda falta entender um pouco melhor como as mensagens de erro vão parar no terminal.

Quando o shell é iniciado, ele recebe acesso a três fluxos de dados:

- Fluxo de entrada, ou *entrada padrão* (*stdin*);
- Fluxo de saída, ou *saída padrão* (*stdout*);
- Fluxo de erros, ou *saída padrão de erros* (*stderr*).

Por padrão, o fluxo de entrada espera a digitação de dados pelo teclado, o fluxo de saída é apresentado no terminal, e o fluxo da saída de erros está conectado com a saída padrão, fazendo com que as mensagens de erro também sejam exibidas no terminal.

Como tudo nos sistemas unix-like é tratado como arquivo, esses três fluxos não podem fugir à regra. Eles estão na pasta `/dev/fd` com os nomes `0`, `1` e `2`:

Exemplo 1.37 – Localizando os fluxos de dados '0', '1' e '2'.

```
:~$ ls /dev/fd
0    1    2...
```

Onde...

- `0` é a entrada padrão;
- `1` é a saída padrão;
- `2` é a saída padrão de erros.

> *Sendo ainda mais preciso,* `0`, `1` *e* `2`*, bem como todos os arquivos na pasta* `/dev/fd`*, são chamados de "descritores de arquivos" ("file descriptors", em inglês, dai o nome da pasta,* `fd`*).*

A coisa mais interessante, porém, é que esses fluxos de dados podem ser redirecionados, o que nos permite desviar dados, que normalmente iriam para a saída padrão, para arquivos, por exemplo, e isso é feito com os chamados *operadores de redirecionamento*.

O c*apítulo 5* será dedicado integralmente aos *redirecionamentos de fluxos de dados*, mas nós podemos começar a utilizar este incrível recurso desde já, especialmente os operadores de redirecionamento para arquivos `>` e `>>`.

O operador de redirecionamento `>` desvia o fluxo de dados da saída padrão para um arquivo qualquer. Se o arquivo existir, seu conteúdo será apagado e os dados da saída padrão serão escritos nele. Por outro lado, caso o arquivo não exista, ele será criado e a escrita será feita.

Vamos elaborar um exemplo bem simples e objetivo com o comando `echo`. Como vimos, ele exibe no terminal a *string* que ele receber como argumento.

Se ele *exibe no terminal*, isso quer dizer que ele envia os dados (a *string*, no caso) para a saída padrão! Logo, nós podemos utilizá-lo no nosso exemplo:

Exemplo 1.38a – Redirecionando a saída do comando 'echo' para um arquivo.

```
:~$ echo 'Olá, mundo!' > teste.txt
:~$
```

Repare que nada foi exibido desta vez, porque todos os dados que iriam para a saída padrão foram redirecionados para o arquivo `teste.txt`. Para conferir se foi isso mesmo que aconteceu, vamos utilizar o comando `cat`:

Exemplo 1.38b – Conferindo o conteúdo de 'teste.txt'.

```
:~$ cat teste.txt
Olá, mundo!
```

> *No próximo capítulo nós veremos como utilizar o operador de redirecionamento para arquivos para criar os arquivos dos nossos scripts.*

Por padrão, o operador de redirecionamento para arquivos captura os dados na saída padrão, cujo arquivo na pasta `/dev/fd` é `1`. Por este motivo, o número é omitido no comando. Mas nós podemos utilizar o mesmo operador para redirecionar o fluxo de dados da saída de erros (arquivo `2`), o que faremos utilizando como exemplo o erro que encontramos quando executamos o comando `help ls`:

Exemplo 1.39 – Redirecionando a saída de erros para o arquivo .

```
:~$ help ls 2> teste.txt
:~$ cat teste.txt
bash: help: nenhum tópico de ajuda corresponde a `ls'. [...]
```

Novamente, nada foi exibido na saída padrão – tudo foi redirecionado para o arquivo `teste.txt`, o que nós pudemos conferir, na sequência, com o comando `cat`.

Este recurso é muito útil quando queremos redirecionar mensagens de erro para um arquivo de *log*, mas não com este operador. Como vimos, o operador `>` apaga o conteúdo do arquivo antes de escrever qualquer coisa nele. Para manter as mensagens anteriores em um *log*, nós precisamos do operador de redirecionamento `>>`, que faz a inserção dos dados no final do arquivo – o que nós chamamos de *append*.

Então, aproveitando que o nosso arquivo `teste.txt` já tem uma saída do erro anterior registrada, nós podemos fazer um *append* de novos erros da forma abaixo:

Exemplo 1.40a – Criando um arquivo de log.

```
:~$ help mkdir 2>> teste.txt
```

O que podemos também conferir com o comando cat...

Exemplo 1.40b – Conferindo o arquivo de log.

```
:~$ cat teste.txt
bash: help: nenhum tópico de ajuda corresponde a `ls'...
bash: help: nenhum tópico de ajuda corresponde a `mkdir'...
```

1.9.6 – Enviando saídas para o limbo

Às vezes, porém, não nos interessa registrar mensagens de erro – nós só queremos nos livrar delas. Para isso, o GNU/Linux conta com um arquivo especial que eu costumo chamar brincando de "o limbo do sistema operacional", que é o arquivo `/dev/null`, cuja característica mais relevante para nós é o fato dele descartar qualquer coisa que seja escrita nele.

> *Em sistemas unix-like, ele é chamado de dispositivo nulo, e é disso que consiste a sua principal utilidade.*

Ainda utilizando o comando `help` (afinal, nós ainda queremos utilizá-lo para determinar se um comando é ou não é builtin), nós podemos descartar a mensagem redirecionando a saída padrão de erros para `/dev/null`:

Exemplo 1.41 – Desviando a saída de erros para /dev/null.

```
:~$ help -d ls 2> /dev/null
:~$
```

Neste caso, apenas as mensagens na saída padrão seriam exibidas:

Exemplo 1.42 – O que acontece quando não há erros.

```
:~$ help -d help 2> /dev/null
help - Display information about builtin commands.
```

Mas nós também podemos redirecionar ambas as saídas ao mesmo tempo para o dispositivo nulo. Para isso, nós utilizamos o operador de redirecionamento `&>`:

Exemplo 1.43 – Redirecionando stdout e stderr para /dev/null.

```
:~$ help -d help &> /dev/null
:~$ help -d ls &> /dev/null
:~$
```

Deste modo, nada é exibido e nós podemos verificar o estado de saída do comando para sabermos se ele terminou com sucesso ou com erro:

Exemplo 1.44 – Testando se um comando é ou não é builtin.

```
:~$ help -d help &> /dev/null
:~$ echo $?
0
```

```
:~$ help -d ls &> /dev/null
:~$ echo $?
1
```

Como já dissemos em notas anteriores, o estado de saída de comandos é a base de toda a lógica condicional da programação no Bash. O shell não estabelece conceitos ou formas de expressar valores booleanos para *falso* ou *verdadeiro*, a não ser por analogia.

Nós podemos considerar um estado de saída de erro como *falso*, mas o erro sempre será uma condição resultante da avaliação da execução de um comando. Aliás, este é o ponto que devemos ter sempre em mente quando programamos em shell: *tudo são comandos*.

2 – O Bash como linguagem de programação

Se, para alguns, a ideia de enxergar o Bash como uma linguagem de programação pode causar certa estranheza, para outros, isso é algo totalmente impensável! Porém, as argumentações não costumam girar em torno dos aspectos técnicos de se utilizar um shell para programar nem das aplicações possíveis de serem desenvolvidas, mas do simples fato de que a palavra "script", para eles, sugere algo diferente de um programa (em alguns casos, até algo inferior).

Sem dúvida, um script pode ser muitas coisas: desde um arquivo contendo uma série de comandos simples para a execução em lote no modo não-interativo, até os códigos complexos de grandes aplicações para o *desktop* ou para a *web*, como os scripts escritos em PHP, Javascript ou Python – e muito pouca gente questiona se esses scripts são programas ou não.

De todo modo, o ponto é que o termo "script" só é utilizado para diferenciar programas que são executados diretamente a partir da sua forma binária[6] daqueles que dependem de um intermediário para interpretar as instruções escritas no código.

> *No frigir dos ovos, isso não passa de uma forma coloquial de distinguir os programas que precisam ser compilados dos programas que são interpretados.*

Neste capítulo, nós tentaremos entender melhor o que é um programa, o que pode ser feito em Bash e o que ele pode nos oferecer como linguagem de

6 Em um arquivo **binário**, os dados estão registrados numa forma que permite que eles sejam transferidos diretamente para a memória. O termo vem do fato de que, em vez de caracteres, esses arquivos contêm apenas sequências binárias – cadeias de *zeros* e *uns*.

programação. Nosso objetivo principal e fazer uma apresentação geral dos conceitos, estruturas e elementos que podem ser encontrados na maioria das linguagens, observar como eles são tratados em Bash, e fornecer as bases necessárias para que você seja capaz de se desenvolver como um programador que compreende as nuances do *bashismo*[7].

2.1 – O que é um programa

A palavra inglesa *script* pode ser traduzida como "roteiro", e esta é uma das palavras da nossa língua que, concordemos ou não, atuam como sinônimos diretos da palavra *programa*. Ou seja, até por uma questão semântica, scripts não se parecem com programas nem são um tipo de programa – scripts são programas, e todo programa é um script!

Se pensarmos bem, programar é o ato de planejar como e quando algo será executado com o fim de atingir algum propósito. Esse "algo" pode ser várias coisas: uma viagem, um jantar com a pessoa amada, uma reforma na casa, um projeto do trabalho ou até, já no contexto da computação, alguma tarefa que queremos que o computador realize para nós. Computadores, aliás, são ferramentas interessantes, porque eles já foram criados com o propósito de receberem e executarem as instruções de um planejamento – tudo que temos que fazer é roteirizar, as instruções de uma forma que ele nos entenda.

Se nós estivermos trabalhando diretamente com o processador do computador, as sequências de instruções serão passadas no que chamamos de *linguagem de máquina*, que é basicamente uma forma de representar com números os sinais que cada pino do processador deverá receber. Geralmente, isso é feito escrevendo os números diretamente na memória do computador com a ajuda de alguma ferramenta eletrônica. Como você pode imaginar, é

7 ***Bashismo*** é um termo lúdico que se refere à escrita de scripts e comandos de acordo com as peculiaridades do Bash ou de um jeito que só tem como ser interpretado pelo Bash.

uma tarefa tediosa, demorada e que exige um alto grau de planejamento e organização.

2.1.1 – Linguagens compiladas e interpretadas

Da necessidade de simplificar a tarefa de programar computadores, surgiram os primeiros programas dedicados a traduzir instruções escritas em linguagens mais próximas do que nós, humanos, somos capazes de entender, para as sequências numéricas binárias de que o computador precisa para executar alguma coisa. Os programas que fazem esse tipo de tradução são os *compiladores* e *interpretadores*, e cada um deles está associado a uma forma específica de escrever os programas – as *linguagens de programação*.

Independente de como a tradução das instruções de um programa será passada para o processador, o fato é que ele sempre esperará receber sequências numéricas binárias vindas da memória. Então, a diferença mais fundamental entre uma linguagem compilada e uma linguagem interpretada diz respeito apenas a como esses números são disponibilizados na memória.

Uma linguagem *compilada* é aquela em que, através de um programa chamado *compilador*, o código em texto do programa (que também é chamado de *código fonte* ou apenas *fonte*), é transformado em sequências binárias que serão gravadas em outro arquivo. Quando o arquivo resultante é executado, todo seu conteúdo binário é carregado na memória a fim de ser processado.

Já nas linguagens *interpretadas*, o código fonte não passa por nenhuma transformação e nem é gravado em outro arquivo. Portanto, diferente do que acontece nos programas compilados, que já estão no formato binário, códigos em texto (também chamados de *scripts*) não podem ser carregados na memória. Quem vai para a memória, em vez disso, é o próprio interpretador da linguagem e, de lá, ele mesmo cuida da execução das instruções contidas no script.

É justamente por isso que o conteúdo dos scripts precisa ser passado, de alguma forma, como um parâmetro dos executáveis desse tipo de linguagem: seja através da linha da *shebang*, no próprio arquivo do script, ou da invocação do executável do interpretador, por exemplo:

Exemplo 2.1 – Executando programas interpretados na linha de comandos.

```
# Executando um script em PHP...
:~$ php script.php

# Executando um script em Python...
:~$ python script.py

# Executando um script em Bash...
:~$ bash script.sh
```

A principal diferença em relação às outras linguagens interpretadas é que, sendo o shell do sistema operacional, o Bash já está carregado e só precisa ser copiado para outra área da memória quando um script é executado. Os interpretadores, porém, terão obrigatoriamente que ser copiados de um dispositivo de armazenamento qualquer (um disco, por exemplo), carregados na memória, e só então poderão fazer o seu trabalho de interpretar códigos.

> *Obviamente, esta diferença pode não existir se o Bash não for o shell em uso no seu sistema operacional ou se o script não for executado diretamente em um terminal.*

2.1.2 – O Bash é um interpretador de comandos

Para ser mais exato, o Bash é um *interpretador de linhas de comandos* que podem conter comandos internos, palavras-chave, construtores de linguagem, estruturas de controle de fluxo, expressões, funções... e até chamadas a outros programas, mas tem algo que não muda: cada linha de um programa escrito em Bash é um comando do shell. Foi isso que quisemos dizer com a

nossa afirmação ao final do primeiro capítulo: *tudo são comandos*. Por mais que você acredite que essa informação seja irrelevante, ela faz toda diferença no modo de pensar, escrever e compreender um código em Bash.

Um exemplo bem característico dessa diferença é o fato da lógica dos nossos programas ser toda baseada no estado de saída de linhas de comandos em vez de estados lógicos (*verdadeiro* ou *falso*).

Observe o código abaixo:

Exemplo 2.2 – Uma estrutura de decisão 'if' em Bash.

```
if [[ $fruta == 'banana' ]]; then
    echo 'sucesso'
else
    echo 'erro'
fi
```

Essa é uma estrutura de decisão `if`, bastante comum no Bash, e pode parecer bem familiar para quem chega de outras linguagens, mas há diferenças sutis a que devemos estar muito atentos. Para começar, a estrutura `if` (na verdade, um *comando composto*) não reage à avaliação de expressões condicionais, mas ao estado de saída de qualquer comando.

O exemplo abaixo, portanto, é tão válido quanto o anterior:

Exemplo 2.3 – Testando estados de saída com 'if'.

```
if tty -s; then
    echo 'sucesso'
else
    echo 'erro'
fi
```

O que nos leva a deduzir, por observação, que a parte que está entre colchetes duplos (`[[ ]]`) no *exemplo 2.2* também é um comando! Em ambos os exemplos, o papel do `if` é acompanhar o estado de saída do comando

executado e dirigir o fluxo do programa para o *bloco de comandos* que corresponda ao caso: *then* → *sucesso* ou *else* → *erro*.

> *O comando composto* `[[ ]]` *avalia expressões condicionais, tal como o comando interno* `test`*, que veremos mais adiante ainda neste capítulo. Mesmo assim, ele não nos retorna valores como "verdadeiro" ou "falso" – ele simplesmente encerra sua execução com "sucesso" ou "erro", de acordo de com a avaliação da expressão em seu interior.*

Outro bom exemplo de como funciona a "lógica" do Bash está nos *operadores de controle*, especialmente os dois mais populares, `&&` e `||`, utilizados no *encadeamento condicional de comandos*.

2.1.3 – Operadores de controle

Geralmente, a primeira palavra da linha de um *comando simples* (para diferenciar dos *comandos compostos*) corresponde ao comando (ou programa) a ser executado. Em seguida, se houver, podem vir os argumentos e, por fim, sempre teremos um *operador de controle*, sem o qual, o comando não será executado.

No Bash, o operador de controle padrão é a quebra de linha (gerada quando teclamos `Enter` no terminal, por exemplo), mas ele não é o único. Por exemplo, tanto no terminal quanto nos scripts, é possível escrever vários comandos em uma mesma linha utilizando o ponto e vírgula (`;`) para separá-los – para a alegria dos programadores em C e PHP!

Sendo assim, a ocorrência de um operador de controle `;` indica para o shell que o comando deve ser executado, tal como se tivéssemos teclado um `Enter`, e que ainda pode haver outro comando encadeado na sequência...

```
Comando 1; Comando 2; …; Comando n
```

Neste caso, ou com quebras de linha em um script, todos os comandos subsequentes são executados *síncrona e incondicionalmente*, ou seja, a despeito do estado de saída do comando anterior, o comando seguinte aguardará seu término e será executado.

> *Aliás, quando se chega de uma vida de experiência na programação em linguagens como C e PHP, é muito comum sair terminando as linhas dos comandos com o `;`. Isso não causa nenhum erro e tem o mesmo efeito de duas quebras de linha seguidas, mas certamente denuncia as nossas origens. Então, não estranhe se esse hábito gerar "alfinetadas" por parte dos "bashistas". Em vez disso, leve na brincadeira e tente assimilar e praticar as particularidades do Bash.*

2.1.4 – Executando comandos em segundo plano

Um outro operador de controle que encadeia uma série de comandos incondicionalmente é o `&`, também chamado de *operador de comandos assíncronos*.

```
Comando 1 & Comando 2
```

Com ele, o segundo comando não espera o término do primeiro para ser executado. Neste caso, o primeiro comando é executado em outra sessão do shell (um *subshell*) e, para todos os efeitos, do ponto de vista do fluxo normal de execução, é como se ele tivesse sido encerrado com estado de saída de sucesso (`0`).

> *Como não há nada a ser avaliado quanto ao estado de saída de um comando que é executado assincronamente (ou, "em background", como se diz coloquialmente), o comando seguinte não precisa estar na mesma linha.*

2.1.5 – Encadeando comandos condicionalmente

De volta ao que falávamos sobre a lógica no Bash, os operadores de controle `&&` (*e*) e `||` (*ou*) são utilizados para encadear comandos de forma condicionada aos seus estados de saída. Por exemplo:

Exemplo 2.4a – Condicionando a execução de comandos numa lista.

```
:~$ true && echo 'sucesso' || echo 'erro'
```

O comando interno `true`, utilizado no exemplo, só tem semelhança com o valor booleano literal *true* no nome. Sua finalidade é não fazer nada além de ser executado e sair com *sucesso*. Seu irmão malvado, o comando interno `false`, por sua vez, sempre termina com estado de saída de *erro* quando executado. Portanto, temos três comandos nesta lista encadeada, mas apenas dois deles serão executados. Quem decide isso são os operadores condicionais `&&` e `||` a partir do estado de saída do comando testado.

Caso o comando `true` termine com sucesso (e ele sempre termina com sucesso), o comando que vier após o operador `&&` será executado e a mensagem `sucesso` será exibida. Caso contrário, só o comando que vier após o operador `||` será executado, causando a exibição da mensagem `erro`.

Observe:

Exemplo 2.4b – Condicionando a execução de comandos numa lista.

```
:~$ true && echo 'sucesso' || echo 'erro'
sucesso
:~$ false && echo 'sucesso' || echo 'erro'
erro
```

De certa forma, o funcionamento dos operadores `&&` e `||` pode ser comparado ao de uma estrutura condicional `if`, mas existem algumas diferenças importantes.

Por exemplo:

Exemplo 2.5 – Diferença entre o 'if' e os operadores '&&' e '||'.[8]

```
:~$ true && echo 'A'; false || echo 'B'
A
B
:~$ if true; then echo 'A'; false else echo 'B'; fi
A
```

Como podemos ver, a primeira linha fez com que tanto `A` quanto `B` fossem exibidos, ao passo que a segunda, com o comando composto `if`, causou a exibição apenas da string `A`. O problema aqui é que, no primeiro caso, estamos lidando com dois *operadores*, enquanto que, no segundo, nós temos uma estrutura que delimita *blocos de comandos*. De fato, a primeira linha equivaleria a isso:

```
:~$ true && echo 'A'
A
:~$ false || echo 'B'
B
```

Em termos mais tecnicamente precisos, os operadores `&&` e `||` são *binários*, ou seja, são operadores que trabalham com apenas dois termos: o comando testado e o comando que será executado caso a condição representada pelo operador seja atendida. Veja este outro exemplo:

Exemplo 2.6 – Os operadores de controle condicional são binários!

```
:~$ true && false || echo 'Olha eu aqui!'
Olha eu aqui!
```

Aqui, `true` termina com *sucesso* e executa o comando `false` que, por sua vez, sempre termina com *erro*, causando a execução do comando `echo`. Como os operadores são binários, a linha do exemplo equivale às duas linhas abaixo:

8 Meu muito obrigado ao amigo e conspirador André Amaral por este exemplo.

```
:~$ true && false
:~$ false || echo 'Olha eu aqui!'
Olha eu aqui!
```

Para nunca mais se esquecer dessa ideia, a minha sugestão é que você pense das seguintes formas:

- **Comando composto "if":** se o comando testado terminar com sucesso, execute o "*bloco then*"; caso contrário, execute o "*bloco else*".
- **Operador de controle "&&":** se o comando testado terminar com sucesso, execute "*o comando seguinte*" – pronto!
- **Operador de controle "||":** se o comando testado terminar com erro, execute "*o comando seguinte*" – pronto!

Outro detalhe importante é que, sendo binários, os operadores `&&` e `||` exigem que, pelo menos, o termo da esquerda (o comando a ser testado) esteja presente na linha de comando, ou teremos um erro de sintaxe no interpretador de comandos. O exemplo abaixo provocaria esse tipo de erro:

Exemplo 2.7a – '&&' e '||' exigem o termo da esquerda!

```
# Isso provoca erro!
True
&&
echo 'ok'
||
echo 'não ok'
```

Já isso funcionaria...

Exemplo 2.7b – Comandos após '&&' e '||' podem estar em outra linha!

```
true &&
echo 'ok' ||
echo 'não ok'
```

```
# Saída: 'ok'

false &&
echo 'não ok' ||
echo 'ok'

# Saída: 'ok'
```

Quando o segundo termo dos operadores `&&` e `||` não está na mesma linha, a próxima linha que contiver um comando será executada (ou não) de acordo com a condição especificada e o estado de saída do comando testado. Sendo assim, `true &&` e `false ||` habilitam a execução do comando seguinte, enquanto `true ||` e `false &&` impedem. Para entender melhor a "lógica" dos operadores de controle condicional.

Observe como as linhas do *exemplo 2.7b* foram processadas:

```
true && --------> (sucesso) pode executar.
echo 'ok' || ---> (sucesso) não pode executar.
echo 'não ok'

# Saída: 'ok'

false && -----------> (erro) não pode executar.
echo 'não ok' || ---> (sucesso) não poderia executar,
echo 'ok'             mas esta linha também não pôde
                      ser executada!
# Saída: 'ok'
```

> *Então, tenha muito cuidado com o que pode parecer lógico para você, nós estamos lidando com operadores que reagem aos estados de saída dos comandos testados, o que pode causar confusões.*

O exemplo abaixo ilustra como é fácil cometer erros quando ignoramos a lógica dos operadores de controle condicional:

Exemplo 2.8 – Cuidado com a lógica dos operadores '&&' e '||'!

```
false || -------> (erro) pode executar!
echo 'ok' && ---> (sucesso) pode executar!
echo 'não ok'

# Saída: 'ok' e 'não ok'!
```

2.1.6 – Classificações do Bash como linguagem de programação

Para aplacar a inquietação daqueles que precisam categorizar tudo para poderem dormir mais tranquilos (como eu), vejamos como o Bash pode ser classificado no espectro dos principais conceitos que caracterizam uma linguagem de programação.

Brincadeiras à parte, todas essas classificações refletem os principais estilos e técnicas que podemos aprender e aplicar na programação em cada linguagem, e o Bash, embora altamente flexível neste aspecto, também possui seus estilos e técnicas.

De forma resumida, podemos dizer que o Bash é uma linguagem...

- **De propósito geral:** uma linguagem que pode ser utilizada na criação de uma vasta gama de aplicações.
- **Interpretada:** é o interpretador de comandos que executa as instruções escritas no código.
- **De alto nível:** oferece uma sintaxe que se aproxima mais das linguagens humanas do que da linguagem da máquina.
- **Procedural (ou imperativa):** o código especifica os passos que devem ser executados para a realização do objetivo desejado.
- **Estruturada:** o código é escrito com ênfase em sequências de instruções (comandos), decisões e iterações (laços de repetição, ou *loops*).

- **De tipagem indeterminada:** o Bash só trabalha com dados do tipo *string* e *inteiro* (números sem casas decimais), mas não faz distinção entre eles até que sejam utilizados em algum contexto, o que, de certa forma, também pode enquadrá-lo na categoria das linguagens de *tipagem dinâmica*.

> *Apesar de trabalhar apenas com strings e inteiros, outros tipos de dados podem ser utilizados através dos comandos e programas utilitários que entrarem nos nossos códigos.*

Sendo uma linguagem de propósito geral, o Bash permite a criação de programas para as mais diversas finalidades, desde pequenos scripts de automação para tarefas diárias até a criação de programas bem mais complexos.

O que não deve ser esquecido, porém, é o fato de que, quanto mais complexo, mais os programas escritos em Bash dependerão de outros programas especializados para as tarefas não cobertas nativamente pelos recursos do shell. Em princípio, isso não representa problema algum – afinal, toda a Filosofia Unix se baseia na criação de sistemas complexos a partir de pequenas ferramentas especializadas capazes de trocar dados entre si.

> *Nestes casos, o Bash funcionaria como uma "cola", unindo e atribuindo um fluxo para o tráfego e o processamento dos dados.*

De forma alguma isso quer dizer que programas mais complexos não possam ser escritos em Bash puro, mas sempre vale a pena pesquisar e avaliar se as soluções externas não poderiam oferecer soluções mais vantajosas, especialmente se essas soluções vierem de ferramentas que estão disponíveis entre os programas utilitários centrais do sistema operacional GNU – os chamados *core utils,* como *sed*, *grep*, *awk* e outros programas.

2.2 – Nosso primeiro programa em Bash

A aplicação mais elementar de um programa em Bash é a execução de comandos em lote. Considere, por exemplo, a situação hipotética de um usuário que precise, por algum motivo, executar as quatro linhas de comando abaixo em sequência e com uma certa frequência:

```
:~$ whoami
blau
:~$ hostname
enterprise
:~$ uptime -p
up 2 weeks, 3 days, 19 hours, 42 minutes
:~$ uname -rms
Linux 4.19.0-6-amd64 x86_64
```

Esses utilitários exibem as seguintes informações:

whoami	Exibe o nome do usuário.
hostname	Exibe o nome da máquina na rede.
uptime	Informa há quanto tempo a máquina está ligada.
uname	Exibe informações sobre o kernel.

Sabendo que o shell permite a execução de comandos a partir de um arquivo, faz muito mais sentido criar um script contendo todos esses comandos e executar o lote todo de uma vez invocando apenas o nome desse script. O conteúdo do arquivo teria a seguinte aparência:

Exemplo 2.9 – Como seria um script de comandos em lote.

```
whoami
hostname
uptime -p
uname -rms
```

Se esse arquivo fosse salvo com o nome `infos.sh`, por exemplo, o script poderia ser executado assim:

Exemplo 2.10 – Executando o script 'infos.sh'.

```
:~$ bash infos.sh
blau
enterprise
up 2 weeks, 3 days, 19 hours, 42 minutes
Linux 4.19.0-6-amd64 x86_64
```

Observe que o script `infos.sh` foi executado como argumento de uma invocação do executável do Bash. Isso foi necessário por dois motivos: o nosso arquivo não tem permissão de execução e nem contém a linha do interpretador de comandos (a *shebang*).

> *Na verdade, a linha do interpretador seria totalmente desnecessária neste exemplo – afinal, o script não contém nada que precise ser interpretado especificamente pelo Bash.*

Para que seja possível chamar o nosso script pelo nome, vamos alterar sua permissão de execução:

Exemplo 2.11 – Tornando 'infos.sh' executável.

```
:~$ chmod +x infos.sh
```

Mas ainda não será possível chamá-lo *apenas* pelo nome. Acontece que todo programa precisa ser localizado pelo shell antes de ser executado, ou seja, é preciso informar um *caminho* até ele. Aqui no exemplo, nós estamos supondo que o script foi salvo no diretório corrente, mas isso não nos isenta da necessidade de informar um caminho, tanto que só será possível executar o script desta forma:

Exemplo 2.12 – Executando um script no diretório corrente.

```
:~$ ./infos.sh
```

Onde o `./` representa justamente o diretório em que nós estamos trabalhando no momento.

2.2.1 – A variável PATH é o caminho

Para que a informação do caminho seja realmente dispensável, o nosso script precisa estar salvo em um dos diretórios listados na variável de ambiente `PATH`. Se você não sabe quais são esses diretórios, basta executar no terminal...

Exemplo 2.13 – Exibindo os caminhos na variável 'PATH'.

```
:~$ echo $PATH
```

Aqui, o comando `echo` irá exibir todas as pastas que estão listadas na variável de ambiente `PATH`, cada uma delas separada por dois pontos (`:`) das outras. Por exemplo, está é a listagem aqui no meu sistema:

```
:~$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games:/home/bl
au/bin
```

Esta lista mostra que os aplicativos e scripts que eu executar no prompt de comando serão procurados nestes caminhos na seguinte ordem:

```
/usr/local/bin
/usr/bin
/bin
/usr/local/games
/usr/games
/home/blau/bin
```

Repare que, no meu caso, a última pasta em que o shell fará a busca pelos arquivos dos executáveis será em `/home/blau/bin`, que é um diretório (`bin`) na minha pasta de usuário (`/home/blau`). Este caminho não existia originalmente na variável de ambiente `PATH` quando eu instalei o meu sistema, eu tive que incluí-lo com o seguinte comando:

Exemplo 2.14 – Incluindo temporariamente um caminho na variável 'PATH'.

```
:~$ PATH=$PATH:/home/blau/bin
```

Como você pode ver, o comando é uma operação de atribuição, como aquela vista no *exemplo 1.10*, mas algo parece diferente, por isso vamos tentar entendê-lo da mesma forma que o shell faria.

A primeira coisa que o shell faz antes de interpretar uma linha de comando é procurar pelas possíveis expansões, o que ele descobriria pela presença do sinal `$`. Como vimos, o `$` é utilizado na expansão dos valores armazenados nas variáveis (também chamada de *expansão de parâmetros*), logo, o shell continuaria percorrendo a linha em busca de um nome de variável válido, encontrando os caracteres `P`, `A`, `T`, `H`...

Opa! O dois pontos (`:`)não é um caractere válido no nome de variáveis!

Logo, o shell sabe que o nome da variável é apenas `PATH` e tentará providenciar a sua expansão, ou seja, a string `$PATH` será substituída na linha do comando pela string correspondente a toda a lista de caminhos já armazenados na variável `PATH`.

Como não existe mais nada a ser expandido, todo o restante da linha será deixado exatamente como está, resultando em uma grande string correspondendo ao valor anterior em `PATH` acrescida da string relativa ao caminho que nós estamos incluindo na lista, e esta string resultante será o novo valor da variável `PATH`.

> ***Parabéns!*** *Você acabou de aprender como se faz uma "concatenação de strings" em Bash!*

Agora, todos os executáveis que forem salvos na minha pasta `~/bin` serão encontrados pelo shell quando eu quiser chamá-los apenas pelo nome. Então, vamos em frente!

Exemplo 2.15 – Movendo 'infos.sh' para um diretório listado em 'PATH'.

```
:~$ mv infos.sh ~/bin
:~$ infos.sh
blau
enterprise
up 2 weeks, 3 days, 19 hours, 42 minutes
Linux 4.19.0-6-amd64 x86_64
```

Só tem um problema: a inclusão da pasta `~/bin`, da forma que foi feita, é temporária e será perdida assim que eu iniciar outra sessão do shell. É por isso que o comando que nós demos no terminal deve ser escrito no arquivo `~/.bashrc`:

```
PATH=$PATH:/home/blau/bin
```

Geralmente, eu incluo este comando na última linha do meu arquivo `~/.bashrc`. Para que a mudança tenha efeito imediato, como já vimos no tópico sobre o prompt de comandos (*exemplo 1.11*), nós precisamos executar o comando `source`:

Exemplo 2.16 – Aplicando as mudanças do 'PATH' na sessão atual do shell.

```
:~$ source .bashrc
```

2.2.2 – Portabilidade e a linha do interpretador

Como vimos, no script `infos.sh`, a linha do interpretador é opcional porque não há nenhum comando no código que exija a interpretação de um shell específico. Contudo, se existisse (ou até por uma padronização das nossas práticas), ela seria obrigatória e poderia ser escrita de duas formas:

```
#!/bin/bash
```

Ou...

```
#!/usr/bin/env bash
```

Ambas são válidas, mas a segunda geralmente é considerada a mais portável. Pessoalmente, eu tenho minhas ressalvas, mas é melhor tentarmos entender como as duas funcionam antes de entrarmos nessa discussão.

Em comum, existe o fato de que ambas invocarão o executável do Bash, a diferença está apenas na forma como essa invocação é feita. Enquanto `/bin/bash` expressa o caminho completo até o executável `bash`, `/usr/bin/env` é a chamada (também com o caminho completo) para o programa `env`, este sim, responsável por encontrar e executar o programa que ele receber como parâmetro – no caso, `bash`.

O utilitário `env` (de *environment*, "ambiente"), sem argumentos, pode ser utilizado para listar as configurações de ambiente atuais do shell, mas a sua principal função é permitir a alteração dessas configurações antes de executar um programa. Daí a definição que encontramos no manual (`man env`):

> *"Executa um programa em um ambiente modificado".*

Contudo, a modificação é opcional e, se não houver nenhum outro argumento além do nome do programa, este será localizado através da variável `PATH` e executado.

A ideia da maior portabilidade vem daí: não importa em que diretório o executável do Bash foi instalado pelo sistema operacional, sempre será possível encontrá-lo a partir das configurações de ambiente.

Por outro lado, daí minhas ressalvas, contar com a variável `PATH` para encontrar o Bash (ou qualquer outro shell) pode ser um risco de segurança que, embora remoto, algumas instituições com operações mais críticas não estão dispostas a correr em suas instalações.

O risco vem da possibilidade de algum programa malicioso (*malware*) ser instalado em qualquer um dos diretórios listados na variável `PATH` que apareça antes da localização real do interpretador que queremos executar (lembre-se: o shell fara a busca do executável seguindo a ordem em que os diretórios aparecem na lista). Assim, em vez de executar o Bash, o programa malicioso é que seria executado. Isso pode ser especialmente mais arriscado se os caminhos definidos pelo usuário forem colocados no início da lista de caminhos executáveis.

```
PATH=/home/usuário/pasta:$PATH
```

Em vez do que fizemos:

```
PATH=$PATH:/home/usuário/pasta
```

Mas, veja, a minha ressalva não tem uma relação direta com aspectos de segurança (também acho isso meio exagerado), mas com o fato de que instituições com esse tipo de preocupação acabam restringindo as definições de ambiente que envolvam caminhos ou exigindo que os scripts que utilizam contenham o caminho completo daquilo que se pretende executar como interpretador – e assim, lá se vai o argumento da portabilidade.

De qualquer forma, isso é muito raro, e a opção com o utilitário `env` ainda é a que tem mais chances de ser portável. Portanto, daqui para frente, ao menos nos exemplos deste livro, o nosso padrão será:

```
#!/usr/bin/env bash
```

2.2.3 – O shell não sabe o que são extensões de arquivos

Um detalhe muito interessante diz respeito aos nomes que damos aos arquivos dos nossos scripts: para o shell, as extensões de arquivos não têm nenhum significado especial. Na verdade, o costume de nomear scripts em shell com a terminação `.sh` só tem utilidade para a *nossa* organização e para deixar claro *para nós* que o arquivo é um script em shell.

2.2.4 – Organizando o fluxo de trabalho com links simbólicos

Por falar em organização, nomes e caminhos, para que os nossos scripts sejam executados a partir de qualquer local, seus arquivos não precisam estar necessariamente nos diretórios listados na variável `PATH` – mas algo precisa estar lá.

Falando um pouco do meu fluxo de trabalho, é muito raro eu copiar ou mover os arquivos dos meus scripts para a minha pasta `~/bin` ou qualquer outro caminho da variável `PATH`. Em vez disso, eu prefiro criar e fazer a manutenção dos meus scripts nas pastas de seus respectivos projetos. Para que eu possa executá-los de qualquer local, eu acho muito mais prático criar os chamados *links simbólicos* (ou "ligações simbólicas") apontando para os locais onde os scripts realmente estão.

Por exemplo, digamos que eu esteja trabalhando no projeto "infos" e o caminho do script seja `~/projetos/infos/infos.sh`. Em tese, eu precisaria deixar os meus executáveis na pasta `~/bin`, mas, em vez de mover ou copiar o

arquivo `infos.sh` para lá, o que poderia dificultar bastante a manutenção do código, eu criaria uma *ligação simbólica* apontando para ele com a ajuda do comando `ln -s`:

Exemplo 2.17 – Criando uma ligação simbólica.

```
:~$ ln -s ~/projetos/infos/infos.sh ~/bin/infos
```

Deste modo, um arquivo seria criado na pasta `~/bin` com o nome `infos`, e ele faria o script `~/projeto/infos/infos.sh` ser executado sempre que seu nome fosse chamado na linha de comandos:

Exemplo 2.18 – Executando o script 'infos.sh' a partir do link simbólico 'infos'.

```
:~$ infos
blau
enterprise
up 2 weeks, 3 days, 19 hours, 42 minutes
Linux 4.19.0-6-amd64 x86_64
```

Não é nosso objetivo entrar em detalhes sobre os programas utilitários apresentados neste livro, mas é importante que você saiba que existem dois tipos de ligações simbólicas em sistemas *unix-like*: as *fortes* e as *fracas* (que são, efetivamente, as *ligações simbólicas* ou *symlinks*).

Uma *ligação forte* – que é criada quando executamos o utilitário `ln` sem a opção `-s` – equivale a darmos um segundo nome para o mesmo arquivo (sim, um arquivo pode ter vários nomes), ao passo que uma *ligação fraca* é um pequeno arquivo especial que contém apenas um caminho para o verdadeiro arquivo.

Quando a ligação simbólica é acessada para leitura ou escrita, o *kernel* "troca" seu nome por uma referência ao caminho e ao nome do arquivo para o qual ela aponta – exceto no caso de uma remoção, situação em que só a ligação simbólica é deletada e o arquivo original é preservado.

> *O fato da ligação simbólica estar numa pasta listada na variável* `PATH` *não isenta o arquivo original de ter que receber permissão para ser executado!*

Uma experiência interessante seria conferir se o *symlink* realmente foi criado corretamente. Nós podemos fazer isso utilizando o programa `ls` (listar arquivos) com a opção `-l` (exibição detalhada em coluna):

```
:~$ ls -l ~/bin/infos
```

Se tudo tiver corrido bem, nós veremos, entre outras informações, algo bem parecido com isso:

```
/home/blau/bin/infos -> /home/blau/projetos/infos/infos.sh
```

Mas existe outra forma de fazermos essa verificação utilizando apenas os recursos nativos do shell.

2.2.5 – Quem disse que o Bash não trabalha com valores lógicos?

Antes de entrarmos neste assunto, porém, me parece que este é o momento perfeito para um *plot twist* muito importante!

> *Eu disse que a lógica das estruturas de decisão no Bash se baseia em estados de saída de comandos, disse que* `true` *e* `false` *são comandos, mas nunca disse que o Bash seria incapaz de expressar valores lógicos ou de realizar operações lógicas!*

Como a maioria das linguagens, o Bash também permite a construção de operações que expressam valores lógicos. Para nós, não existem os valores booleanos literais *true* e *false*, mas nós podemos trabalhar com seus equivalentes inteiros. Isso pode ser demonstrado facilmente com uma *expansão de expressões aritméticas* – `$(( ))`.

Observe com atenção:

Exemplo 2.19 – Expandindo avaliações lógicas.

```
:~$ a=10; b=15
:~$ echo $(($a == $b))
0
:~$ echo $(($a != $b))
1
```

Nós conversaremos melhor sobre isso mais adiante, mas, por ora, basta saber que o comando composto `(( ))` (equivalente do comando interno `let`) é utilizado para avaliar operações aritméticas e lógicas. Quando ele recebe um cifrão na frente, o shell faz uma expansão do valor resultante dessas operações. Deste modo, o comando `echo` do exemplo está "vendo" apenas os valores resultantes expandidos.

Repare também que nós estamos realizando duas operações diferentes: uma operação de *igualdade,* com o operador `==`, e uma operação de *não-igualdade*, utilizando o operador `!=`. Operações deste tipo (as *comparações*) sempre expressam aquilo que o estudo da lógica chama de *proposição* – uma declaração que, quando submetida a uma análise, poderá ou não corresponder à verdade. Na programação, essa análise é sempre seguida de uma *avaliação*, e é só assim que uma expressão passa a representar um valor.

> ***Lembre-se:*** *"avaliar" significa "atribuir valor".*

2.2.6 – Toda falsidade é um erro

Sendo assim, quando eu proponho que o valor em `a` é igual (`==`) ao valor em `b`, essa expressão será avaliada e, se a sua declaração for *verdadeira*, ela receberá um valor que, no Bash, será o número inteiro `1` – caso seja *falsa*, o valor será `0`.

De fato, outras linguagens utilizam os valores `1` e `0` para representar os valores *true* e *false*, respectivamente, mas, quanto aos *estados de saída* das avaliações de expressões lógicas, ou seja, dos comandos em si, uma avaliação que resulte em `0` (falso) causará um estado de saída de erro (`1`) e qualquer avaliação diferente de `0` resultará num estado de saída de sucesso (`0`).

Aqui está um exemplo que confirma o que estou afirmando:

Exemplo 2.20 – Testando avaliações lógicas.

```
:~$ a=10; b=15
:~$ ((x=$a == $b)); echo $x $? # Falso / Erro
0 1
:~$ ((x=$a != $b)); echo $x $? # Verdadeiro / Sucesso
1 0
```

Nele, a expressão `($a == $b)` (falsa) foi avaliada com o valor `0`, enquanto que a expressão `($a != $b)` (verdadeira) recebeu o valor `1`.

> ***Repare bem:*** *as avaliações lógicas só podem ser feitas sobre expressões, mas nem toda expressão está sujeita a avaliações lógicas!*

2.2.7 – Programar é uma forma de expressão

Uma *avaliação lógica* é aquela que atribui à expressão os valores que a linguagem utiliza para representar a verdade ou a falsidade de uma proposição. Quem determina o tipo de avaliação que uma expressão receberá é o *operador* em uso. Deste modo, ainda no *exemplo 2.20*, os operadores de comparação `==` e `!=` farão com que as suas respectivas expressões sejam avaliadas logicamente.

Não podemos nos esquecer de que há um outro operador nos exemplos: o operador de atribuição utilizado na primeira linha de comando: `a=10` e `b=15`.

Pode não parecer, mas até uma operação de atribuição é uma expressão e, portanto, possui um valor. Vamos conferir:

Exemplo 2.21 – Atribuições também são expressões e recebem valores.

```
:~$ echo $(( (a=10) + (b=15) ))
25
```

Mas existe uma diferença entre este último exemplo e as atribuições que aparecem nos exemplos anteriores: nós estamos pedindo ao shell que avalie essas expressões. No Bash, sem um comando para avaliar as expressões, elas não recebem valores, por exemplo:

Exemplo 2.22 – Expressões precisam ser avaliadas por algum comando.

```
:~$ a=b=10
:~$ echo $a
b=10
:~$ a=$((b=10))
:~$ echo $a
10
```

2.2.8 – Novamente: cuidado com o que você acha que é lógico!

No começo deste capítulo nós dissemos que o comando composto `[[ ]]` avaliava *expressões condicionais*. Para mim, essa definição (que vem do manual) tem sérios problemas, mas vamos deixar isso de lado por enquanto. No momento, é muito importante que você entenda que a avaliação de expressões pode gerar situações diferentes dependendo do comando que nós utilizamos. Observe:

Exemplo 2.23 – Testando a comparação de duas strings.

```
:~$ a=banana; b=laranja
:~$ (( a == b )); echo $?
0
```

```
:~$ [[ $a == $b ]]; echo $?
1
```

> ***Para refrescar a memória:*** *a variável especial* `?` *armazena o estado de saída do último comando.*

Tirando o fato de que o comando composto `(( ))` dispensa o uso do `$` para acessar o valor das variáveis (ele não depende de expansões para obter os valores), por que diabos o shell registrou dois estados de saída diferentes para a mesma operação de comparação?!

Por observação, é fácil dizer que a string na variável `a` é diferente da string na variável `b`. Portanto, está claro que a expressão está sendo avaliada como *falsa*. A diferença está em como os dois comandos tratam a avaliação.

No segundo comando, com `[[ ]]`, o valor resultante da avaliação, (`1`, *falso*) é passado diretamente para o shell e é registrado na variável `?` como um estado de saída (um *erro*, no caso). Já no primeiro comando, a avaliação também atribuiu o valor `1` (*falso*) à comparação – e nós poderíamos tê-lo capturado:

```
:~$ a=banana; b=laranja
:~$ (( c = (a == b) ))
:~$ echo $c
1
```

Acontece que o comando `(( ))` tem uma peculiaridade: ele sempre sai com *erro* (estado de saída `1`) se o valor resultante da avaliação das expressões for `0`, e sempre sai com *sucesso* (estado de saída `0`) se o valor resultante for qualquer coisa diferente de `0` – e o valor resultante no exemplo foi `1` (*falso*, diferente de `0`).

Só para você fixar:

```
:~$ ((0)); echo $?
1
:~$ ((1)); echo $?
0
```

> *Sim, inteiros literais também são expressões e recebem o mesmo valor que eles representam, enquanto as strings sempre receberão o valor* `0`.

2.2.9 – Testando "expressões condicionais"?!

Agora que entendemos o que são *expressões* e o que significa *avaliar*, nós podemos falar um pouco da minha "bronca" com a descrição do comando `[[ ]]` nos manuais do Bash. Na verdade, meu problema tem a ver com algo bastante elementar: não existe nenhuma expressão que possa ser qualificada como *"condicional"*.

Ser "condicional" é um atributo de estruturas responsáveis por tomar decisões, estas sim, condicionadas, entre outras coisas, à avaliação de expressões a que se possam atribuir os valores de *verdadeiro* ou *falso*. No fim das contas é somente isso que se quer do comando `[[ ]]`: avaliar se a proposição de uma expressão é verdadeira ou falsa.

Custava escrever isso no manual?

> *Avalia se a proposição de uma expressão é verdadeira ou falsa.*

Mas a coisa piora muito mais... Veja o que aparece na descrição exibida pelo comando `help -d`:

```
:~$ help -d [[
[[ ... ]] - Executa um comando condicional.
```

Como assim, "executa"?!

O comando `[[ ]]`, assim como seus equivalentes `test` e `[ ]`, não executa absolutamente nada! Aliás, a descrição que eu mostrei aqui anteriormente vem exatamente desses dois comandos herdados do Bourne Shell (o shell "sh" original):

```
:~$ help -d test
test - Avalia uma expressão condicional.

:~$ help -d [
[ - Avalia uma expressão condicional.
```

> *Eu espero que você tenha entendido que essa minha "bronca", embora legítima, foi apenas um recurso para fixar os conceitos que estamos trabalhando e para mostrar que devemos manter sempre um espírito crítico em relação a tudo, especialmente o que chega até nós com ares de autoridade.*

2.2.10 – Voltando ao link simbólico do nosso script

Talvez você tenha até se esquecido, mas eu estou aqui para lembrá-lo de que estamos tentando verificar se a criação do link simbólico para o arquivo `infos.sh` foi bem-sucedida utilizando apenas os recursos do próprio Bash. Dentre as várias possibilidades, a nossa opção será o comando composto `[[ ]]` – entendeu agora por que demos tantas voltas?

Alternativamente, nós podemos chegar aos mesmos resultados com os comandos `test` e `[ ]`, mas o foco deste livro é o Bash, então é bom nós começarmos a nos acostumar com os *bashismos*.

> *Os comandos `test` e `[ ]` são "sinônimos", ou seja, eles são 100% intercambiáveis, ao passo que o comando `[[ ]]` permite o uso de recursos adicionais exclusivos do Bash.*

A coisa mais empolgante do comando `[[ ]]` é que, além dos operadores de comparação entre strings e números, ele nos oferece diversos outros operadores especializados em arquivos! Ou seja, se eu quiser saber se o arquivo `~/projetos/infos/infos.sh` existe, basta criar uma expressão e testá-la:

Exemplo 2.24 – Testando a existência de um arquivo.

```
:~$ [[ -f ~/projetos/infos/infos.sh ]]; echo $?
0
```

Aqui, o operador *unário* (que recebe apenas um termo) `-f` fará com que a expressão seja avaliada como verdadeira se o arquivo existir, e isso leva o comando `[[ ]]` a ser encerrado com estado `0` (*sucesso*).

Pelo bem da sua sanidade mental, jamais leia esse tipo de comando como *"se o arquivo tal existir..."* Em vez disso, prefira ler a expressão como uma afirmação: *"o arquivo tal existe"*, e deixe que o comando decida se isso é verdade ou não.

Nós podemos utilizar o mesmo operador para testar a existência do link que nós criamos no *tópico 2.2.4*:

Exemplo 2.25 – Testando a existência de um arquivo de link simbólico.

```
:~$ [[ -f ~/bin/infos ]]; echo $?
0
```

Como vimos, o arquivo `~/bin/infos` existe (estado de saída `0`), mas nós não vimos tudo. Para começar, quando o arquivo testado é um link simbólico, quem está realmente sendo verificado é o arquivo para o qual ele aponta. Isso pode fazer diferença caso exista um arquivo com o mesmo caminho e nome do link que pretendíamos criar. Então, precisamos garantir que o arquivo encontrado seja mesmo um link simbólico, e o comando `[[ ]]` tem outro operador unário para isso:

Exemplo 2.26 – Testando se um arquivo existe e é um link simbólico.

```
:~$ [[ -L ~/bin/infos ]]; echo $?
0
```

Sucesso!

> *Para obter uma lista completa dos operadores disponíveis para uso em expressões no comando* `[[ ]]`*, e em seus equivalentes herdados do shell "sh", basta abrir um terminal e executar o comando* `help test`*. Os operadores disponíveis apenas para o Bash podem ser encontrados com o comando* `help [[`*.*

Agora, veja que interessante: nós mal aprendemos a criar um script e já temos quase todo o conhecimento necessário para não termos que pensar nisso nunca mais!

2.3 – Um script para criar scripts

O script que vamos estudar aqui é o mesmo que eu utilizo quando quero criar rapidamente um novo projeto em Bash. Mas é preciso que você conheça um pouco mais as minhas "manias" para entender as escolhas que eu fiz na criação do programa.

2.3.1 – A escolha do editor

Para começar, eu gosto de escrever meus scripts em Bash em dois editores: o *Vim* e o *Nano*, ambos disponíveis na *interface da linha de comando* (CLI). O motivo da minha escolha é simples: se eu vou testar e utilizar meus programas no terminal, nada mais conveniente do que escrevê-los no terminal.

No entanto, por questões de facilidade e padronização, sempre que precisarmos exemplificar algo neste livro, nós utilizaremos o editor *Nano*, e

aqui estão todos os atalhos que você precisa conhecer neste momento para trabalhar com ele:

Atalho	**O que faz...**
`CTRL` + `S`	Salva o arquivo aberto.
`CTRL` + `0`	Salva o arquivo aberto com outro nome.
`CTRL` + `C`	Cancela a operação em curso.
`CTRL` + `X`	Sai do editor.
`CTRL` + `K`	Recorta a linha atual ou a seleção.
`ALT` + `6`	Copia a seleção atual.
`CTRL` + `U`	Cola o conteúdo recortado ou copiado.
`ALT` + `#`	Alterna a exibição da numeração de linha.
`ALT` + `X`	Alterna a exibição da ajuda.
`ALT` + `U`	Desfaz as últimas ações.
`ALT` + `E`	Refaz as ações desfeitas.

Existem mais atalhos e até configurações bem interessantes, mas estes são suficientes para você começar a se divertir, porque agora nós temos que falar sobre a minha segunda "decisão de projeto": onde salvar os modelos gerados pelo nosso programa.

2.3.2 – Padronizando uma pasta de projetos

Como já foi dito neste capítulo, eu prefiro tratar cada um dos meus programas como um projeto que tem a sua própria pasta dentro de uma pasta chamada `projetos`. Portanto, cada novo script gerado pelo nosso programa deverá criar uma pasta em `projetos` com o nome do script. Por exemplo, se o novo

projeto se chamar “infos”, deverão ser criados um novo diretório e um novo arquivo no caminho:

```
/home/blau/projetos/infos/infos.sh
                      ↑      ↑
           novo diretório  novo arquivo
```

2.3.3 – Tornando o arquivo executável

Você também já sabe que a minha pasta de arquivos executáveis é o diretório `~/bin`, porque foi assim que eu defini na variável `PATH`. Eu também disse que eu prefiro fazer uma ligação simbólica nesta pasta em vez de mover ou copiar meus programas para lá. Sendo assim, o nosso programa deverá dar permissão para que o script gerado seja executável e criar uma ligação simbólica em `~/bin` apontando para ele, o que pode ser feito com os comandos (ainda usando o projeto “infos” como exemplo):

```
chmod +x ~/projetos/infos/infos.sh
ln -s ~/projetos/infos/infos.sh ~/bin/infos
```

2.3.4 – Verificações e tratamento de erros

Segundo uma das leis de Murphy...

> *“Se algo pode dar errado, dará errado – e dará da pior forma, no pior momento, e de modo a causar o maior dano possível.”*
>
> *– Edward Murphy*

O segredo, então, é tentar prever o que pode dar errado e elaborar dispositivos para lidar com essas situações. No mínimo, os seguintes possíveis problemas precisam ser antecipados:

- Nomes de projetos com caracteres inválidos;
- Nomes de projetos já existentes;
- Nome do projeto não informado.

Por padrão, qualquer um desses problemas fará o programa ser terminado com estados de saída diferentes de `0` (erro) e a exibição de uma mensagem informando erro ocorrido conforme a tabela abaixo:

Erro	Mensagem
1	Nome do projeto não informado.
2	O nome do projeto contém caracteres inválidos.
3	Já existe um projeto com o nome [nome do projeto].

Além disso, toda mensagem de erro deve ser seguida de uma ajuda sobre o uso do programa, por exemplo:

```
:~$ nb
Nome do projeto não informado!
Uso: nb nome-do-projeto
:~$
```

2.3.5 – Como o programa será utilizado

O nosso programa será chamado de `nb` (de "*novo*" e "*bash*"). Para utilizá-lo, nós abriremos um terminal e executaremos:

```
:~$ nb nome-do-novo-projeto
```

É importante que o nome do novo projeto seja formado apenas por letras minúsculas, números e os caracteres traço (`-`) e sublinhado (`_`), sem espaços, acentos, pontuação ou outros símbolos gráficos.

> *O shell tem poucas limitações quanto aos nomes dos arquivos, mas a decisão de evitar caracteres além dos especificados tem a ver com boas práticas na organização de projetos e arquivos.*

2.3.6 – O modelo do novo script

Como última decisão de projeto, quando um novo script for gerado, ele já deverá conter um cabeçalho completo segundo o modelo abaixo:

```
#!/usr/bin/env bash
# ------------------------------------------------
# Projeto  : (automático)
# Arquivo  : (automático)
# Descrição:
# Versão   : 0.0.0
# Data     : (automático)
# Autor    : Blau Araujo <blau@debxp.org>
# Licença  : GNU/GPL v3.0
# ------------------------------------------------
# Uso:
# ------------------------------------------------
```

Repare que três campos deverão ser preenchidos automaticamente no momento em que o novo projeto for criado.

2.3.7 – O que falta saber

Observando todas as nossas decisões, nós só precisamos descobrir:

- Como capturar o nome do projeto passado como parâmetro de execução do nosso programa;
- Como saber se o utilizador informou um nome para o novo projeto;
- Como validar o nome do novo projeto;

- Como verificar se a pasta do novo projeto já existe;
- Como criar a pasta do projeto;
- Como obter as informações automáticas do modelo;
- Como criar o arquivo do script já com o conteúdo do modelo;
- Como abrir o editor após a criação do arquivo...

Parece muita coisa, mas... Ei, pelo menos já sabemos tornar arquivos executáveis e criar links simbólicos!

Falando sério, a coisa é bem mais simples do que parece e, como fizemos todas as vezes neste livro, nós vamos partir do conhecido até chegarmos ao desconhecido.

2.3.8 – Recebendo dados do usuário como parâmetros

No *tópico 1.5*, nós utilizamos a variável interna `0` que, dependendo do contexto, poderia conter o nome do shell em uso ou o nome do script que está sendo executado. Pois bem, o contexto a que nos referimos é o modo de execução do shell – no modo interativo, ela contém o nome do shell; no modo não-interativo, é o caminho é o nome do script.

Vamos testar isso na prática já escrevendo o nosso programa *nb*.

A primeira coisa a fazer é criar a pasta `~/projetos/nb`:

Exemplo 2.27 – Criando a pasta do programa 'nb'.

```
:~$ mkdir -p ~/projetos/nb
:~$ cd ~/projetos/nb
:~/projetos/nb$
```

A função do utilitário `mkdir` é criar diretórios. Com a opção `-p`, ele muda o seu comportamento padrão, para permitir a criação de um diretório filho (`nb`, no caso) no mesmo comando em que criamos o diretório pai (`projetos`). Aliás,

o `mkdir` poderia interromper o nosso comando com erro em duas situações: tanto se a pasta `~/projetos` *não existisse*, quanto no caso da pasta `nb` *já existir*, e a opção `-p` evita que isso aconteça: se `~/projetos` não existir, ele será criado; se `nb` existir, isso não causará um erro.

É bem provável que você já conheça o comando seguinte, `cd`. O que talvez seja novidade é o fato dele também ser um comando interno do shell, diferente de outros "comandos" muito utilizados, como o `ls` e o `mkdir`, que são programas utilitários.

Já na pasta do projeto `nb`, chegou a hora de criarmos o arquivo do nosso script, o que pode ser feito de várias formas. Talvez a forma mais comum seja através do próprio editor (o *Nano*, no nosso caso):

Exemplo 2.28 – Criando um novo arquivo com o Nano.

```
:~/projetos/nb$ nano nb.sh
```

Mas, não faça isso ainda!

Embora eu considere esta a forma mais prática de criar arquivos para edição, existem outros métodos que poderão nos ajudar a criar arquivos através de scripts, e nós já vimos uma dessas formas no *tópico 1.9.5*, quando falamos da possibilidade de criarmos arquivos de *log* (registros de históricos e erros) com os redirecionamentos para arquivos.

Na ocasião, ficou demonstrado que nós podemos enviar dados nos fluxos de saída (*stdout* e *stderr*) para arquivos com os operadores de redirecionamento para arquivos. Também vimos que existem duas formas de fazer esse redirecionamento: com o operador `>`, que apaga o arquivo de destino antes de enviar os dados, ou com o operador `>>`, que posiciona o ponteiro de escrita no final do arquivo e só começa a inserir dados a partir daí. Em comum, ambos têm a capacidade de criar o arquivo de destino caso ele não exista.

A questão aqui é: o que acontece se utilizarmos os redirecionamentos da saída para arquivos *sem um fluxo de saída*?

Se a sua resposta foi: *"a ausência de dados também é um dado"*, você acertou, porque é assim que os redirecionamentos funcionam!

Outra pergunta: sem utilizar estruturas nem operadores condicionais, como poderíamos testar a existência de um arquivo e só criá-lo se ele não existir?

Se você respondeu: com o operador de redirecionamento `>>`, meus parabéns, você é um *shelleiro*[9] nato!

Veja o exemplo:

Exemplo 2.29 – Criando um arquivo vazio com o redirecionamento '>>'.

```
:~/projetos/nb$ >> nb.sh
```

Se o arquivo `nb.sh` existir, nada será alterado, já que não há dados para anexar ao seu final. Caso não exista, ele será criado, mas nada será escrito.

Mas, também não faça isso ainda!

Foi muito interessante saber que podemos criar arquivos vazios de forma segura com o redirecionamento `>>`, mas será que nós queremos mesmo um arquivo *vazio*? Afinal de contas, se nós sempre teremos que escrever a linha do interpretador nos nossos programas, por que não aproveitar o redirecionamento para fazer isso de uma vez?

Exemplo 2.30 – Criando um novo script com o redirecionamento '>>'.

```
:~/projetos/nb$ echo '#!/usr/bin/env bash' >> nb.sh
```

Aqui, o comando `echo` enviaria a string da *shebang* para a saída padrão, mas ela está sendo redirecionada para o final do arquivo inexistente `nb.sh`. Ou seja, em vez da string aparecer no terminal, ela será escrita no nosso novo script – perfeito!

9 ***Shelleiro*** é um termo lúdico que eu aprendi com o professor Júlio Neves e se refere aos aficionados pelos recursos nativos do shell.

Mas... Mas nada, pode fazer isso agora.

> ***Repare bem:*** *a string da shebang está entre aspas simples porque isso impede que o shell faça expansões ou identifique algo como um comando, o que é especialmente crítico no terminal, já que o ponto de exclamação (`!`) pode estar configurado para dar acesso ao histórico de comandos.*

Antes de abrirmos o script `nb.sh`, nós vamos aproveitar para torná-lo executável:

Exemplo 2.31 – Tornando o script 'nb.sh' executável.

```
:~/projetos/nb$ chmod +x nb.sh
```

Agora sim, com o script aberto no editor, vamos escrever um comando para exibir o conteúdo da variável `0`:

```
#!/usr/bin/env bash

echo $0
```

Salve o arquivo (`Ctrl`+`S`), saia do editor (`Ctrl`+`X`), e execute o script:

Exemplo 2.32 – Expansão de '$0' a partir de um script.

```
:~/projetos/nb$ ./nb.sh
./nb.sh
```

Como esperávamos, a saída foi o caminho (`./`) e o nome do script. Mas, o mais interessante ainda não é isso. Veja o que acontece se você passar o script como um parâmetro do executável `bash`:

Exemplo 2.33 – Expansão de '$0' quando o script é chamado pelo 'bash'.

```
:~/projetos/nb$ bash nb.sh
nb.sh
```

Para entender o que está acontecendo, é preciso saber de mais alguns detalhes. Por exemplo, nós já dissemos que a variável `0` pertence a um conjunto de variáveis especiais do shell chamadas de *parâmetros posicionais*. Como toda enumeração no shell começa de zero, é fácil deduzir que a variável `0` contém o primeiro parâmetro passado para...

Para quem?

É aí que está o pulo do gato!

Quando executamos um script, *quem realmente está sendo executado é o shell!* No nosso caso, é o Bash *em modo não-interativo*, e o nome do script entra como seu primeiro parâmetro (`0`), que é o caminho até o arquivo que contém os comandos que deverão ser interpretados.

Isso quer dizer que, se nós passarmos um parâmetro na chamada do script, ele será o parâmetro `1`. Do nosso script? Não, será o parâmetro `1` da sessão não-interativa do shell iniciada pelo nosso script. Vamos testar isso alterando o nosso código:

```
#!/usr/bin/env bash

echo Parâmetro 0: $0 - Parâmetro 1: $1
```

Executando na linha de comando...

Exemplo 2.34 – Expandindo um segundo parâmetro posicional.

```
:~/projetos/nb$ ./nb.sh banana
Parâmetro 0: ./nb.sh - Parâmetro 1: banana
```

Muito bem, essa foi fácil. Então, vamos alterar o script `nb.sh` para que ele possa expandir mais parâmetros posicionais. Só que, desta vez, não precisamos mais ver o que tem no parâmetro `0` – afinal, ele sempre será o nome do script.

```
#!/usr/bin/env bash

echo $1 - $2 - $3
```

Testando no terminal:

```
:~/projetos/nb$ ./nb.sh banana laranja abacate
banana - laranja - abacate
```

Perfeito! Reparou que já somos capazes de passar dados para o script através da linha de comandos? Vamos tentar outra vez:

```
:~/projetos/nb$ ./nb.sh casa verde casa amarela
casa - verde - casa
```

Estranho? Eu não acho (nem o Bash). Veja bem, nós passamos quatro parâmetros na chamada do nosso script:

```
casa verde casa amarela
  1    2     3     4
```

Para o shell, os parâmetros do script serão cada uma das strings que aparecerem delimitadas por espaços. Se o que pretendíamos passar era "casa verde" e "casa amarela", os parâmetros precisam estar entre aspas:

Exemplo 2.35 – Passando strings com espaços como parâmetros.

```
:~/projetos/nb$ ./nb.sh 'casa verde' 'casa amarela'
casa verde - casa amarela -
```

Funcionou, mas há outro problema aqui: nosso script prevê a passagem de três parâmetros posicionais, mas nós só passamos dois, o que fez com que um traço ficasse sobrando na saída exibida:

```
casa verde - casa amarela –
```

Isso poderia ser “maquiado” de várias formas, mas nós podemos aproveitar a oportunidade para conhecermos um pouco mais do que o comando composto `[[ ]]` tem para oferecer.

Para efeito de demonstração, imagine que nosso script realmente deve receber qualquer quantidade de parâmetros entre um e três – como podemos verificar se eles foram passados antes de tentar exibi-los?

Dentre várias as formas possíveis, existe a possibilidade de fazermos testes:

```
#!/usr/bin/env bash

[[ –n $1 ]] && echo $1
[[ –n $2 ]] && echo – $2
[[ –n $3 ]] && echo – $3
```

Atenção! *Os espaços depois de* `[[` *e antes de* `]]` *são obrigatórios!*

Aqui, o operador unário `–n` testa se a variável está ou não vazia. Assim, a exibição de cada parâmetro passado fica condicionada ao estado de saída do comando `[[ ]]`, que será sucesso apenas se o parâmetro testado existir.

Por padrão, o comando `[[ ]]` sempre avaliará qualquer string entre os colchetes como sucesso e qualquer string vazia como erro:

```
:~$ [[ '' ]]; echo $?
1
:~$ [[ banana ]]; echo $?
0
```

Portanto, nós podemos dispensar o operador `–n`:

```
#!/usr/bin/env bash

[[ $1 ]] && echo $1
[[ $2 ]] && echo - $2
[[ $3 ]] && echo - $3
```

Vamos testar o script:

```
:~/projetos/nb$ ./nb.sh banana
banana
:~/projetos/nb$ ./nb.sh banana laranja
banana - laranja
:~/projetos/nb$ ./nb.sh banana laranja abacate
banana - laranja - abacate
```

> *Certamente, não é o código mais bonito do mundo, mas funciona como demonstração.*

Existem outras variáveis internas do Bash que podem facilitar muito a nossa vida quando trabalhamos com parâmetros posicionas, que é o caso da variável `#`, que armazena o número de parâmetros passados para o script, e das variáveis `*` e `@`, que armazenam os parâmetros.

> *Nenhuma dessas três variáveis inclui o parâmetro na variável `0`, ou seja, tanto a contagem quanto a listagem dos parâmetros começa com o parâmetro na variável `1`.*

Visualmente, não existe nenhuma diferença no conteúdo das variáveis `*` e `@`, mas ela existe.

Vamos experimentar com o nosso script:

```
#!/usr/bin/env bash
```

```
# Experimento 1

echo "Total de parâmetros: $#"
echo "Parâmetros (*): $*"
echo "Parâmetros (@): $@"
```

Neste primeiro experimento, nós vamos apenas expandir as variáveis para observar o que é exibido:

```
:~/projetos/nb$ ./nb.sh a b 'c d'
Total de parâmetros: 3
Parâmetros (*): a b c d
Parâmetros (@): a b c d
```

Como esperávamos, não há nada que denuncie visualmente alguma diferença entre os conteúdos de `@` e `*`. Essa diferença só será notada numa situação: quando a expansão for feita entre aspas duplas (`"$@"` ou `"$*"`). Mesmo assim, nós só conseguiremos ver o que acontece se pudermos, a partir da expansão dessas duas variáveis, obter cada uma das strings passadas como parâmetros para o script. Uma alternativa para isso é o comando composto `for`.

O `for` é uma das estruturas de repetição disponíveis no Bash, e o seu uso geral é muito simples:

```
for VAR in LISTA; do
    COMANDOS
done
```

Para cada elemento em `LISTA`, haverá uma iteração em que este elemento será armazenado temporariamente na variável `VAR` e o bloco de `COMANDOS` será executado.

> *Os elementos da* `LISTA` *funcionam como os parâmetros passados para um script, ou seja, são uma lista de valores separados por espaços.*

Vamos observar como isso funciona no script:

```
#!/usr/bin/env bash

# Experimento 2

for v in a b c d; do
    echo $v
done
```

Cada valor na lista `a b c d`, um de cada vez, será armazenado na variável `v` e fará com que o comando echo seja executado.

Portanto...

```
:~/projetos/nb$ ./nb.sh
a
b
c
d
```

Se quisermos que uma string contendo espaços seja tratada como um único elemento, basta escrevê-la entre aspas:

```
#!/usr/bin/env bash

# Experimento 3

for v in a b 'c d'; do
    echo $v
done
```

O que produziria na saída...

```
:~/projetos/nb$ ./nb.sh
a
```

```
b
c d
```

Agora que sabemos como funciona o `for`, vamos deixar que a lista seja obtida pela expansão dos parâmetros posicionais armazenados nas variáveis `@` e `*`:

```
#!/usr/bin/env bash

# Experimento 4

echo 'Variável *'

for i in $*; do
    echo $i
done

echo 'Variável @'

for j in $@; do
    echo $j
done
```

Testando o script...

```
:~/projetos/nb$ ./nb.sh a b 'c d'
Variável *
a
b
c
d
Variável @
a
b
c
d
```

Perfeito!

Agora podemos ver se as aspas fazem realmente diferença. Vamos ao quinto experimento:

```
#!/usr/bin/env bash

# Experimento 5

echo 'Variável * entre aspas'

for i in "$*"; do
    echo $i
done

echo 'Variável @ entre aspas'

for j in "$@"; do
    echo $j
done
```

Executando o script...

```
:~/projetos/nb$ ./nb.sh a b 'c d'
Variável * entre aspas
a b c d
Variável @ entre aspas
a
b
c d
```

Percebeu o que aconteceu aqui? As aspas duplas em volta da expansão `"$*"` fizeram com que a toda a lista expandida fosse tratada como um só elemento, enquanto que, na expansão `"$@"`, todos os elementos continuaram sendo tratados individualmente, inclusive o terceiro parâmetro, `'c d'`, que teve a sua unidade preservada.

Agora nós já sabemos capturar os dados passados para o nosso script e como obter informações sobre eles. Também descobrimos que o comando `[[ ]]`

pode nos ajudar com a validação desses dados, e isso já nos permite começar a escrever a primeira parte do programa:

```
#!/usr/bin/env bash

# Valida o número de parâmetros...
if [[ $# -ne 1 ]]; then
 echo 'Número incorreto de parâmetros!'
 exit 1
fi
```

2.3.9 – Explorando igualdades e desigualdades

Uma coisa importante que precisamos saber sobre os comandos `test`, `[ ]` e `[[ ]]`, é que eles utilizam operadores diferentes para comparar strings e valores numéricos. Isso é necessário porque, no Bash (ou em qualquer shell *unix-like*), todos os dados são do tipo *indeterminado*, e essa indeterminação vem do fato que nós já destacamos diversas vezes neste livro: *cada linha de um programa escrito em Bash é um comando do shell*.

Repare bem nas implicações dessa afirmação: uma *linha de comando* é uma instrução (uma ordem) que é dada ao sistema operacional, e essa instrução sempre será composta pela chamada a uma função interna do shell (que é o que nós costumamos chamar de *comando*) ou a um programa. Adicionalmente, a linha de comando também pode conter os argumentos que serão passados para esta função ou este programa, mas tudo isso é apenas uma cadeia de caracteres, uma string, que será passada para o kernel do sistema operacional. Consequentemente, nenhuma função do shell, nenhum programa, independente da linguagem em que seja escrito, jamais receberá da linha de comando algo que não seja uma string.

Na prática, portanto, dizer que *os dados são de tipo indeterminado* é o mesmo que dizer: *para o shell, todos os dados são cadeias de caracteres, logo, strings*. Eventualmente, um programa ou uma função do shell poderá contextualizar

os dados e processar uma conversão de tipos, mas os comandos `test`, `[ ]` e `[[ ]]` precisam ser capazes de avaliar expressões que comparem dados conforme os valores que eles representam.

Por exemplo, numericamente, `5` e `05` expressam o mesmo valor, mas são strings diferentes. Se você não conhece as peculiaridades do Bash, tendo alguma experiência em programação, certamente você tentaria avaliar uma comparação de igualdade com o operador `==`, mas veja o que aconteceria:

```
:~$ [[ 5 == 05 ]]; echo $?
1
```

O estado de saída foi `1` (*erro*), não porque o Bash não sabe matemática básica, mas porque você deu o comando errado. Acontece que existem três operadores de comparação de igualdade para uso com o comando composto `[[ ]]`:

```
STRING1 = STRING2
STRING1 == PADRÃO (ou STRING2 nos comandos 'test' e '[ ]')
INTEIRO1 -eq INTEIRO2
```

Os três resultando no valor de *verdadeiro* caso os termos da esquerda sejam iguais aos termos da direita.

> *Nos comandos `test` e `[ ]`, herdados do shell 'sh', o operador `==` tem a mesma função do operador `=`, ou seja, ele só compara strings.*

Então, se o nosso objetivo era comparar numericamente `5` com `05`, ambos inteiros, o operador correto seria `-eq` (de *equal*, em inglês):

Exemplo 2.36 – Comparando valores numéricos com o comando '[[ ]]'.

```
:~$ [[ 5 -eq 05 ]]; echo $?
0
```

Repare, porém, que `5` e `05` não deixaram de ser strings! Aliás, a expressão inteira é só uma lista de argumentos do comando composto `[[ ]]`, o que fica ainda mais claro quando usamos o comando `test`:

Exemplo 2.37 – Comparando valores numéricos com o comando 'test'.

```
:~$ test 5 -eq 05; echo $?
0
```

Voltando à primeira parte do nosso script, nós utilizamos outro operador de comparação numérica, desta vez, para avaliar uma *não-igualdade*: o operador `-ne` (do inglês *not equal*). ,

```
[[ $# -ne 1 ]]
```

Aqui, a ideia é comparar o número total de argumentos passados na chamada do script (obtido com a expansão da variável `#`) com a quantidade mínima de parâmetros esperados, ou seja, um parâmetro posicional obrigatório correspondendo ao nome do projeto. Se o valor em `#` for diferente (*not equal*) de `1`, a expressão será avaliada como *verdadeira* e o comando sairá com *sucesso*. Aqui está uma tabela com os operadores de comparação numérica:

`-eq`	É igual.	`-ne`	Não é igual.
`-lt`	É menor.	`-gt`	É maior.
`-le`	É menor ou igual.	`-ge`	É maior ou igual.

O próximo passo será a validação do nome do projeto. Como dissemos, ele só pode ter letras minúsculas sem acentos, números e os caracteres traço (`-`) e sublinhado (`_`). Para nossa sorte, o Bash nos oferece formas relativamente simples de fazer esse tipo de validação contando apenas com seus recursos nativos – tudo depende da nossa capacidade de identificar e expressar *padrões*.

2.3.10 – Comparando padrões e expressões regulares

Como vimos, os operadores de comparação de strings são diferentes dos operadores de comparação numérica. Também existem algumas diferenças entre a interpretação desses operadores pelos comandos `test` e `[ ]` e o comando composto `[[ ]]`. Veja na tabela abaixo:

Expressão	Comando '[[ ]]' (Bash)	Comandos 'test' e '[ ]'
`Str1 = Str2`	As strings são iguais.	As strings são iguais.
`Str1 == Str2`	São iguais, mas Str2 é um *padrão*.	As strings são iguais.
`Str1 != Str2`	São diferentes, mas Str2 é um *padrão*.	As strings são diferentes.
`Str1 =~ REGEX`	Str1 "casa" com a *expressão regular*.	Erro!
`Str1 < Str2`	Str1 é ordenada antes de Str2.	Str1 é ordenada antes de Str2.
`Str1 > Str2`	Str1 é ordenada após Str2.	Str1 é ordenada após Str2.
`-z Str`	A string é vazia.	A string é vazia.
`-n Str`	A string não é vazia.	A string não é vazia.

No Bash, como podemos ver, os operadores `==` e `!=` comparam a string da esquerda com o padrão descrito à direita. O padrão pode até ser uma string literal, mas o comando `[[ ]]` sempre verá o segundo termo como um padrão que segue as mesmas regras da *geração de nomes de arquivos* (aqueles caracteres "coringa" que usamos quando queremos listar vários arquivos com algum padrão em comum). Além disso, o operador `=~`, que só é aplicável ao comando `[[ ]]`, faz a comparação da string à esquerda com uma *expressão*

regular, que é uma forma bem mais poderosa de representação de padrões de texto.

> ***Atenção!*** *É muito comum encontrarmos tutoriais e postagens de fóruns solucionando problemas supostamente em Bash com o comando* `[ ]`*, que funciona no Bash, mas é herdado do shell 'sh' e avalia strings de forma diferente.*

Considere, por exemplo, as strings `banana` e `bandana`. Vamos fazer alguns testes com elas:

Exemplo 2.38 – Testando a igualdade de duas strings.

```
:~$ a=banana; b=bandana
:~$ [[ $a == $b ]]; echo $?
1
```

O que aconteceu aqui?

Para começar, nunca se esqueça de que o shell sempre fará as expansões cabíveis antes de executar o comando. Então, o comando do exemplo acima será interpretado assim:

```
[[ banana == bandana ]]
```

O que, obviamente é falso e fará com que o comando saia com estado de *erro* (1). Contudo, como estamos utilizando o comando `[[ ]]`, o segundo termo da expressão é tratado como um padrão – neste caso, um padrão indicando a presença obrigatória dos caracteres `b-a-n-d-a-n-a` em suas respectivas posições de ocorrência na string. Então, vamos modificar um pouco o exemplo e testar novamente:

Exemplo 2.39 – Testando a correspondência de uma string ao padrão.

```
:~$ a=banana; b=bandana
:~$ [[ $a == ban* ]]; echo $?
0
:~$ [[ $b == ban* ]]; echo $?
0
```

Pelas regras da geração de nomes de arquivos (também chamadas de *globbing*), um asterisco (`*`) no padrão significa *zero ou mais caracteres quaisquer*. Ou seja, o padrão representa qualquer string que comece com os caracteres `b-a-n`, nesta ordem, seguidos de zero ou mais caracteres quaisquer. Portanto, o padrão "casa" com `ban-ana` e `ban-dana`, o que resulta em *sucesso* (`0`) nos dois comandos.

Veja como a ausência de um caractere também valida ambas as strings:

```
:~$ a=banana; b=bandana
:~$ [[ $a == ban*ana ]]; echo $?
0
:~$ [[ $b == ban*ana ]]; echo $?
0
```

Aliás, a comparação abaixo pode dar um nó na cabeça de quem ainda não entendeu as diferenças entre o comando `[[ ]]` e os comandos `test` e `[ ]`:

```
:~$ [[ a == ? ]]; echo $?
0
```

A presença do caractere interrogação (`?`), na geração de nomes de arquivos, representa *um único caractere existente qualquer*, e a string `a` tem exatamente um caractere, o que fez o comando sair com *sucesso*.

Se fosse com o comando `test`, a interrogação seria tratada como uma string que é literalmente uma interrogação:

```
:~$ [ a == ? ]; echo $?
1
```

Viu o que eu disse sobre tomar cuidado com o que você encontra na internet?

Além de divertida para *bugar* a mente dos seus colegas, esta é uma boa forma de testar se a string na variável tem 'n' caracteres:

Exemplo 2.40 – Testando se a string tem 'n' caracteres.

```
:~$ a=b; b=cd
:~$ [[ $a == ? ]]; echo $?
0
:~$ [[ $b == ? ]]; echo $?
1
:~$ [[ $a == ?? ]]; echo $?
1
:~$ [[ $b == ?? ]]; echo $?
0
```

"Boa", mas existem formas bem melhores.

Com os padrões de formação de nomes de arquivos, nós também podemos representar um conjunto de caracteres que são válidos em determinada posição da string. Isso é feito a partir da definição de uma *lista* de caracteres entre colchetes, também chamada de *classe de caracteres.* Que tal sofisticarmos um pouco os nossos exemplos com um `for` para vermos como isso funciona?

Exemplo 2.41 – Testando se a string contém um dos caracteres da lista.

```
:~$ txt='porta porto porte'
:~$ for s in $txt; do [[ $s == *[ae] ]] && echo $s; done
porta
porte
```

Só para relembrar, a variável `txt` será expandida para a string que contém as palavras `porta`, `porto` e `porte` separadas por espaço, o que é uma lista válida de strings para o comando composto `for`. Então, a cada *loop*, uma dessas palavras será armazenada na variável `s` e testada. Se "casar" com o padrão, ela será exibida.

Mas, o que o padrão `*[ae]` representa?

Como vimos, `*` corresponde a zero ou mais caracteres quaisquer, ao passo que a lista `[ae]`, na última posição da string, diz que o último caractere tem que ser `a` ou `e`, condição que só é satisfeita pelas strings `porta` e `porte`.

Uma coisa muito interessante sobre as classes de caracteres, é que elas permitem a representação de faixas de caracteres da *tabela ASCII* (mais sobre isso em `man ascii`, no seu terminal). Por exemplo, para indicar o intervalo dos caracteres minúsculos entre `a` e `z`, nós escrevemos `a-z`, se forem os maiúsculos, `A-Z`, e se forem dígitos de `0` a `9`, `0-9`.

Observe:

Exemplo 2.42 – Representando faixas de caracteres na lista.

```
:~$ var='33 3a 35 39'
:~$ for s in $var; do [[ $s == 3[0-9] ]] && echo $s; done
33
35
39
```

Aqui, a string `3a` não "casou" com o padrão porque o caractere `a` não aparece na tabela ASCII entre os caracteres `0` e `9`. Mas, poderia ser o contrário:

```
:~$ var='33 3a 35 39'
:~$ for s in $var; do [[ $s == 3[a-z] ]] && echo $s; done
3a
```

Agora, só a string `3a` contém um caractere na faixa entre `a` e `z` na última posição.

Um detalhe Importantíssimo: as faixas de caracteres na formação de nomes de arquivos só obedecem à sequência de caracteres da tabela ASCII devido a uma opção do Bash que costuma vir habilitada por padrão na maioria das distribuições GNU/Linux (tanto para o modo interativo quanto para o não-interativo), a opção `globasciiranges`.

Você pode conferir se ela está habilitada no seu sistema utilizando o comando interno `shopt`:

Exemplo 2.43 – Verificando a opção 'globasciiranges'.

```
:~$ shopt globasciiranges
globasciiranges on
```

> *Mais importante ainda é você saber que esta opção não afeta faixas de caracteres em expressões regulares, o que nos deixa à mercê das coleções de caracteres nas nossas definições de localidade!*

Quando trabalhamos com faixas de caracteres, especialmente com strings que contenham caracteres não utilizados na língua inglesa, é muito importante ter em mente que a tabela ASCII não contém caracteres acentuados nem cedilha. Por exemplo:

```
:~$ var=maçã
:~$ [[ $var == ma[a-z][a-z] ]]; echo $?
1
```

Não houve casamento com o padrão porque `ç` e `ã` não existem na tabela ASCII.

2.3.11 – As classes POSIX

Felizmente, existe uma série de implementações POSIX de faixas e caracteres que seriam quase impossíveis de representar numa lista – são as chamadas *classes POSIX*. A tabela abaixo mostra algumas delas:

Classe POSIX	Significado
`[:alnum:]`	Todos os caracteres alfanuméricos (inclusive os acentuados e com cedilha).
`[:alpha:]`	Apenas os caracteres alfabéticos.
`[:digit:]`	Apenas caracteres numéricos.
`[:word:]`	Apenas letras, números e o sublinhado (`_`).
`[:lower:]`	Apenas caracteres alfabéticos minúsculos.
`[:upper:]`	Apenas caracteres alfabéticos maiúsculos.
`[:blank:]`	Espaços e tabulações.
`[:space:]`	Espaços, tabulações e quebras de linha.
`[:punct:]`	Apenas caracteres de pontuação.
`[:cntrl:]`	Apenas caracteres de controle.

Graças a isso, nós podemos conseguir um casamento com a string do exemplo anterior:

Exemplo 2.44 – Utilizando classes POSIX nas listas.

```
:~$ var=maçã
:~$ [[ $var == ma[[:alpha:]][[:alpha:]] ]]; echo $?
0
```

Os padrões com base na formação de nomes de arquivos são muito poderosos, mas carecem de um recurso que nos permita quantificar as ocorrências dos caracteres, daí a necessidade de utilizar uma classe para cada caractere que precisamos representar no exemplo.

> *Esta, entre outras, é uma das grandes diferenças entre esse tipo de representação de padrões e as expressões regulares.*

As classes também podem ser negadas se o primeiro caractere na lista for um circunflexo (`^`) ou uma exclamação (`!`). Por exemplo:

Exemplo 2.45 – Negando classes de caracteres.

```
:~$ var='33 3a 35 39'
:~$ for s in $var; do [[ $s == 3[^0-9] ]] && echo $s; done
3a
:~$ for s in $var; do [[ $s == 3[!a-z] ]] && echo $s; done
33
35
39
```

> *Como `^` e `!` possuem significado especial numa classe de caracteres, sempre que precisarmos representá-los literalmente, eles terão que ser escapados, o que é feito com uma barra invertida (`\`), também conhecida como "caractere de escape".*

A comparação de strings com padrões baseados na formação de nomes de arquivos, embora seja uma pista, não nos ajuda muito na validação de nomes de projeto segundo as exigências do nosso programa. Mas o comando `[[ ]]` pode trabalhar com o operador `=~`, que compara strings com uma expressão regular.

Não há muito espaço neste *pequeno manual* para abordarmos todo o assunto das expressões regulares (talvez, em um *grande manual*). Mas, o que veremos

aqui deve ser suficiente para você pegar o princípio da coisa e começar a fazer as suas próprias descobertas.

Para começar, os símbolos `*` e `?` possuem um significado diferente do que vimos até aqui. Nas expressões regulares (ou apenas *regex*), eles são *quantificadores*, ou seja, eles indicam quantas vezes o caractere à sua esquerda devem ocorrer na string: `*`, zero ou mais vezes, e `?`, zero ou uma. A classe de caracteres, por sua vez, é bem semelhante e, junto com o ponto (`.`), que representa um caractere qualquer, faz parte do conjunto de símbolos chamados de *representantes*.

2.3.12 – Algumas regras para entender de regex no Bash

A primeira regra para entender uma regex é justamente essa:

> ***Regra #1:*** *sem um quantificador, o representante diz que ao menos um caractere tem que existir na posição em que aparecer.*

Por exemplo, se a minha regex contiver o representante ponto (`.`) em dada posição, isso significa que eu espero encontrar apenas um caractere qualquer naquela posição. Já se o ponto for seguido do quantificador asterisco (`.*`), o casamento ocorrerá com zero ou mais caracteres quaisquer.

Simples, mas nada é tão evidente quanto parece, daí a nossa segunda regra:

> ***Regra #2:*** *uma regex não busca uma string que corresponda ao padrão, mas por correspondências ao padrão na string.*

Por exemplo, veja o que aconteceria se nos deixássemos levar apenas pela primeira regra:

```
:~$ var='a ab abc'
:~$ for s in $var; do [[ $s =~ . ]] && echo $s; done
```

```
a
ab
abc
```

O ponto “casa” com apenas um caractere existente qualquer, e todos os caracteres de todas as strings correspondem ao padrão buscado.

Mas as surpresas não param por aí se você não ficar atento. Vamos tentar especificar um pouco melhor o padrão buscado (e ser específico é sempre uma ótima ideia quando o assunto é uma regex):

```
:~$ var='a ab abc'
:~$ for s in $var; do [[ $s =~ .b ]] && echo $s; done
ab
abc
```

A segunda regra continua valendo, mas, se você esperava que apenas a string `ab` fosse exibida, o problema não é exatamente com a regex, mas com a *sua* interpretação do código. Leia novamente a segunda regra e pense comigo: nós pedimos para que a string na variável `s` fosse exibida *se o padrão da regex fosse encontrado*, e ele foi encontrado nas strings `ab` e `abc`! Isso nos leva à terceira regra...

> ***Regra #3:*** *tudo tem limites!*

Se o padrão buscado precisa “casar” com toda a string, nós temos que especificar os limites. Às vezes, é possível utilizar caracteres literais, como no exemplo abaixo:

```
:~$ var='a ab abc'
:~$ for s in $var; do [[ $s =~ a.c ]] && echo $s; done
abc
```

Mas, se existisse uma string `abcd` em `var`, nosso código falharia em exibir apenas `abc` (se este fosse o objetivo):

```
:~$ var='a ab abc abcd'
:~$ for s in $var; do [[ $s =~ a.c ]] && echo $s; done
abc
abcd
```

É para isso que existem dois caracteres especiais que servem como *âncoras*: `^` (início) e `$` (fim). Originalmente, eles servem para indicar o início e o fim de uma linha de texto. No comando `[[ ]]`, porém, eles representam o início e o fim da string que está sendo testada, e isso faz toda a diferença:

```
:~$ var='a ab abc abcd'
:~$ for s in $var; do [[ $s =~ ^.$ ]] && echo $s; done
a
:~$ for s in $var; do [[ $s =~ .b$ ]] && echo $s; done
ab
:~$ for s in $var; do [[ $s =~ a.c$ ]] && echo $s; done
abc
```

Repare que apenas na primeira regex, onde procuramos uma string formada por apenas um caractere, foi preciso especificar o começo e o fim da string. Nas demais, bastou indicar o fim da string.

Obviamente, a formação de uma expressão regular vai depender da variedade do conjunto de strings e do que realmente estamos buscando. Seja como for, é sempre bom ter em mente a quarta regra:

> ***Regra #4:*** *seja específico.*

Antes de tentarmos elaborar uma regex para o nosso programa, vamos conhecer mais alguns caracteres especiais utilizados nas expressões regulares (também chamados de *metacaracteres*):

Metacaractere	Significado
`.`	Representante. Expressa um caractere.
`[…]`	Representante. Lista de caracteres válidos.
`*`	Quantificador. Zero ou mais caracteres.
`?`	Quantificador. Zero ou um caractere.
`+`	Quantificador. Um ou mais caracteres.
`{n,m}`	Quantificador. No mínimo 'n' e no máximo 'm' caracteres.
`^`	Âncora. Início da linha ou da string.
`$`	Âncora. Fim da linha ou da string.
`(…)`	Agrupamento. Especifica um trecho do padrão para efeito de quantificação, para indicar alternativas, ou para permitir acesso posterior.
`|`	Ou. Indica padrões alternativos igualmente válidos.

> ***Importante!*** *As listas de caracteres também podem ser negadas com um* `^` *e os metacaracteres também podem ser escapados com* `\` *a fim de que sejam removidos seus significados especiais.*

Com tudo isso em mente, vamos rever os requisitos dos nomes válidos para os projetos criados com o nosso programa. Nós sabemos que eles só poderão conter:

- Letras minúsculas sem acentos ou cedilha;
- Números;

- Os caracteres traço (-) e sublinhado (_).

Todos os outros caracteres são proibidos.

A parte das letras minúsculas e números é fácil – nós já sabemos representar listas de faixas de caracteres. Como são apenas as letras minúsculas da tabela ASCII e os dígitos, isso atende perfeitamente o que estamos buscando: `[a-z0-9]`. Também podemos incluir nessa lista de caracteres válidos o traço e o sublinhado: `[-_a-z0-9]`.

> ***Detalhe:*** *o traço poderia ser confundido com a definição de uma faixa de caracteres, por isso ele foi colocado no começo da lista.*

Mas isso só representa um caractere válido, e nós queremos que o nome do projeto seja uma string composta por *um ou mais caracteres* que atendam esse padrão. Portanto, o quantificador adequando para o nosso caso é o `+`: `[-_a-z0-9]+`. Por último, para sermos específicos, resta dizermos que o padrão que temos até aqui precisa ser casado do início ao fim da string, portanto, a regex `^[-_a-z0-9]+$` deve resolver o nosso problema.

Vamos testar?

Exemplo 2.46 – Validando strings com uma expressão regular.

```
:~$ var='teste teste_1 teste-2 teste.3 testão'
:~$ regex='^[-_a-z0-9]+$'
:~$ for s in $var; do [[ $s =~ $regex ]] && echo $s; done
teste
teste_1
teste-2
testão
```

Pois é, temos um problema...

Como dissemos, as faixas de caracteres nas expressões regulares não são afetadas pela opção `globasciiranges` e, portanto, obedecem às tabelas de caracteres utilizadas nas configurações de localidade do nosso sistema.

2.3.13 – Sofrendo com as variáveis LANG e LC_*

Nos sistemas GNU/Linux, é possível ver quais são as configurações regionais em uso com o utilitário `locale`. Aqui no meu Debian, por exemplo, estas seriam as minhas configurações:

```
:~$ locale
LANG=pt_BR.UTF-8
LANGUAGE=pt_BR:pt:en
LC_CTYPE="pt_BR.UTF-8"
LC_NUMERIC="pt_BR.UTF-8"
LC_TIME="pt_BR.UTF-8"
LC_COLLATE="pt_BR.UTF-8"
LC_MONETARY="pt_BR.UTF-8"
LC_MESSAGES="pt_BR.UTF-8"
LC_PAPER="pt_BR.UTF-8"
LC_NAME="pt_BR.UTF-8"
LC_ADDRESS="pt_BR.UTF-8"
LC_TELEPHONE="pt_BR.UTF-8"
LC_MEASUREMENT="pt_BR.UTF-8"
LC_IDENTIFICATION="pt_BR.UTF-8"
LC_ALL=
```

Como podemos ver, alguns valores são apresentados sem aspas e outros estão entre aspas. Os valores *sem aspas* são aqueles que já estão definidos no sistema, ao contrário dos que estão *entre aspas*, que foram simplesmente deduzidos pelo `locale` para indicar quais são as regras de localização regional atualmente aplicáveis para cada uma dessas variáveis.

Todas essas variáveis de ambiente afetam a exibição de textos e, para o nosso incômodo, o casamento de expressões regulares. Mas quatro delas são especialmente importantes para nós:

Variável	Descrição
`LANG`	Usada para determinar a categoria da localidade para qualquer categoria não especificada com uma das variáveis iniciadas com `LC_`.
`LC_ALL`	Sobrescreve o valor de `LANG` e qualquer outra variável `LC_` que especifique uma categoria de localidade.
`LC_COLLATE`	Determina a ordem de agrupamento utilizada quando da classificação dos resultados de uma expansão de nomes de arquivos e determina o comportamento de expressões de faixas, classes de equivalência e sequências de agrupamento numa expansão de nomes de arquivos e casamentos de padrões.
`LC_CTYPE`	Determina a interpretação de caracteres e o comportamento de classes de caracteres em expansões de nomes de arquivos e casamentos de padrões.

Às vezes, para a correta execução de scripts e comandos, é preciso fazer um ajuste nos valores dessas variáveis, o que pode ser feito sem muita dificuldade e sem efeitos colaterais, já que são mudanças que afetam apenas a sessão do shell em que forem feitas. Também não é necessário alterar todas as variáveis. Como podemos ver na tabela, a variável `LC_ALL` se sobrepõe às definições feitas nas demais. Sendo assim, basta alterá-la para termos o efeito que desejamos – resta saber como.

Enquanto a *categoria de localidade* `pt_BR.UTF-8`, por exemplo, traz definições específicas para a codificação dos caracteres do nosso idioma, existe uma

localidade especial bem mais simples e portável, a categoria de localidade C. Nela, cada caractere é formado com apenas um byte, geralmente seguindo o padrão da tabela ASCII, e a sua ordem de classificação é baseada no valor numérico de cada caractere.

Para entender melhor como as variáveis `LC_*` e a localidade C afetam a ordem de classificação, o agrupamento e o casamento de padrões, primeiro teremos que desligar a opção `globasciiranges` com o comando `shopt -u`:

Exemplo 2.47 – Desligando 'globasciiranges'.

```
:~$ shopt -u globasciiranges
```

Agora vamos utilizar o comando `[[ ]]` novamente para comparar strings com padrões:

```
:~$ var='a  A  b  B  c  C  d  D  e  E'
:~$ for s in $var; do [[ $s == [a-z] ]] && echo $c; done
a
A
b
B
c
C
d
D
e
E
```

Notou a diferença? Antes, a faixa `[a-z]` seguia a tabela ASCII e só teria casado com os caracteres minúsculos da string em `s`. Agora, ele segue o agrupamento definido na variável `LC_COLLATE`, que é o mesmo utilizado para a ordenação alfabética de diversas listagens no shell.

Para mudar este comportamento, vamos alterar o valor de LC_ALL para categoria de localidade C e tentar novamente:

```
:~$ LC_ALL=C
:~$ var='a  A  b  B  c  C  d  D  e  E'
:~$ for s in $var; do [[ $s == [a-z] ]] && echo $c; done
a
b
c
d
e
```

Ótimo, tivemos o mesmo efeito da opção `globasciiranges` ligada!

Mas o nosso problema não era as comparações de strings com padrões de nomes de arquivos, mas as comparações com expressões regulares.

Então, vamos testar novamente a nossa regex:

```
:~$ var='teste teste_1 teste-2 teste.3 testão'
:~$ regex='^[-_a-z0-9]+$'
:~$ for s in $var; do [[ $s =~ $regex ]] && echo $s; done
teste
teste_1
teste-2
```

Sucesso! Agora o teste se comporta como esperávamos!

Agora só falta restaurar os padrões do sistema e escrever a segunda validação do nosso programa.

Para restaurar `globasciiranges`, nós utilizamos o comando `shopt` com a opção `-s`:

Exemplo 2.48 – Religando 'globasciiranges'.

```
:~$ shopt -s globasciiranges
```

A variável `LC_ALL`, por sua vez, pode ser restaurada simplesmente com a atribuição de um valor nulo, o que faz com que ela assuma o valor padrão definido na variável `LANG`.

Exemplo 2.49 – Restaurando o valor padrão de 'LC_ALL'.

```
LC_ALL=
```

> *No nosso script, a restauração do valor de* `LC_ALL` *é opcional, visto que ela dura apenas enquanto durar a sessão do shell iniciada pelo script e não afetará o restante do sistema.*

Agora sim, vamos ao código do nosso programa:

```
#!/usr/bin/env bash

# Valida o número de parâmetros...
if [[ $# -ne 1 ]]; then
     echo 'Número incorreto de parâmetros!'
     exit 1
fi

# Define LC_ALL para a categoria de localidade C...
LC_ALL=C

#Valida o nome do novo projeto...
if [[ ! $1 =~ ^[-_0-9a-z]+$ ]]; then
     echo 'Nome inválido!'
    exit 2
fi
```

Até aqui, nós já vimos...

- Como capturar o nome do projeto passado como parâmetro de execução do nosso programa;
- Como saber se o utilizador informou um nome para o novo projeto;
- Como validar o nome do novo projeto.

Ainda nos falta saber...

- Como verificar se a pasta do novo projeto já existe;
- Como criar a pasta do projeto;
- Como obter as informações automáticas do modelo;
- Como criar o arquivo do script já com o conteúdo do modelo;
- Como abrir o editor após a criação do arquivo...

Então, respire fundo e mão na massa!

2.3.14 – Testando a existência de pastas e arquivos

Uma das coisas mais comuns do shell é a variedade de opções para a solução dos mesmos problemas. Por exemplo, se queremos apenas confirmar visualmente a existência de uma pasta ou de um arquivo, o utilitário `ls` pode ser suficiente – se existir, o arquivo (ou a pasta) será listado, caso contrário, nós veremos uma mensagem de erro. No *tópico 1.9.6*, nós vimos que há formas de ocultar este tipo de informação enviando as saídas geradas por um comando para o *limbo* (o *dispositivo nulo*, `/dev/null`). Deste modo, até o utilitário `ls` pode ser utilizado na verificação da existência de arquivos nos nossos scripts, por exemplo:

Exemplo 2.50 – Testando a existência de arquivos com o 'ls'.

```
if ls /caminho/arquivo &> /dev/null; then
    echo 'existe'
else
    echo 'não existe'
fi
```

Contudo, existem recursos no shell especificamente desenvolvidos para a verificação da existência de arquivos, são os operadores de nomes de arquivos dos comandos `test`, `[ ]` e `[[ ]]`. Nós já vimos dois desses operadores no *tópico 2.2.10*: os operadores `-f` e `-L`.

No caso do nosso programa, porém, o que precisamos testar é a existência de um diretório onde um *novo projeto* será criado. Já sabendo que o comando `help test` nos dá uma lista dos operadores disponíveis, nosso interesse seria automaticamente atraído para um deles em especial: o operador `-d`, que faz a expressão testada ser avaliada como verdadeira se o arquivo for um diretório.

Exemplo 2.51 – Testando a existência de diretórios com o operador '-d'.

```
if [[ -d nome ]]; then
    echo 'nome existe e é um diretório'
else
    echo 'nome não existe ou não é um diretório'
fi
```

Mas, será que é o que nós queremos?

Pense no seguinte cenário: na pasta testada existe um arquivo comum chamado `nome`. Nós não sabemos disso, mas queremos descobrir se uma nova pasta, também chamada `nome`, pode ser criada no mesmo diretório.

Então, vamos experimentar o operador `-d` novamente:

```
:~$ [[ -d nome ]]; echo $?
1
```

O estado de saída do teste com o operador `-d` foi *erro* (`1`), mas isso significa apenas que `nome` não é um diretório, não que o arquivo não existe. Sendo assim, vamos tentar com o já conhecido operador `-f`:

```
:~$ [[ -f nome ]]; echo $?
0
```

Sucesso! No fim das contas, o arquivo `nome` realmente existia, só não era um diretório.

> *Este é mais um exemplo de que devemos estar sempre atentos quanto ao que "é lógico" e o que "só parece lógico".*

Portanto, nós precisamos de uma expressão que seja avaliada como *verdadeira* para a existência de qualquer tipo de arquivo chamado `nome` (diretório ou arquivo). Se realmente precisarmos saber se o arquivo, caso ele exista, é um diretório, aí sim o operador `-d` seria bem-vindo.

Para o que queremos, existe um outro operador bem adequado: o operador `-a`, que testa apenas a existência do arquivo, independente de seu tipo. Observe o que aconteceria numa pasta hipotética onde coexistissem o arquivo comum `nome-arq` e o diretório `nome-dir`:

Exemplo 2.52 – Testando a existência de arquivos com o operador '-a'.

```
:~$ [[ -a nome-arq ]]; echo $?
0
:~$ [[ -a nome-dir ]]; echo $?
0
```

Isso é perfeito, porque esse é o teste que realmente equivale àquele que fizemos com o utilitário `ls`. No final, o que queremos saber é se poderemos ou não criar a pasta do novo projeto e, se for o caso, avisar ao utilizador que já existe uma pasta ou um arquivo com o nome que ele escolheu.

Mas, observe como o nosso código está ficando:

```
#!/usr/bin/env bash

# Valida o número de parâmetros...
if [[ $# -ne 1 ]]; then
     echo 'Número incorreto de parâmetros!'
     exit 1
fi
```

```
# Define LC_ALL para a categoria de localidade C...
LC_ALL=C

#Valida o nome do novo projeto...
if [[ ! $1 =~ ^[-_0-9a-z]+$ ]]; then
     echo 'Nome inválido!'
     exit 2
fi

# Verifica se existe uma pasta com o nome do projeto...
if [[ -a $1 ]]; then
     echo "Já existe uma pasta chamada $1!"
     exit 3
fi
```

Apesar de funcional, existe muita coisa desnecessária aqui. Então, antes de prosseguirmos, vamos tentar organizar melhor as coisas e aproveitar para puxar mais assuntos novos.

O primeiro problema que podemos notar é que todos os testes, em caso de erro, sempre causam a exibição de uma mensagem de erro e o encerramento do script. As mensagens e os códigos dos erros são diferentes, mas é sempre a mesma rotina:

```
echo mensagem
exit número
```

O que nos dá a oportunidade de falarmos sobre as *funções* no Bash!

2.3.15 – Reaproveitando código com funções

No Bash, as funções seguem um conceito ligeiramente diferente em relação às outras linguagens. O princípio básico ainda é o mesmo: um agrupamento de instruções identificado por um nome pelo qual pode ser chamado. Mas, como

estamos trabalhando com comandos, esses agrupamentos também são comandos, estando ou não associados a um identificador – são os chamados *comandos compostos*.

Mais adiante, no *capítulo 7*, nós veremos os comandos compostos mais a fundo, mas você já deve ter me ouvido (ou melhor, *lido*) utilizar essa expressão várias vezes até aqui. O bloco `if` o *loop* `for`, os comandos `[[ ]]` e `(( ))`, todos eles são comandos compostos e, por definição, eles poderiam ser nomeados para que se tornassem funções! No momento, porém, vamos nos concentrar no comando composto mais utilizado na criação de funções, que é o *agrupamento com chaves*.

O agrupamento com chaves faz com que uma lista de comandos seja executada como uma unidade. Isolada de um contexto, pode não fazer muito sentido – afinal, na linha de comandos ou em scripts, não haveria diferenças perceptíveis na interpretação dos comandos listados. Mas, o fato é que as chaves são *palavras reservadas* que sinalizam para o shell o início e o fim de um grupo de comandos, o que torna obrigatórios os espaços entre elas e os comandos, e o uso do ponto e vírgula (`;`) ou de uma quebra de linha após o último comando da lista:

```
# Agrupamento em uma linha...
{ comando 1; comando 2; … comando n; }

# Agrupamento em múltiplas linhas...
{
comando 1
comando 2
…
comando n
}
```

Para transformar um agrupamento com chaves em uma função, basta dar a ele um nome seguido de parêntesis:

```
# Função em uma linha...
nome( ) { comandos; }

# Função em múltiplas linhas...
nome( ) {
      comandos
}
```

Olhando assim, não parece muito diferente das funções em outras linguagens, mas há quatro pontos bem peculiares do Bash a serem observados:

- As funções não retornam nada;
- Não existe uma declaração de parâmetros;
- Os argumentos são passados por parâmetros posicionais na chamada da função;
- Por padrão, todas as variáveis em uma função são globais.

O retorno de uma função é sempre um inteiro entre 0 e 255 definido pelo estado de saída de seu último comando. Nós podemos controlar isso com o comando interno `return`, que encerra a função, devolve o controle da execução para o fluxo principal do código a partir do ponto onde ela foi chamada, e força um valor para o estado de saída – de todo modo, este valor também só pode ser um inteiro entre 0 e 255.

Por exemplo:

Exemplo 2.53 – Estados de saída de funções.

```
:~$ relaxa( ) { echo 'relaxa'; false; }
:~$ relaxa; echo $?
relaxa
1
:~$ tranquilo( ) { echo 'tranquilo'; false; return 0; }
:~$ tranquilo; echo $?
```

```
tranquilo
0
```

Sem a indicação de um valor, o comando `return` causará o fim da execução da função com o estado de saída do último comando executado antes dele.

Quanto à passagem de argumentos para as funções, nós podemos imaginar que elas são como scripts dentro de nossos scripts, ou seja, ao chamar uma função, nós podemos passar valores para ela da mesma forma que fazemos quando passamos valores para os nossos scripts na linha de comandos. Por baixo do capô, o Bash substituirá temporariamente os valores nas variáveis especiais `1` a `n` pelos argumentos passados na chamada da função, e nós poderemos utilizá-las nos comandos das nossas funções.

Exemplo 2.54 – Passando argumentos para funções.

```
:~$ relaxa() { echo "Relaxa, $1!"; }
:~$ relaxa amigo
Relaxa, amigo!
:~$ relaxa Pedrinho
Relaxa, Pedrinho!
```

Finalmente, sobre o escopo das variáveis numa função, elas são sempre *globais*, ou seja, poderão ser acessadas de qualquer outra parte do código ou da sessão do shell em execução no terminal, a menos que se faça uma definição contrária com o comando interno `local`:

Exemplo 2.55 – Escopo de variáveis em funções.

```
:~$ soma() { a=3; local b=2; echo $((a + b)); }
:~$ soma
5
:~$ echo $a
3
```

```
:~$ echo $b

:~$
```

> ***Importante!*** *Os comandos internos* `return` *e* `local` *só funcionam no contexto de uma função!*

No *capítulo 8* nós abordaremos com mais profundidade as funções, mas o que vimos já é suficiente para resolvermos o primeiro problema do nosso código até aqui: em vez de repetirmos os comandos `echo` e `exit` em cada validação que precisarmos fazer, nós podemos criar uma função e chamá-la quando for o caso.

> ***Importante!*** *A função sempre deve ser definida em um ponto do código anterior à sua chamada!*

Na nova amostra do nosso script, nós incluímos a função `erro` e fizemos todas as mudanças necessárias para utilizá-la.

Veja como ficou:

```
#!/usr/bin/env bash

# Define LC_ALL para a categoria de localidade C...
LC_ALL=C

# Função para encerrar o script com uma mensagem de erro...
erro() {
     echo $2
     exit $1
}

# Valida o número de parâmetros...
```

```
[[ $# -ne 1 ]] && erro 1 'Número incorreto de parâmetros!'

#Valida o nome do novo projeto...
[[ $1 =~ ^[-_0-9a-z]+$ ]] || erro 2 'Nome inválido!'

# Verifica se existe uma pasta com o nome do projeto...
[[ -a $1 ]] && erro 3 "Já existe uma pasta chamada $1!"
```

Bem melhor assim! Mas, você notou as outras mudanças?

Isso mesmo! Em vez do comando composto `if`, nós estamos utilizando os operadores de controle condicional `&&` e `||`.

Normalmente, quando programamos em Bash, nós deixamos o `if` para a execução de blocos contendo mais de um comando, sempre preferindo os operadores de controle condicional para situações mais simples, desde que isso não afete a clareza da visualização da lógica envolvida. Com a mudança, os nossos testes levarão a execução de apenas um comando: a chamada da função `erro`, o que não justifica o uso de uma estrutura complexa como o `if`.

Outra vantagem dessa abordagem é que nós podemos dispensar a negação (`!`) da avaliação da expressão regular.

```
# Antes: sai com 'sucesso' se for inválido...
[[ ! $1 =~ ^[-_0-9a-z]+$ ]]

# Depois: sai com 'erro' se for inválido...
[[ $1 =~ ^[-_0-9a-z]+$ ]]
```

Ela só estava lá para que o comando `[[ ]]` saísse com estado de *sucesso* caso o nome dado ao novo projeto fosse *inválido*. Com o uso de operadores de controle condicional, a negação se torna totalmente desnecessária, porque é possível condicionar a execução de comandos da forma que quisermos – se queremos um comando executado em caso de erro, basta utilizar o operador `||`.

Mas, a principal novidade é a função `erro` – então, vamos analisá-la!

```
erro() { echo $2; exit $1; }
```

Observe que os comandos echo e exit esperam receber os valores nas variáveis especiais `1` e `2`. Como vimos, são as mesmas variáveis criadas e manipuladas pelo próprio Bash no início da sessão do script. Elas podem ou não conter dados passados na linha de comandos quando o script foi iniciado. Se contiverem, esses dados serão temporariamente substituídos na chamada da função e serão restaurados ao final da sua execução, o que fica muito claro no exemplo abaixo.

Script `teste.sh`:

```
#!/usr/bin/env bash

# Função 'oi'...
oi() { echo "Oi, eu sou $1!"; }

# Expandindo a variável '1' antes da chamada da função...
echo "Eu sou $1!"

# Chamada da função 'oi'...
oi Goku

# Expandindo a variável '1' depois da chamada da função...
echo "E daí? Eu continuo sendo $1!"
```

Executando o script 'teste.sh':

```
:~$ teste.sh Vegeta
Eu sou Vegeta!
Oi, eu sou Goku!
E daí? Eu continuo sendo Vegeta!
```

No nosso código, os valores dos parâmetros posicionais são passados na chamada da função `erro`:

```
erro NÚMERO MENSAGEM
        ↑       ↑
        1       2
```

Mas, nós podemos fazer ainda melhor com somente um argumento, e só temos que entender como funcionam os vetores para isso.

2.3.16 – Armazenando mensagens em vetores indexados

Diferente das variáveis que apontam apenas para um valor (*variáveis escalares*), os vetores apontam para um grupo de valores. Cada um desses valores, também chamados de *elementos*, é identificado no vetor por um índice que pode ser um número inteiro (*vetor indexado*) ou uma string (*vetor associativo*). Nós entraremos neste e em outros detalhes no *capítulo 3*. Por enquanto, basta descobrirmos uma forma de criar e expandir os valores nos elementos de um *vetor indexado*.

No caso do nosso programa, são três mensagens de erro, cada uma associada a um número inteiro correspondente a um estado de saída:

```
1 → 'Número incorreto de parâmetros!'
2 → 'Nome inválido!'
3 → "Já existe uma pasta chamada $1!"
```

Uma das diferenças entre as variáveis normais e os vetores, é que os elementos dos vetores são identificados pelo nome do vetor seguido por um subscrito contendo o índice do elemento:

```
vetor → nome do vetor
n → índice do elemento
vetor[n] → nome e subscrito de um elemento
```

Então, na criação de um vetor, basta atribuirmos um valor ao nome do elemento de índice *n*, e é exatamente isso que faremos com as nossas mensagens de erro:

Exemplo 2.56 – Criando um vetor indexado com mensagens de erro.

```
msg_erro[1]='Número incorreto de parâmetros!'
msg_erro[2]='Nome inválido!'
msg_erro[3]="Já existe uma pasta chamada $1!"
```

Para acessar esses valores, basta utilizar a sintaxe completa para a expansão de variáveis:

Exemplo 2.57 – Expandindo o elemento de um vetor.

```
:~$ echo ${msg_erro[1]}
Número incorreto de parâmetros!
```

Com isso, estamos prontos para aplicar mais uma melhoria ao código do nosso programa!

Ou será que não?

Veja com atenção a mensagem em `msg_erro[3]`:

```
msg_erro[3]="Já existe uma pasta chamada $1!"
```

Originalmente, nós pensamos na expansão do primeiro parâmetro (`$1`) passado para o script. Mas, como essa expansão se comportaria na função `erro`, que agora ficaria assim:

```
erro() {
     echo ${msg_erro[$1]}
     exit $1
}
```

Que bom que você estava atento, porque não haverá problema algum!

O fato é que a expansão ocorrerá *antes* do comando de atribuição ser executado. Portanto, a string armazenada em `msg_erro[3]` já terá o nome do projeto passado pelo usuário, e é apenas isso que o comando `echo` verá quando for executado.

Contudo, há uma coisa a mais que podemos fazer na nossa função:

```
erro() {
 echo ${msg_erro[$1]}
 echo 'Uso: rb novo-projeto'
 exit $1
}
```

Assim, além da mensagem de erro, nós também exibimos uma ajuda.

Agora sim, mais tranquilos, vejamos como está o nosso programa:

```
#!/usr/bin/env bash

# Define LC_ALL para a categoria de localidade C...
LC_ALL=C

# Mensagens de erro...
msg_erro[1]='Número incorreto de parâmetros!'
msg_erro[2]='Nome inválido!'
msg_erro[3]="Já existe uma pasta chamada $1!"

# Função para encerrar o script com uma mensagem de erro...
erro() {
```

```
    echo ${msg_erro[$1]}
    echo 'Uso: rb novo-projeto'
    exit $1
}

# Valida o número de parâmetros...
[[ $# -ne 1 ]] && erro 1

#Valida o nome do novo projeto...
[[ $1 =~ ^[-_0-9a-z]+$ ]] || erro 2

# Verifica se existe uma pasta com o nome do projeto...
[[ -a $1 ]] && erro 3
```

Do ponto em que estamos, só falta descobrir:

- Como obter as informações automáticas do modelo;
- Como criar o arquivo do script já com o conteúdo do modelo;
- Como abrir o editor após a criação do arquivo...

Então, vamos tomar um bom café enquanto observamos quais seriam as informações automáticas de que precisamos:

```
#!/usr/bin/env bash
# ------------------------------------------------
# Projeto  : (automático)
# Arquivo  : (automático)
# Descrição:
# Versão   : 0.0.0
# Data     : (automático)
# Autor    : Blau Araujo <blau@debxp.org>
# Licença  : GNU/GPL v3.0
# ------------------------------------------------
# Uso:
# ------------------------------------------------
```

Ora, o nome do projeto e o arquivo do script nós já temos – ambos são obtidos a partir do primeiro parâmetro posicional fornecido na linha de comandos pelo utilizador! Para formalizar nossa conquista, vamos atualizar essas duas linhas na string que precisamos criar para o nosso modelo:

```
# Projeto  : $1
# Arquivo  : $1.sh
```

Resta saber como podemos obter a data e a hora da criação do novo script. Na linha de comandos, bastaria executar o utilitário `date`:

Exemplo 2.58 – Obtendo a data e a hora com o utilitário 'date'.

```
:~$ date
ter nov  3 16:32:16 -03 2020
```

Com o sinal de adição (`+`) seguido de uma sequência caracteres, nós podemos controlar o formato de apresentação da data e da hora. Por exemplo, com a sequência `%R`, nós temos a hora no formato `HH:MM`...

Exemplo 2.59 – Hora resumida com o utilitário 'date'.

```
:~$ date +%R
16:34
```

Como a sequência de formatação pode conter espaços, o que poderia ser interpretado como outros argumentos, nós delimitamos a string do formato com aspas...

Exemplo 2.60 – Exibindo data e hora formatadas com o utilitário 'date'.

```
:~$ date '+%d/%m/%Y - %R'
03/11/2020 - 16:36
```

Aliás, o formato do exemplo parece perfeito para o nosso script! Nós só temos que descobrir uma forma de inseri-lo no modelo – e é neste ponto que entram as *substituições de comandos*.

2.3.17 – Substituição de comandos

Do mesmo modo que as expansões de variáveis (`${nome}`) e de expressões aritméticas (`$((expressão))`) são trocadas pelos valores que elas representam, as substituições de comandos são expandidas para a saída de uma linha de comando.

Utilizando o utilitário `date` como exemplo, nós podemos concatenar sua saída com uma string desta forma:

Exemplo 2.61 – Expandindo a saída de um comando.

```
:~$ echo Agora são $(date '+%R')
Agora são 16:42
```

Portanto, esta parte da string do nosso modelo pode ser feita assim:

```
# Data     : $(date '+%d/%m/%Y - %R')
```

> ***Atenção!*** *A substituição de comandos é executada em um subshell. Nós falaremos mais sobre isso no próximo capítulo.*

No código do programa a string do modelo precisa estar armazenada numa variável para que possa ser expandida, mas ela contém várias linhas, o que faria o shell interpretá-la como vários comandos. Então, nós temos que utilizar *aspas* para delimitar o início e o fim da string da mesma forma que delimitamos argumentos quando eles têm espaços.

```
modelo="#!/usr/bin/env bash
# ------------------------------------------------
# Projeto  : $1
# Arquivo  : $1.sh
# Descrição:
# Versão   : 0.0.0
# Data     : $(date '+%d/%m/%Y - %R')
# Autor    : Blau Araujo <blau@debxp.org>
# Licença  : GNU/GPL v3.0
# ------------------------------------------------
# Uso:
# ------------------------------------------------
"
```

2.3.18 – Com aspas duplas, simples ou sem aspas?

Com as aspas, todos os espaços em branco são tratados literalmente, e não como um separador de palavras, o que inclui qualquer quantidade de *espaços*, *tabulações* e *quebras de linha*. O mais importante, porém, é saber avaliar qual tipo de aspas devemos utilizar. Na definição da string do nosso modelo, por exemplo, nós utilizamos aspas duplas (`"`), mas trabalhamos com aspas simples (`'`) quando delimitamos a string de formatação passada como argumento do utilitário `date`. Isso não foi uma decisão arbitrária, e o exemplo abaixo pode nos ajudar a entender melhor como as aspas funcionam.

Observe com atenção:

Exemplo 2.62 – Como o shell interpreta as aspas (ou sua ausência).

```
:~$ fruta=banana

:~$ echo A $fruta é uma fruta:    nutritiva
A banana é uma fruta: nutritiva

:~$ echo "A $fruta é uma fruta:    nutritiva"
A banana é uma fruta:    nutritiva
```

```
:~$ echo 'A $fruta é uma fruta:    nutritiva'
A $fruta é uma fruta:    nutritiva
```

Repare que nós temos uma string que contém a expansão de uma variável e vários espaços em branco. Sem as aspas, a expansão é executada, mas todos os espaços consecutivos são colapsados para apenas um. Com aspas, sejam simples ou duplas, os espaços são preservados, mas só as *aspas duplas* permitem que ocorra a expansão da variável.

Uma regrinha que eu adotei para facilitar a minha vida, foi sempre observar esses dois critérios: os *espaços em branco* e as *expansões*. Além disso, por ordem de prioridade, eu sempre vejo se posso deixar a linha de comando *sem aspas*, a menos que existam espaços que precisem ser preservados. Neste caso, eu priorizo as *aspas simples*, a menos que eu queira fazer a expansão de alguma variável – que é quando eu recorro às *aspas duplas*.

> ***Por que isso importa?*** *Muitas pessoas argumentam que existe um ganho no processamento quando evitamos utilizar aspas duplas sem necessidade, pois as aspas simples poupam o shell do trabalho de buscar símbolos que indiquem a necessidade de realizar alguma expansão. A verdade, porém, é que esse ganho é quase sempre desprezível, mas pode se tornar considerável se a linha do comando tiver que ser processada milhares de vezes.*

2.3.19 – Finalizando o seu script

Neste ponto, nós já temos quase tudo que precisamos para finalizar o código do nosso programa – e eu prometo que os detalhes que faltam serão bem mais fáceis de resolver.

Primeiro, tendo a string do modelo numa variável, nós podemos fazer um redirecionamento de seu conteúdo para o arquivo do novo script, que será o

nome do projeto acrescido da "extensão" `.sh`. Mas, além disso, nós precisamos definir em que pasta o arquivo será criado.

A solução que eu adotei, para meu uso pessoal, foi padronizar um local de destino para os novos projetos: o diretório `~/projetos`. Então, se quiser me acompanhar nessa decisão, você pode criar uma variável para armazenar o nome da pasta de destino desta forma:

```
projetos=$HOME/projetos
```

> ***Você decide!*** *Se quiser trabalhar com outra pasta de destino, basta alterar na linha acima de acordo com a sua preferência.*

O mais importante, porém, é que você entenda por que eu utilizo a variável de ambiente `HOME`. O problema é que a expansão do til (`~`) nem sempre é confiável em scripts – acontece muito dele ser interpretado literalmente em alguns contextos. Isso seria previsível na maior parte das vezes, mas é uma alternativa de trabalho simples que me permite dedicar atenção a detalhes mais relevantes.

> *A variável `HOME` sempre será expandida para o caminho completo do diretório pessoal do usuário logado no shell em execução.*

Com mais isso definido, nós já podemos criar a pasta do novo projeto:

```
projetos=$HOME/projetos

mkdir -p $projetos/$1
```

> *A opção `-p` do utilitário `mkdir` tem duas finalidades: criar todas as pastas faltantes até a última pasta do caminho informado e evitar que o comando retorne um erro no caso da pasta que estamos tentando criar já existir.*

Com a pasta criada, vamos criar o arquivo:

```
echo $modelo >> $projetos/$1/$1.sh
```

Para que o novo script tenha permissão de execução, nós utilizamos o já conhecido utilitário `chmod`:

```
chmod +x $projetos/$1/$1.sh
```

Finalmente, para que ele possa ser executado de qualquer lugar, nós precisamos criar uma ligação simbólica apontando para ele no diretório `~/bin` (ou na sua pasta pessoal para arquivos executáveis):

```
ln -s $projetos/$1/$1.sh $HOME/bin/$1
```

Muito bom, mas existem alguns probleminhas que você mesmo deve ter percebido. Vamos olhar como ficou essa parte do código:

```
projetos=$HOME/projetos

mkdir -p $projetos/$1
echo $modelo >> $projetos/$1/$1.sh
chmod +x $projetos/$1/$1.sh
ln -s $projetos/$1/$1.sh $HOME/bin/$1
```

Veja quantas vezes nós repetimos a expansão `$projetos/$1`!

Embora isso represente um custo mínimo, se pensarmos apenas em termos de processamento, a legibilidade do código passou longe daqui e mandou lembranças!

Então, vamos tentar outra forma de organizar o nosso código:

```
caminho_destino=$HOME/projetos
caminho_projeto=$caminho_destino/$1
arquivo_destino=$caminho_projeto/$1.sh
arquivo_symlink=$HOME/bin/$1

mkdir -p $caminho_projeto
echo $modelo >> $arquivo_destino
chmod +x $arquivo_destino
ln -s $arquivo_destino $arquivo_symlink
```

Mesmo sendo adepto da tese de que quanto mais comentários melhor, eu tenho que admitir que um código assim quase nem precisa ser comentado – tudo é claro e autoexplicativo.

> *Nós poderíamos criar aqui os testes para verificar se todos os comandos dessa última etapa foram bem-sucedidos, mas vamos deixar essa parte por sua conta – é um bom exercício!*

Por último, basta incluir o comando para a execução do editor da sua preferência e o código estará pronto! Para meu uso, eu prefiro o editor Nano, mas você pode adaptar o código para abrir o novo script com o editor da sua preferência. Veja como fica:

```
nano $arquivo_destino
```

Com isso, nosso código está finalmente completo!

Agora, cabe a você adaptá-lo ao seu jeito de trabalhar. Note que ele não serve apenas para criar scripts em Bash. Com algumas mudanças, ele poderá criar modelos de projetos para o desenvolvimento em qualquer outra linguagem. Mas, acima de tudo, este pequeno projeto nos ajudou a conhecer e aplicar os elementos mais essenciais de qualquer desenvolvimento em Bash.

Nos próximos capítulos, nós vamos rever alguns desses conceitos e conhecer coisas novas, mas você já deve estar mais preparado para encarar as peculiaridades do trabalho com o shell.

Antes, porém, veja como ficou o seu primeiro programa em Bash...

Bom trabalho!

Exemplo 2.63 – O código do nosso primeiro programa em Bash (nb.sh)

```
#!/usr/bin/env bash

# PROCEDIMENTOS DE VALIDAÇÃO --------------------------

# Define LC_ALL para a categoria de localidade C...
LC_ALL=C

# Mensagens de erro...
msg_erro[1]='Número incorreto de parâmetros!'
msg_erro[2]='Nome inválido!'
msg_erro[3]="Já existe uma pasta chamada $1!"

# Função para encerrar o script com uma mensagem de erro...
erro( ) {
     echo ${msg_erro[$1]}
     echo 'Uso: rb novo-projeto'
     exit $1
}

# Valida o número de parâmetros...
[[ $# -ne 1 ]] && erro 1
```

```
#Valida o nome do novo projeto...
[[ $1 =~ ^[-_0-9a-z]+$ ]] || erro 2

# Verifica se existe uma pasta com o nome do projeto...
[[ -a $1 ]] && erro 3

# PROCEDIMENTOS DE CRIAÇÃO DO NOVO PROJETO ------------

# String do modelo a ser inserido no novo script...
modelo="#!/usr/bin/env bash
# ------------------------------------------------
# Projeto  : $1
# Arquivo  : $1.sh
# Descrição:
# Versão   : 0.0.0
# Data     : $(date '+%d/%m/%Y - %R')
# Autor    : Blau Araujo <blau@debxp.org>
# Licença  : GNU/GPL v3.0
# ------------------------------------------------
# Uso:
# ------------------------------------------------
"

# Pastas e arquivos de destino...
caminho_destino=$HOME/projetos
caminho_projeto=$caminho_destino/$1
arquivo_destino=$caminho_projeto/$1.sh
arquivo_symlink=$HOME/bin/$1

# Cria a pasta de destino...
mkdir -p $caminho_projeto

# Cria o arquivo de destino...
echo "$modelo" >> $arquivo_destino

# Torna o arquivo de destino executável...
chmod +x $arquivo_destino
```

```
# Cria uma ligação simbólica para o novo script...
ln -s $arquivo_destino $arquivo_symlink

nano $arquivo_destino
```

3 – Variáveis

No código ou na linha de comandos, uma variável é basicamente um *nome* (ou *identificador*) que aponta para uma determinada informação a que chamamos de *valor*. No Bash, como em qualquer outra linguagem, nós podemos criar, alterar e eliminar variáveis.

A forma mais direta de criação de uma variável é através de um tipo de operação chamada de *atribuição*. Numa atribuição, nós definimos um *nome*, e atribuímos a ele um *valor* utilizando o operador `=` (*operador de atribuição*).

Veja alguns exemplos:

Exemplo 3.1 – Atribuindo valores a variáveis.

```
:~$ fruta=banana
:~$ nota=10
:~$ nomes=(João Maria José)
:~$ retorno=$(whoami)
:~$ soma=$((a + 10))
```

> ***Atenção!*** *Não é permitido utilizar espaços antes ou depois do operador de atribuição!*

Além da operação de atribuição, nós também podemos criar variáveis através de alguns comandos, como o comando interno `read`, por exemplo, que lê um dado que o usuário digita no terminal ou que vem de um arquivo, e atribui o valor lido a um nome:

Exemplo 3.2 – Atribuindo valores lidos pelo comando 'read' a variáveis.

```
:~$ read -p 'Digite seu nome: ' nome
Digite seu nome: Renata
```

```
:~$ echo $nome
Renata
```

O *builtin* `declare` é outra forma de criação de variáveis. Com ele, entre outras coisas, nós também podemos especificar se o valor armazenado será de um tipo diferente do padrão (que é o tipo *indeterminado*).

Por exemplo:

Exemplo 3.3 – Definindo que o valor na variável 'soma' será um inteiro.

```
:~$ declare -i soma
```

Mas, tenha sempre em mente que as variáveis no Bash *não são declaradas*. Uma declaração, a rigor, envolveria a reserva de um certo espaço em *bytes* na memória de acordo com o tipo do dado que será armazenado na variável, e não é bem assim que o Bash funciona. Para começar, `declare` é um comando cuja função básica é definir e listar *atributos* de variáveis. Então, no exemplo acima, tudo que ele faz é dizer ao interpretador que os valores recebidos na variável `soma` deverão ser tratados como valores numéricos inteiros ou expressões que resultem em inteiros, e não como strings, como seria feito por padrão.

Tente analisar o exemplo abaixo:

Exemplo 3.4 – Definindo como o shell interpreta valores numéricos.

```
:~$ declare -i soma; a=15; b=10; c=$a+$b
:~$ echo $c
15+10
:~$ soma=$c
:~$ echo $soma
25
```

Repare que a variável `c` contém apenas uma string que se parece com uma soma, por isso ela é expandida como `15+10`. Por outro lado, a variável `soma`,

que possui o atributo de inteiro, definido pela opção `-i` do comando `declare`, permite a interpretação da string como uma expressão aritmética e, portanto, é expandida para o valor `25`.

Uma das consequências das variáveis não serem declaradas no Bash, é que este é um comando totalmente válido:

Exemplo 3.5 – Expandindo variáveis não definidas previamente.

```
:~$ echo $fruta; echo $?

0
```

Para o shell, o nome `fruta` não possui um valor associado, por isso a expansão é feita para uma string vazia (a quebra de linha do exemplo veio do comando `echo`) e nenhum erro é gerado.

3.1 – Nomeando variáveis

Os nomes das variáveis que nós mesmos criamos em nossos códigos e na linha de comandos podem conter apenas: letras maiúsculas e minúsculas da tabela ASCII, dígitos e o caractere sublinhado (`_`). Todas as combinações desses caracteres são válidas, menos uma: um número nunca pode aparecer como o primeiro caractere do nome.

Aqui estão alguns exemplos de nomes válidos:

```
fruta
Fruta
_fruta
fruta1
frutaVermelha
fruta_vermelha
```

Já estes nomes são inválidos:

```
:~$ maçã=fruta
bash: maçã=fruta: comando não encontrado

:~$ 2a=10
bash: 2a=10: comando não encontrado
```

3.2 – Vetores (*arrays*)

Quando um nome de variável aponta para um único valor, nós dizemos que essa variável é *escalar*. Mas, nós também podemos criar variáveis que apontam para uma coleção de valores – são os chamados *vetores*, também conhecidos como *arrays* ou *matrizes*.

No Bash, um vetor pode ser criado atribuindo ao nome uma lista de valores entre parêntesis:

Exemplo 3.6 – Criando um vetor a partir de uma lista.

```
:~$ alunos=(João Maria Renata)
```

Alternativamente, nós podemos criar o mesmo vetor do exemplo acima pela atribuição individual de valores a cada um de seus elementos:

Exemplo 3.7 – Criando um vetor a partir de seus elementos.

```
:~$ alunos[0]=João; alunos[1]=Maria; alunos[2]=Renata
```

Repare que o nome da variável é `alunos`, e os números que aparecem entre os colchetes são seus *índices* (ou *subscritos*), daí eles receberem o nome de *vetores indexados* (ou *arrays indexadas*). No Bash, este é o tipo padrão de vetor, mas também podemos criar vetores cujos índices são strings – os chamados *vetores associativos*. O detalhe, porém, é que esse tipo de vetor depende da definição de um atributo especial através do comando `declare`:

Exemplo 3.8 – Criando um vetor associativo.

```
:~$ declare -A carro
:~$ carro[vw]=fusca
:~$ carro[lindão]=jipe
:~$ carro[de boi]=carroça
```

Também podemos criar vetores indexados e associativos especificando seus índices quando a atribuição é feita com uma lista de valores:

Exemplo 3.9 – Especificando os índices nas listas de valores.

```
:~$ declare -A carro
:~$ carro=([vw]=fusca [fiat]=palio [ford]=k)
:~$ alunos=([3]=João [1]=Maria [5]=José)
```

> *No caso específico dos vetores indexados, se os índices não forem especificados, o shell atribuirá um número inteiro sequencial a partir do zero (`0`) ao índice de cada elemento de acordo com a sua ordem de aparição na lista.*

3.3 – Acessando valores

No código, assim como na linha de comandos, toda ocorrência do nome de uma variável iniciada com o cifrão (`$`) em um comando fará o shell *expandir* o valor armazenado na variável antes de executar o comando. Como vimos no capítulo anterior, a forma padrão de se fazer referência ao nome de uma variável é `${NOME}`, mas as chaves são dispensáveis quando se trata de uma variável escalar. Então, as duas formas de acesso, abaixo, são válidas:

```
:~$ a=banana; b=laranja
:~$ echo Fui ao mercado comprar $a e ${b}.
Fui ao mercado comprar banana e laranja.
```

Porém, se eu quisesse exibir *"comprar bananas e laranjas"*, veja o que aconteceria:

```
:~$ a=banana; b=laranja
:~$ echo Fui ao mercado comprar $as e ${b}s.
Fui ao mercado comprar e laranjas.
```

Neste caso, a variável `a` não foi expandida – na verdade, ela nem sequer foi referenciada no comando! Repare que, sob qualquer ponto de vista, nosso ou do shell, nós tentamos expandir uma variável não definida de nome `as`. A variável `b`, por sua vez, não passou por esse tipo de problema porque nós utilizamos as chaves para separar seu nome do restante da string. Corrigindo, então...

Exemplo 3.10 – Expandindo variáveis escalares.

```
:~$ a=banana; b=laranja
:~$ echo Fui ao mercado comprar ${a}s e ${b}s.
Fui ao mercado comprar bananas e laranjas.
```

> *Muitos programadores em Bash chamam isso de "proteger a variável". Embora faça algum sentido, eu evito utilizar e recomendar essa terminologia porque não se trata de uma "proteção", e sim da sintaxe padrão para a expansão de variáveis.*

Quando a variável é vetorial (indexada ou associativa), o uso das chaves e do *subscrito* (a parte entre colchetes) tornam-se obrigatórios:

Exemplo 3.11 – Expandindo variáveis vetoriais.

```
:~$ alunos=(João Maria José)
:~$ echo ${alunos[1]}
Maria
```

Sem o subscrito, com ou sem as chaves, o shell expande o elemento de índice `0` de uma *array* indexada:

```
:~$ echo ${alunos}
João
```

Para expandir todos os valores no vetor, nós utilizamos os símbolos `@` e `*` dentro dos colchetes:

Exemplo 3.12 – Expandindo todos os elementos de vetores.

```
:~$ alunos=(João Maria 'Luis Carlos')
:~$ echo ${alunos[*]}
João Maria Luis Carlos
```

Assim como vimos quando falamos de parâmetros posicionais, a diferença entre o `*` e o `@` está na forma como os elementos contendo espaços em branco são tratados na expansão – se o vetor for acessado entre aspas e com o `@` no subscrito, os espaços serão preservados como parte do elemento expandido:

Exemplo 3.13 – Expandindo todos os elementos de vetores.

```
:~$ alunos=(João Maria 'Luis Carlos')
:~$ for nome in ${alunos[*]}; do echo $nome; done
João
Maria
Luis
Carlos
:~$ for nome in "${alunos[@]}"; do echo $nome; done
João
Maria
Luis Carlos
```

3.4 – Indireções

É possível expandir os índices em vez dos elementos de um vetor utilizando uma exclamação (`!`) antes do nome e `*` ou `@` no subscrito:

Exemplo 3.14 – Expandindo os índices de um vetor.

```
:~$ alunos=(João Maria 'Luis Carlos')
:~$ echo ${!alunos[@]}
0 1 2
```

Quando utilizada em variáveis escalares, a exclamação faz com que seja expandida uma *indireção*, que é a expansão de uma variável cujo nome corresponde ao valor armazenado em outra variável.

> *Para utilizar símbolos que modificam as expansões de variáveis escalares, o uso das chaves torna-se obrigatório.*

Exemplo 3.15 – Expandindo uma indireção.

```
:~$ animal=cavalo; cavalo=equino
:~$ echo $animal
cavalo
:~$ echo ${!animal}
equino
```

3.5 – Número de elementos e de caracteres

As expansões também podem ser modificadas para que seja retornado o número de caracteres de uma variável escalar ou o número de elementos de um vetor:

Exemplo 3.16 – Expandindo números de caracteres e de elementos.

```
:~$ fruta=banana
:~$ animais=(zebra leão gnu)
:~$ echo ${#fruta}
6
:~$ echo ${#animais[@]}
3
:~$ echo ${#animais[1]}
4
```

3.6 – Escopo de variáveis

No nosso contexto, o *escopo* é a disponibilidade de uma variável em uma sessão do shell, em partes diferentes do código de um script, ou em sessões derivadas da sessão atual do shell (como *sessões filhas* e *subshells*). No Bash, a menos que se defina o contrário, uma variável sempre estará disponível em qualquer parte *da sessão corrente do shell*, ou seja: em princípio, todas as variáveis são *globais*. Todavia, quando falamos da disponibilidade das variáveis entre sessões diferentes do shell, existem, pelo menos, três situações que precisamos considerar.

3.6.1 – Sessões filhas

Uma *sessão filha* é uma sessão do shell iniciada a partir de uma sessão já em execução (*sessão mãe*) e é o que acontece, por exemplo, quando executamos um script ou o próprio executável do Bash na linha de comandos. Sendo mais técnico, uma sessão filha é um subprocesso derivado de um processo do shell que é o *líder da sessão*, como podemos ver com o nosso velho conhecido, o utilitário `ps` e a variável especial `$`, que expande para o número do processo (PID) do shell corrente.

```
:~$ echo $$
21335
:~$ bash
:~$ echo $$
22524
:~$ ps t
  PID TTY      STAT   TIME COMMAND
21335 pts/0    Ss     0:00 bash
22524 pts/0    S      0:00 bash
22599 pts/0    R+     0:00 ps t
```

Desta vez, nós utilizamos o `ps` com a opção `t` para que a saída ficasse limitada à exibição aos processos em execução no terminal e também para termos as informações da coluna `STAT`, especialmente o `s` minúsculo que aparece na linha do processo 21335, indicando que este é o *processo líder da sessão*.

> ***Importante!*** *Nem sempre a sessão do shell iniciada junto com uma nova instância de um terminal é um processo líder de sessão – isso depende das configurações do sistema operacional e do ambiente gráfico em uso.*

Voltando aos escopos, as variáveis criadas numa sessão mãe não estão disponíveis nas sessões filhas, e é por isso que dizemos que todas as variáveis de uma sessão do shell são, em princípio, *locais* em relação às demais sessões.

Observe o exemplo:

```
:~$ fruta=banana
:~$ echo $fruta
banana
:~$ bash
:~$ echo $fruta

:~$
```

Na sessão filha iniciada com a invocação do executável `bash`, a variável `fruta` não estava definida e nada foi expandido. Contudo, é possível "copiar" uma variável para as sessões filhas, e uma das formas de se fazer isso é com o comando interno `export`:

Exemplo 3.17 – Exportando variáveis para sessões filhas com 'export'.

```
:~$ fruta=banana
:~$ echo $fruta
banana
:~$ export fruta
:~$ bash
:~$ echo $fruta
banana
```

Um detalhe importante do comando `export`, é que ele altera um dos atributos da variável – o *atributo de exportação*. Quando ligado, este atributo faz com que a variável seja copiada para cada nova sessão criada a partir da sessão mãe original. Isso significa que `fruta` estaria disponível não apenas para a sessão filha que nós inciamos, mas também para as sessões netas, bisnetas e assim por diante.

Outra forma de manipular o atributo de exportação de uma variável é através do comando `declare` com a opção `-x`:

Exemplo 3.18 – Exportando variáveis para sessões filhas com 'declare'.

```
:~$ declare -x fruta=banana
:~$ echo $fruta
banana
:~$ bash
:~$ echo $fruta
banana
```

3.6.2 – Variáveis de ambiente

Uma *variável de ambiente* é uma variável que teve, em algum momento, o seu atributo de exportação ligado e, portanto, está disponível para as sessões filhas enquanto durar a sessão onde ela foi criada. Todavia, nos exemplos *3.17* e *3.18*, a variável `fruta` seria destruída assim que o terminal fosse fechado.

Para que isso não aconteça com as variáveis exportadas, quando um novo processo do shell é iniciado, um ou mais scripts de configuração são executados e elas são recriadas (foi por isso que pudemos usar a variável de ambiente `HOME` no script do *tópico 2.3.19*).

Para listar todas as variáveis com atributo de exportação, inclusive as que nós mesmos criamos durante uma sessão, basta executar:

Exemplo 3.19 – Listando variáveis de ambiente e exportadas.

```
:~$ export -p
```

A lista retornada é longa demais para ser transcrita neste livro, mas, se você testou os exemplos anteriores, provavelmente a variável `fruta` aparecerá assim em uma das últimas linhas:

```
declare -x fruta="banana"
```

Como estamos falando de sessões e ambientes, esta é uma ótima oportunidade para conhecermos a variável de ambiente `SHLVL` (que deve ter aparecido na listagem do *exemplo 3.19*, se você testou), cuja função é armazenar o nível relativo da sessão corrente do shell.

O resultado do exemplo abaixo pode variar de acordo com a sua distribuição GNU/Linux ou o seu ambiente gráfico, mas é bem provável que, se você abrir um terminal e expandir a variável `SHLVL`, o nível da sessão inicial do shell também seja `1`, o que será incrementado a cada subnível:

Exemplo 3.20 – Expandindo o nível relativo da sessão corrente do shell.

```
:~$ echo $SHLVL
1
:~$ bash
:~$ echo $SHLVL
2
:~$ bash
:~$ echo $SHLVL
3
```

3.6.3 – Subshells

Diferente das sessões filhas, um *subshell*, que também é um processo e tem o seu próprio PID, não requer a reconfiguração do shell, pois ele herda uma cópia de todas as variáveis (de ambiente ou não) em uso na sessão mãe, inclusive as variáveis `$` (PID da sessão corrente) e `SHLVL` (nível da sessão corrente), o que pode causar algumas confusões se você não estiver atento.

Veja o exemplo:

```
:~$ echo $$
30785
:~$ (echo $$)
30785
```

Os comandos agrupados com parêntesis, diferentes dos comandos agrupados com chaves, sempre são executados em um subshell. Como podemos ver no exemplo, porém, o PID expandido em `(echo $$)` foi o mesmo do processo que está em execução no terminal. Isso aconteceu porque, como dissemos, todas as variáveis de um subshell são cópias herdadas da sessão mãe.

Felizmente, o Bash possui uma variável que apenas ele pode controlar e, portanto, não pode ser copiada para o ambiente de um subshell – a variável `BASHPID`, que também armazena o PID da sessão corrente.

Sendo assim, agora nós temos como descobrir o PID de um subshell:

Exemplo 3.21 – Identificando o processo de um subshell.

```
:~$ echo $$ $BASHPID
30785 30785
:~$ (echo $$ $BASHPID)
30785 31763
```

No que diz respeito ao escopo, um subshell herda a uma cópia de todas as variáveis em uso na sessão mãe. Porém, como são cópias, as alterações feitas no subshell não são refletidas nas variáveis originais da sessão mãe.

Observe o que acontece no exemplo abaixo:

Exemplo 3.22 – Escopo das variáveis em um subshell.

```
:~$ fruta=banana; animal=gato
:~$ echo $fruta $animal
banana gato
:~$ (animal=zebra; echo $fruta $animal)
banana zebra
:~$ echo $fruta $animal
banana gato
```

Dentro do agrupamento com parêntesis, nós alteramos o valor da variável `animal` para `zebra`, mas a alteração só teve efeito na cópia da variável existente no subshell – a original continuou intacta. No final, isso é mais uma confirmação da regra que diz que, em princípio, as variáveis são sempre *locais* para uma sessão em relação às demais.

3.7 – Variáveis inalteráveis (*read-only*)

Normalmente, nós podemos criar variáveis com um determinado valor e mudá-lo à vontade quando necessário...

```
:~$ fruta=banana
:~$ echo $fruta
banana
:~$ fruta=laranja
:~$ echo $fruta
laranja
```

Porém, existem casos onde precisamos garantir que um determinado valor não seja mudado. Com o comando interno `readonly`, o shell retornará um erro caso ocorra uma tentativa de alterar o valor da variável.

Exemplo 3.23 – Tornando uma variável read-only com 'readonly'.

```
:~$ fruta=banana
:~$ readonly fruta
:~$ fruta=laranja
bash: fruta: a variável permite somente leitura
```

Outra forma de tornar uma variável inalterável (*read-only*), é com a opção `-r` do comando interno `declare`:

Exemplo 3.24 – Tornando uma variável read-only com 'declare -r'.

```
:~$ declare -r animal=gato
:~$ animal=zebra
bash: animal: a variável permite somente leitura
```

Variáveis inalteráveis só podem ser destruídas com o fim da sessão em que foram criadas.

Por falar em destruição...

3.8 – Destruindo variáveis

Internamente, quando criamos uma variável, ela entra para um tipo de tabela na memória que o shell utiliza para rastrear seu nome, seus atributos e as alterações do seu valor. Cada sessão do shell tem a sua própria tabela para controlar variáveis, logo, com o fim da sessão, todas as variáveis relacionadas com aquele processo deixam de existir.

Mas, antes de aprendermos a *destruir* uma variável, nós precisamos entender o que isso significa. Como as variáveis *não são declaradas* no Bash, não existe diferença prática entre uma variável *vazia* e uma variável *não definida*. Vejamos no exemplo:

```
:~$ fruta=
:~$ echo Eu quero uma $fruta para o meu $animal
Eu quero uma para o meu
```

Aqui, a variável `fruta` foi definida e iniciada com o valor de uma string vazia, mas a variável `animal` sequer foi definida. Em ambos os casos, a expansão resultou numa string vazia, mas por motivos totalmente diferentes: `fruta` existia e estava vazia, enquanto `animal` não existia e, portanto, não havia o que expandir. Isso pode ser confirmado com a opção `-p` do comando `declare`, utilizada para exibir os atributos de uma variável:

Exemplo 3.25 – Exibindo os atributos de uma variável 'declare -p'.

```
:~$ fruta=
:~$ declare -p fruta
declare -- fruta=""
:~$ declare -p animal
bash: declare: animal: não encontrado
```

Como dissemos, na prática não há diferença e, a menos que o uso de recursos do sistema seja uma preocupação, nós podemos atribuir uma string vazia à

variável sempre que quisermos torná-la nula. Mas, se nós realmente quisermos destruí-la, será preciso utilizar o comando interno `unset`:

Exemplo 3.26 – Destruindo uma variável com o comando 'unset'.

```
:~$ fruta=banana
:~$ declare -p fruta
declare -- fruta="banana"
:~$ unset fruta
:~$ declare -p fruta
bash: declare: fruta: não encontrado
```

3.9 – Variáveis são parâmetros

Uma das coisas que mais podem nos confundir no Bash é a sua terminologia. Mas, (quase) sempre há um bom motivo por detrás de cada escolha feita na hora de dar nomes aos bois, inclusive da decisão de chamar *variáveis* de *parâmetros*.

No Bash, como vimos no *tópico 2.3.8*, tudo que digitamos na linha de comando ou numa linha dos nossos códigos é um *parâmetro* que será passado para o interpretador. Por padrão, esses parâmetros são numerados de acordo com a sua ordem de aparição (por isso são chamados de *parâmetros posicionais*), de forma que o parâmetro `0` será o nome do comando ou do programa e todos os argumentos e opções que vierem depois serão numerados a partir de `1`. Então, podemos concluir que, do ponto de vista do Bash, tudo que escrevemos numa linha de comando são parâmetros numerados automaticamente.

Mas, veja o que aconteceria se o primeiro parâmetro fosse um nome que o shell é incapaz de identificar:

```
:~$ fruta
bash: fruta: comando não encontrado
```

Basicamente, o erro acima informa que não existe nenhum comando interno, programa, função ou apelido (*alias*) com o nome `fruta`. Para que o nome `fruta` tenha algum significado para o Bash, ele precisa receber algum valor – e a forma utilizada para dar significado a um nome vai determinar como ele será interpretado pelo shell: como uma função, um apelido ou uma variável.

Por exemplo:

```
# Criando e destruindo um apelido (alias)...
:~$ alias fruta='ls -la'
:~$ unalias fruta

# Criando e destruindo uma função...
:~$ fruta() { echo 'Olá, mundo!'; }
:~$ unset -f fruta

# Criando e destruindo uma variável...
:~$ fruta=banana
:~$ unset fruta
```

No exemplo do comando interno `alias`, o nome `fruta` recebeu um *comando simples* como significado; na criação da função, seu significado passou a ser um *comando composto*; na atribuição, contudo, o seu significado tornou-se apenas um *valor*. Este valor associado ao nome `fruta` pode ser alterado com o tempo, daí o termo *"variável"*, mas, para o shell, ele é um *parâmetro nomeado* que pode ser acessado e manipulado livremente – ao contrário dos parâmetros posicionais e especiais do shell que só podem ser controlados por ele mesmo.

Quando se programa em Bash, entender esses princípios é muito mais do que uma mera curiosidade. Às vezes, é muito mais importante saber dessas coisas do que saber que comando utilizar. Os comandos e seus exemplos de uso nós podemos encontrar em toda parte, mas nenhum desses exemplos e manuais fará com que você seja capaz de escolher o procedimento A ou B ou de entender por que eles não funcionaram no seu código. Aliás, é muito comum

nós não encontrarmos algo nos manuais porque simplesmente não conhecemos todos os conceitos por detrás dos termos utilizados.

O *capítulo 4* contém um tópico que é um bom exemplo disso: o tópico sobre *expansões de parâmetros (tópico 4.9)*. Aqui mesmo, até este ponto, eu só usei termos como *"expansão de variáveis"* para dar tempo de você chegar a essa explicação. Todavia, o termo *"variável"*, quando falamos do Bash, é só uma analogia, uma forma coloquial de abordar um conceito que possui uma equivalência em outras linguagens, mas é tratado de forma totalmente diferente pelo shell.

> *Nós podemos (e iremos) continuar chamando os parâmetros nomeados de variáveis. O objetivo aqui não é propor uma correção na forma como nos referimos ao conceito, mas esclarecer qual é, de fato, o conceito.*

3.10 – Parâmetros especiais

A tabela abaixo contém uma lista de parâmetros especiais do Bash que todos nós deveríamos tatuar na ponta da língua:

Parâmetro	O que expande...
`0` a `n`	Parâmetros posicionais passados para um script ou uma sessão do shell. O parâmetro `0` expande o nome do shell ou do script em execução.
`@`	Todos os parâmetros posicionais. Quando entre aspas duplas (`"$@"`), faz com que o shell trate parâmetros com espaços escapados como uma única palavra.
`*`	Todos os parâmetros posicionais.
`#`	Número total de parâmetros posicionais (desconta o parâmetro `0`).

`?`	Estado de saída do último comando executado.
`$`	PID da sessão corrente do shell.
`!`	Identificação do processo mais recente colocado para ser executado ao fundo.
`-`	A lista dos *flags* de configuração do shell.
`_`	No começo de uma sessão, expande o caminho utilizado para invocar o shell. Depois disso, expande para o último argumento do último comando executado.

4 – Expansões do shell

Se você leu algum dos capítulos anteriores, com certeza já se deparou com vários exemplos de expansões do shell. Mas, é aqui que nós vamos conhecer melhor este que é um dos recursos mais práticos e surpreendentes do Bash. Para entender o que são as expansões e o seu funcionamento, é fundamental que você seja capaz de visualizar como o shell processa a linha de um comando.

Quando nós escrevemos um comando e teclamos `Enter`, ou quando um *operador de controle padrão* (uma quebra de linha ou o `;`) é encontrado, o shell inicia uma análise da linha recebida. O primeiro passo dessa análise é a busca por componentes léxicos (*tokens*[10]) através da divisão da linha entre *palavras* e *operadores* seguindo as regras das aspas (*tópico 4.3*). É nesta primeira etapa que, se necessário, ocorre a *expansão dos aliases*[11].

4.1 – A expansão dos *aliases*

Se você observar bem o exemplo abaixo, fica evidente a semelhança entre a atribuição da string de um comando a uma variável e a forma como definimos um *alias*:

```
# Criando um 'alias'...
:~$ alias hd='help -d'
```

10 Um ***token***, também chamado de ***componente léxico***, é qualquer sequência de caracteres que possua um significado para a linguagem, como palavras-chave, operadores, identificadores ou até, no caso do shell, comandos, programas e seus argumentos (os *parâmetros* da linha de comando).

11 ***Alias*** significa ***apelido***, mas nós utilizaremos o termo original, em inglês, pelo fato de, neste contexto, não se tratar de um apelido qualquer – é a expansão de um apelido criado com o comando *alias*.

```
# Criando uma variável contendo a string de um comando...
:~$ hd='help -d'
```

Na verdade, tirando três fatos: a expansão da variável exige o `$` e o *alias*, além de ser expandido primeiro, vem configurado para funcionar apenas no modo interativo, as duas expansões provocam exatamente o mesmo efeito – a string do comando é executada:

```
:~$ hd alias
alias - Define or display aliases.
:~$ $hd help
help - Display information about builtin commands.
```

Eventualmente, nós podemos ficar tentados a utilizar alguns aliases nos nossos scripts, mas esta é uma opção que vem desabilitada por padrão na maioria dos sistemas operacionais. Com o comando interno `shopt`, você pode habilitar a opção `expand_aliases` antes de utilizá-los no seu código:

```
shopt -s expand_aliases
```

Mas, como vimos, a expansão de um parâmetro cumpre muito bem esse papel e, como ainda veremos, as funções cumprem melhor ainda!

> ***Atenção!*** *Nós podemos "criar" aliases no modo não-interativo por padrão, o que não podemos é "expandi-los"!*

4.2 – As outras expansões

Nas etapas seguintes, o shell realiza as demais expansões e classifica as partes resultantes da divisão da linha do comando em vários tipos de elementos, identificando comandos simples e compostos, argumentos, nomes de

arquivos, redirecionamentos, etc... Só então, se não houver nenhum erro, o comando é de fato executado.

Como regra genérica, você pode partir do princípio de que todas as expansões serão realizadas *antes* da execução da linha do comando, mas o shell segue uma ordem bem clara de precedência:

- Expansão de chaves;
- Expansão do til;
- Expansão de parâmetros;
- Expansões aritméticas;
- Substituição de comandos;
- Substituição de processos;
- Divisão de palavras;
- Expansões de nomes de arquivos;
- Remoção de aspas.

Neste capítulo, nós daremos preferência a uma ordem mais didática de apresentação das expansões, mas você precisa ficar atento à sequência acima para evitar surpresas nos seus códigos.

4.3 – Como funcionam as aspas

Como vimos no *tópico 2.3.17*, as aspas têm influência direta sobre a forma como as palavras são separadas na interpretação da linha de um comando. Uma string que contenha um espaço, por exemplo, pode ser vista como duas palavras ou como uma só, se estiver entre aspas:

```
olá mundo   → Sempre vista como duas palavras
'olá mundo' → Pode ser vista como uma palavra
```

Também vimos que o tipo das aspas, ou até a ausência delas, pode permitir ou impedir que as expansões do shell sejam realizadas:

```
:~$ echo "Pasta pessoal: ~"
Pasta pessoal: ~

:~$ echo 'Pasta pessoal: ~'
Pasta pessoal: ~

:~$ echo Pasta pessoal: ~
Pasta pessoal: /home/blau
```

Então, em vez de pensarmos nas aspas como uma forma de permitir que coisas aconteçam, é muito mais seguro (e correto) pensarmos que elas *removem* significados especiais de certos caracteres e palavras, que elas *inibem* certos tratamentos especiais, ou que elas *impedem* que certas expansões aconteçam.

Eu vou repetir, especialmente para aqueles que defendem que a forma mais segura de programar em Bash é sair colocando aspas em tudo: as aspas *removem*, *inibem* e *impedem* – elas são mecanismos *restritivos*!

Neste contexto, existem outros mecanismos restritivos no shell, entre eles: as *aspas simples*, as *aspas duplas* e o *caractere de escape* (`\`), e é o que veremos na sequência.

4.3.1 – Caractere de escape

O *caractere de escape* exerce sobre um caractere o mesmo efeito que as *aspas simples* exercem sobre strings. Uma barra invertida (`\`) remove o significado especial do caractere que vier depois dela, a menos que ela mesma esteja entre *aspas simples* e também perca o seu significado especial:

Exemplo 4.1 – Caractere de escape.

```
:~$ fruta=banana
:~$ echo \$fruta
$fruta
:~$ echo '\$fruta'
\$fruta
```

Quando utilizado antes de uma quebra de linha, o caractere de escape faz com que ela seja ignorada e a linha continue:

Exemplo 4.2 – Escapando quebras de linhas.

```
:~$ echo \ [Enter]
> olá, \   [Enter]
> mundo!   [Enter]
olá, mundo!
```

Esse recurso também pode ser utilizado nos scripts quando lidamos com linhas muito longas.

> *No modo interativo, o caractere* `>` *sempre é exibido quando estamos na continuação da linha de um comando – é o chamado "prompt de continuação interativa", que é definido pela variável de ambiente* `PS2`.

O caractere de escape pode escapar a si mesmo!

Exemplo 4.3 – Escapando o caractere de escape.

```
:~$ echo \\
\
```

4.3.2 – Aspas simples

As aspas simples (`'string'`) removem o significado especial de todos os caracteres na string entre elas, inclusive o caractere de escape. Por este

motivo, é impossível escapar o caractere `'` se ele aparecer na string, o que certamente causaria problemas.

Exemplo 4.4 – Retirando poderes especiais dos caracteres com aspas simples.

```
:~$ fruta=banana

:~$ echo '$fruta'
$fruta

:~$ echo '$fruta 'madura'
>  ← O shell não entende o comando como completo!

:~$ echo '$fruta' 'madura'
$fruta madura  ← São duas strings literais!
```

4.3.3 – Aspas duplas

As aspas duplas (`"string"`) também *removem* os significados especiais de todos os caracteres da string entre elas, exceto de quatro: do cifrão (`$`), do acento grave (`` ` ``), do caractere de escape (`\`) e, caso a expansão do histórico esteja habilitada, da exclamação (`!`).

> *O caractere de escape só tem seu significado especial preservado quando seguido de* `$`, `` ` ``, `"`, `\` ou de uma quebra de linha. Em qualquer outra situação, ele é literal.

É aqui que reside o maior problema, pois muitos dizem que as aspas duplas *"permitem que as expansões ocorram"*, quando o que acontece é justamente o contrário! Na verdade, em certos contextos, eu mesmo digo algo parecido, mas sempre no sentido de que elas permitem a *expansão de parâmetros* (variáveis) ou a *substituição de comandos*, ambas iniciadas com `$` ou entre acentos graves (`` ` ``), que são os caracteres que têm os seus "poderes" preservados. Todas as outras expansões são inibidas quando entre aspas

(sejam duplas ou simples), e foi isso que vimos no exemplo que demos logo no início do tópico:

```
:~$ echo "Pasta pessoal: ~"
Pasta pessoal: ~
```

> *Na verdade, a expressão que eu mais costumo utilizar é "para evitar a expansão de...", geralmente quando explico por que algo deve estar entre aspas simples.*

4.3.4 – Expansão de caracteres ANSI-C

Existe uma situação especial que ocorre quando as aspas simples são precedidas do cifrão (`$'string'`). Neste caso, se a string contiver os caracteres de controle do padrão ANSI-C, eles serão decodificados.

Os caracteres de controle ANSI-C são conjuntos formados pela barra invertida (`\`) seguida de um caractere ou de um valor numérico inteiro correspondente ao valor de um caractere na tabela ASCII.

4.3.5 – Espaços e quebras de linha

Um uso bem interessante para as expansões de caracteres ANSI-C, é a decodificação do caractere da quebra de linha (`\n`) com o comando `echo`. Repare, abaixo, que o caractere de escape (`\`) faz com que o `n` seja tratado literalmente:

```
:~$ echo banana\nlaranja
bananananlaranja
```

Uma alternativa para que `\n` seja visto como um caractere de controle seria a opção `-e` do comando `echo`, que habilita a interpretação dos caracteres ANSI-C, mas apenas se a string estiver entre aspas duplas ou simples:

```
:~$ echo -e banana\nlaranja
bananananlaranja  ← O 'n' foi escapado e nada mudou!

:~$ echo -e 'banana\nlaranja'
banana
laranja

:~$ echo -e "banana\nlaranja"
banana
laranja
```

Isso é uma consequência do processo da divisão de palavras que ocorre logo no início da análise de um comando: sem as aspas para delimitarem os argumentos e definirem como os caracteres da string serão interpretados, a barra invertida é vista como um caractere de escape antes do `echo` ter a chance de ver que ela estava lá.

> ***Importante!*** *Embora seja relativamente confiável em sistemas GNU/Linux, o comando* `echo` *com a opção* `-e` *não é consistente com todas as implementações do shell em sistemas unix-like, devendo ser evitado em favor de soluções como a expansão de caracteres de controle ANSI-C ou do comando interno* `printf`*.*

A outra alternativa, é a expansão de caracteres de controle ANSI-C:

Exemplo 4.5 – Expandindo caracteres de controle ANSI-C.

```
:~$ echo banana$'\n'laranja
banana
laranja
:~$ echo $'banana\nlaranja'
banana
laranja
```

Mas, repare que o comportamento é outro quando utilizamos as aspas:

```
:~$ echo "banana$'\n'laranja"
banana$'\n'laranja

:~$ echo 'banana$'\n'laranja'
banana$nlaranja
```

Com as aspas duplas, as aspas simples não possuem significado especial e o cifrão (`$`), portanto, não é seguido de nada que possa ser expandido. Com as aspas simples, por sua vez, nós temos duas strings literais (`banana$` e `laranja`) com um `n` literal entre elas.

Mas, nós ainda temos uma terceira opção: armazenar o caractere de quebra de linha numa variável!

```
:~$ nl=$'\n'

:~$ echo banana${nl}laranja
banana laranja

:~$ echo "banana${nl}laranja"
banana
laranja
```

Pode não parecer, mas funcionou muito bem nos dois casos, embora nós só possamos notar no segundo (não faz sentido testar com aspas simples, elas não expandem nada). Claro, não é o efeito que queremos, mas a presença do espaço entre as strings mostra que o caractere `\n` foi decodificado – o problema está, novamente, na forma como o shell analisa os comandos antes de executá-los.

Tente acompanhar:

1. Quando nós teclamos o `Enter` no primeiro exemplo (sem aspas), o shell ainda não sabia da existência de um caractere de quebra de linha

na string, logo, ele entende que está lidando com apenas uma linha de comando.

2. O shell também não encontrou um par de aspas envolvendo toda string, o que poderia sugerir uma string com múltiplas linhas (uma estrutura equivalente a um *here-doc*), logo, ele deduz que a quebra de linha está apenas separando os argumentos do comando `echo`, tal como um espaço ou uma tabulação fariam.
3. Todo caractere equivalente a um ou mais espaços em branco é colapsado para apenas um espaço – como, de fato, aconteceu.

Para entender melhor o segundo ponto da análise, observe o exemplo abaixo:

```
:~$ echo 'banana [Enter]
> laranja'       [Enter]
banana
laranja
```

Quando encontra aspas abertas no começo da string, o shell fica esperando pelo seu fechamento para concluir o comando. Neste caso, como existem as aspas, ele trata o argumento do comando `echo` como uma string com múltiplas linhas, o que é totalmente equivalente a isso:

```
:~$ echo $'banana\nlaranja'
banana
laranja
```

4.3.6 – Comentários – a restrição máxima

Nós não costumamos pensar nos comentários dessa forma, mas eles são o mecanismo restritivo mais radical que podemos encontrar no shell. Na linha de um comando, quando uma palavra é precedida de uma cerquilha (#), tudo, a partir dela, é simplesmente ignorado.

Exemplo 4.6 – Utilizando comentários.

```
:~$ echo olá # mundo
olá
:~$ # echo olá mundo
:~$
```

Mas, cuidado com as aspas:

```
:~$ echo 'olá # mundo'
olá # mundo
```

Aqui, a cerquilha perdeu seus "poderes" especiais, mas nem é só por conta das aspas simples. O ponto principal, neste caso, é que ela não está precedendo uma palavra, mas fazendo parte dela!

É raro, mas pode acontecer dos comentários estarem desabilitados para o *modo interativo*, e os exemplos acima podem não funcionar. Se for o seu caso, você pode habilitá-los com o `shopt`:

Exemplo 4.7 – Habilitando comentários no modo interativo.

```
:~$ shopt -s interactive_comments
```

Para o modo *não-interativo* (os nossos scripts) os comentários sempre estão habilitados – portanto, você não tem nenhuma desculpa para não comentar o seu código!

4.4 – Divisão de palavras

Quando as *expansões de parâmetros*, as *substituições de comandos* ou as *expansões aritméticas* não acontecem entre aspas duplas, o resultado pode conter um ou mais caracteres que, se estiverem definidos numa variável especial chamada `IFS` (de *internal field separator*, em inglês), funcionarão como terminadores de palavras. O shell, portanto, faz uma varredura do

resultado dessas expansões utilizando os caracteres em `IFS` para dividi-lo em *palavras*.

> *Uma "palavra" é qualquer sequência de caracteres que é tratada como uma unidade pelo shell.*

Por padrão, os separadores definidos na variável `IFS` são os caracteres *espaço*, *tabulação* e *quebra de linha*. Seja quais forem os separadores, porém, eles serão removidos do *início* e do *fim* da string resultante das expansões – os que sobrarem no "interior" da string, estes sim, serão tratados como separadores na divisão das palavras. Por fim, após a divisão, eles também serão removidos, de modo que restem apenas as palavras.

> *Se o valor em `IFS` for nulo, não haverá divisão de palavras!*

É uma prática bastante comum manipular o valor em `IFS` para forçar o shell a dividir o resultado de uma expansão da forma desejada. Isso não é errado nem oferece riscos, mas é um recurso que, pessoalmente, eu sempre tento evitar.

O problema não está em poder ou não alterar o valor em `IFS`, mas na motivação para isso. Na maioria das vezes, o programador é levado a esse tipo de solução apenas por não ter feito um tratamento correto dos dados ou porque seu código está desnecessariamente complexo.

Além disso, nós sempre devemos ponderar se uma solução em *Bash puro* é realmente a solução mais adequada. Especialmente no trabalho com dados em texto, eu sempre me pergunto: será que não está na hora de recorrer a um utilitário do sistema, como o `awk` ou o `sed`, por exemplo? Como diz o nosso amigo, Paulo Kretcheu:

> *O fato de ser possível fazer alguma coisa não significa que nós somos obrigados a fazer essa coisa.*

4.5 – Remoção de aspas

Este deve ser o tópico mais curto de todo este livro, mas é uma etapa muito importante no processamento de expansões e, portanto, não pode ser ignorada: após todas as outras expansões serem realizadas, todas as ocorrências de *caracteres de escape* (`\`), aspas simples (`'`) e aspas duplas (`"`) que não resultarem diretamente dessas expansões serão removidas.

4.6 – Expansão do til

Seja por ter lido este livro desde o início até aqui, ou por já conhecer o uso do shell no terminal, é bem provável que você saiba que o til (`~`) é expandido para o caminho completo das nossas pastas pessoais:

Exemplo 4.8 – Expandindo o til.

```
# Expandir o til...
:~$ echo ~
/home/blau

# é o mesmo que expandir a variável HOME...
:~$ echo $HOME
/home/blau
```

Mas, o que talvez você não saiba, é que o Bash nos permite fazer algumas coisas bem interessantes com ele.

4.6.1 – Expandindo pastas de usuários específicos

Por exemplo, digamos que o seu sistema seja compartilhado com outros usuários – você também pode expandir as pastas pessoas deles com o til:

Exemplo 4.9 – Expandindo a pasta de um usuário específico.

```
:~$ echo ~blau
/home/blau
:~$ echo ~nena
/home/nena
```

4.6.2 – Expandindo o diretório corrente

Agora, vamos supor que você seja um adepto do minimalismo e, por isso, removeu todas as referências ao diretório de trabalho corrente do seu prompt. Para saber onde está, claro, você sempre pode expandir a variável `PWD` ou executar o comando interno `pwd`, mas também pode obter a mesma informação com uma expansão do til:

Exemplo 4.10 – Expandindo o diretório corrente.

```
:~$ echo $PWD
/home/blau/estou/aqui

:~$ pwd
/home/blau/estou/aqui

:~$ echo ~+
/home/blau/estou/aqui
```

4.6.3 – Expandindo o último diretório visitado

O til também permite a expansão do último diretório visitado:

Exemplo 4.11 – Expandindo o último diretório visitado.

```
:~$ cd Documementos
:~/Documentos$ echo ~-
/home/blau
:~/Documentos$ cd -
:~$ echo ~-
/home/blau/Documentos
```

Repare que o comando interno `cd` faz com que o caractere especial `-` (traço) seja expandido para o conteúdo da variável interna `OLDPWD`, que é utilizada pelo shell para armazenar o último diretório visitado, que é basicamente o que a expansão `~-` faz. Então, os comandos abaixo são equivalentes:

```
:~$ cd Documementos
:~/Documentos$ cd ~-
:~$ cd ~-
:~/Documentos$

:~$ cd Documementos
:~/Documentos$ cd $OLDPWD
:~$ cd $OLDPWD
:~/Documentos$

:~$ cd Documementos
:~/Documentos$ cd -
:~$ cd -
:~/Documentos$
```

4.6.4 – Expandindo diretórios empilhados

Além do comando `cd`, o shell possui mais dois comandos internos que alteram o diretório corrente: `pushd` e `popd`. A diferença deles para o `cd` é que, além da troca do diretório corrente, eles armazenam os diretórios visitados numa pilha que pode ser listada com o comando interno `dirs`.

Enquanto `pushd` e `popd` não são usados, a pilha de diretórios contém apenas o diretório corrente:

```
:~$ dirs
~
```

Com `pushd`, nós mudamos o diretório corrente e o inserimos no topo da pilha:

Exemplo 4.12 – Empilhando diretórios com 'pushd'.

```
:~$ pushd testes
~/testes ~
:~/testes$ pushd ~/Downloads
~/Downloads ~/testes ~
:~/Downloads$ pushd ~
~ ~/Downloads ~/testes ~
:~$
```

Com `popd`, nós podemos remover uma das entradas da pilha a partir de seu número (a partir do zero). A contagem pode ser positiva (`+N`), para a localização ser feita da esquerda para a direita, ou negativa (`-N`), para contarmos na direção contrária:

Exemplo 4.13 – Removendo diretórios da pilha com 'popd'.

```
:~$ dirs
~ ~/Downloads ~/testes ~
:~$ popd +2
~ ~/Downloads ~
```

Com o comando `dirs`, nós também podemos exibir apenas uma das entradas da pilha utilizando o mesmo sistema de indexação de `popd`:

Exemplo 4.14 – Exibindo diretórios específicos da pilha com 'dirs'.

```
:~$ dirs
~ ~/Downloads ~/testes ~
:~$ dirs +1
```

```
~/Downloads
:~$ dirs -1
~/testes
```

O til também pode ser expandido para uma dessas entradas da pilha, tal como o comando `dirs` exibiria com a opção `-l`:

Exemplo 4.15 – Expandindo entradas da pilha de diretórios com '~'.

```
:~$ dirs -l
/home/blau /home/blau/notas /home/blau/testes

:~$ echo ~0
/home/blau
:~$ echo ~-0
/home/blau/testes
:~$ echo ~1
/home/blau/notas
```

4.6.5 – Atenção para a ordem da expansão do til

Não podemos nos esquecer de que a expansão do til é a segunda a ser feita, logo depois da *expansão de sequências entre chaves* e antes das *expansões de parâmetros*. Isso significa que não podemos concatenar a expansão de uma variável com ele esperando que ocorra a expansão de algum de seus recursos. Por exemplo:

```
:~$ echo ~$USER
~blau
```

Como a variável de ambiente `USER` não estava expandida quando o til foi expandido, ele foi interpretado como o caractere literal `~`.

> *A ordem que o Bash obedece no processamento das expansões é que me faz optar quase sempre pela expansão da variável* `HOME` *em vez da expansão do til nos meus códigos – os parâmetros são expandidos na mesma etapa, o que torna tudo muito mais simples, seguro e consistente.*

4.7 – Expansões de chaves

Uma *expansão de chaves* é um mecanismo que o shell oferece para a geração de strings que contenham algum tipo de sequência. Para que ela aconteça, a linha do comando deve conter um padrão formado por um preâmbulo (opcional) seguido de uma expressão representando uma sequência entre chaves e um apêndice (também opcional):

```
Padrão: preâmbulo{sequência}apêndice
```

O preâmbulo e o apêndice serão repetidos em todas as strings geradas a partir da expansão das sequências:

Exemplo 4.16 – Expansões de chaves.

```
:~$ echo {1..5}
1 2 3 4 5
:~$ echo a{1..5}
a1 a2 a3 a4 a5
:~$ echo {1..5}b
1b 2b 3b 4b 5b
:~$ echo a{1..5}b
a1b a2b a3b a4b a5b
```

As sequências podem ser intervalos de *caracteres* da tabela ASCII, intervalos de números inteiros, ou listas de strings:

```
# Intervalo de inteiros...
:~$ echo {1..5}
1 2 3 4 5

# Intervalo de caracteres da tabela ASCII...
:~$ echo {a..e}
a b c d e

# Intervalo de caracteres na ordem reversa...
:~$ echo {e..a}
e d c b a

# Isso é o que existe na tabela ASCII entre ‘Z’ e ‘a’...
:~$ echo {Z..a}
Z [  ] ^ _ ` a  ← O caractere ‘\’ foi omitido pelo shell!

# Uma lista de strings...
:~$ echo ba{na,nda,ca}na
banana bandana bacana
```

Quando usamos as expansões de chaves, nós devemos ficar atentos a três regrinhas básicas:

1. Os espaços no padrão devem ser *escapados*.
2. Os caracteres do início de do fim das sequências devem ser do mesmo tipo.
3. As expansões de chaves são realizadas antes de que qualquer outra expansão, portanto, não podemos expandir variáveis no padrão.

Aqui estão alguns exemplos de expansões *inválidas*:

Exemplo 4.17 – Expansões de chaves inválidas.

```
# Início e fim não são do mesmo tipo...
:~$ echo {1..z}
{1..z}

# Tentando expandir parâmetros numa sequência...
```

```
:~$  a=12; b=23
:~$ echo {$a..$b}
{12..23}

# Espaço não escapado...
:~$ echo a {1..5}
a 1 2 3 4 5
```

O escape dos espaços pode ser feito de várias formas:

Exemplo 4.18 – Escapando espaços nos padrões de expansões de chaves.

```
:~$ echo a\ {1..5}
a 1 a 2 a 3 a 4 a 5

:~$ echo a' '{1..5}
a 1 a 2 a 3 a 4 a 5

:~$ echo a{b\ a,c\ b,d\ c}b
ab ab ac bb ad cb
```

Quando a sequência é um intervalo, nós ainda podemos definir um salto entre os valores ou os caracteres:

Exemplo 4.19 – Definindo saltos nos intervalos de expansões de chaves.

```
:~$ echo {1..10..2}
1 3 5 7 9

:~$ echo {a..j..2}
a c e g i
```

Quando o intervalo é de números inteiros, nós podemos preencher dígitos à esquerda com zeros:

Exemplo 4.20 – Preenchendo dígitos à esquerda com zeros.

```
:~$ echo {072..112..10}
072 082 092 102 112
```

4.8 – Expansão de nomes de arquivos

É provável que você já esteja bem familiarizado com esse tipo de expansão (também chamada de *glob*), que é aquela que nós utilizamos quando precisamos representar vários nomes de arquivos com algo em comum através de símbolos especiais. Mas, é igualmente provável que você ainda não tenha entendido o que realmente acontece quando utilizamos os asteriscos e as interrogações para listar ou manipular arquivos na linha de comandos, e é nisso que vamos trabalhar agora.

4.8.1 – Como funciona a expansão de nomes de arquivos

Quando o shell analisa uma linha de comando, há um momento em que ele busca pelos caracteres `*`, `?` e `[`. Se um desses caracteres aparecer numa palavra, ela será identificada como sendo um *padrão* e será comparada com uma lista de nomes de arquivos. Para cada nome de arquivo que corresponder ao padrão, uma string com o nome desse arquivo será gerada na expansão. Se nada for encontrado, o Bash normalmente mantém a palavra na sua forma original na linha do comando, a menos que esteja configurado para gerar uma string vazia.

> *É a opção* `nullglob` *que determina se haverá ou não a expansão para uma string vazia no caso do padrão não possuir correspondência. Ela vem desligada por padrão na maioria dos sistemas, mas pode ser alterada com o comando* `shopt`.

O que quase sempre passa despercebido por nós, é o fato de que a expansão de nomes de arquivos é apenas um mecanismo de *geração de strings* – como a *expansão de chaves* que vimos no tópico anterior. Por exemplo, quando nós digitamos o comando `ls *.txt` no terminal, não é o utilitário `ls` que vai tentar entender o que é `*.txt` para sair procurando arquivos com a terminação `.txt`.

Na verdade, o `ls` podem nem chegar a saber que nós digitamos `*.txt`. Antes dele receber qualquer argumento, o shell vai proceder a expansão dos nomes de arquivos para, se houver correspondências, gerar uma lista de nomes de arquivos que atendem ao padrão. Caso contrário, o `*.txt` será deixado como está e, obviamente, o `ls` não encontrará um arquivo com este nome e sairá com erro.

Por exemplo, considere um diretório com os arquivos abaixo:

```
aula.doc
aula.md
aula.txt
exemplo.doc
exemplo.md
exemplo.txt
teste.doc
teste.md
teste.txt
```

Se quisermos listar todos os nomes de arquivos que terminam com `.md`, nós executamos:

```
:~$ ls *.md
```

Mas, antes do comando ser executado, o shell substituiu `*.md` pela lista de nomes de arquivos que ele gerou ao casar o padrão com a lista dos arquivos existentes no diretório. Então, no exemplo, é como se nós tivéssemos digitado:

```
:~$ ls aula.md exemplo.md teste.md
```

4.8.2 – Listando arquivos com o comando 'echo'

Na verdade, se o objetivo fosse apenas exibir uma lista de nomes de arquivos que correspondem ao padrão, nós nem precisaríamos do `ls`, pois o shell só

gera strings com os nomes dos arquivos que existirem no diretório. Então, o comando abaixo é perfeitamente válido para esta finalidade:

Exemplo 4.21 – Exibindo uma expansão de nomes de arquivos.

```
:~$ echo *.md
aula.md exemplo.md teste.md
```

Agora, imagine que você procura um arquivo iniciado com o caractere `b`. Como não existe nenhum no nosso diretório hipotético, a expansão não acontece e o padrão é exibido intacto:

```
:~$ echo b*
b*
```

Com o `ls`, nós teríamos um erro:

```
:~$ ls b*
ls: não foi possível acessar 'b*': Arquivo ou diretório
inexistente
```

4.8.3 – A opção 'nullglob'

Repare, na mensagem de erro, que `b*` foi o argumento que o utilitário recebeu. Mas, este comportamento de manter a string do padrão pode ser alterado com a opção do shell `nullglob`. Com ela habilitada, padrões sem correspondência resultam em uma string vazia:

Exemplo 4.22 – Efeito da opção 'nullglob' habilitada.

```
# Comportamento padrão...
:~$ echo b*
b*

# Habilitando 'nullglob'...
```

```
:~$ shopt -s nullglob

# Novo resultado da expansão...
:~$ echo b*

:~$
```

Consequentemente, o utilitário `ls` não teria argumentos e exibiria:

```
:~$ ls b*
aula.doc  aula.txt     exemplo.md   teste.doc  teste.txt
aula.md   exemplo.doc  exemplo.txt  teste.md
```

Para restaurar o estado de `nullglob`:

```
:~$ shopt -u nullglob
```

4.8.4 – Formação de padrões

A formação básica dos padrões para os nomes de arquivos utiliza apenas os três símbolos da tabela abaixo:

Símbolo	Significado
`*`	Zero ou mais caracteres.
`?`	Um caractere.
`[…]`	Um caractere listado entre os colchetes.

Mas, existem alguns detalhes que precisamos conhecer sobre eles, especialmente sobre o asterisco (`*`) e a lista (`[…]`).

O *asterisco* (`*`)...

- Casa com qualquer string, inclusive uma string vazia.

- O padrão `*/` casa apenas com diretórios.
- Se a opção do shell `globstar` estiver habilitada, dois asteriscos seguidos (`**`) casarão com todos os arquivos de zero ou mais diretórios e seus subdiretórios.
- Com `globstar` habilitado, o padrão `**/` casa apenas com diretórios e subdiretórios.

A *lista* (`[…]`)...

- Casa com um dos caracteres entre os colchetes.
- O nome oficial da lista é *classe de caracteres*.
- Se o primeiro caractere listado for `^` ou `!`, a lista representará os caracteres proibidos.
- Um par de caracteres separados por um traço (`-`) indicam uma faixa de caracteres da tabela definida pelos padrões de localização do sistema (veja o *tópico 2.3.13* para mais detalhes).
- Para forçar o uso da tabela de caracteres da *localidade C* (ASCII) nas faixas, a variável de ambiente `LC_ALL` deve receber o valor `C` (também no *tópico 2.3.13*).
- Para diferenciar um traço literal (`-`) do traço que especifica uma faixa de caracteres (`x-y`), ele deve ser o primeiro caractere da lista.
- Da mesma forma, para diferenciar um caractere `]` do colchete que fecha a lista, ele deve vir no começo da lista.
- A lista pode conter as classes de caracteres POSIX (veja o *tópico 2.3.11*).

A expansão de nomes de arquivos ignora os arquivos ocultos, a menos que seja incluído um ponto (`.`) no início do padrão ou que a opção do shell `dotglob` esteja habilitada (e não é comum estar).

O ponto não tem significado especial nas expansões de nomes de arquivos. Para o shell, eles só têm algum valor quando são o primeiro caractere do

nome de um arquivo, o que indica que o arquivo é oculto e não deve ser, em princípio, incluído na listagem que será comparada com o padrão. Portanto, o único sentido de escrever padrões do tipo `*.*`, é se você estiver procurando arquivos que tenham um ponto em qualquer posição de seus nomes.

4.8.5 – Ignorando nomes de arquivos

A variável especial `GLOBIGNORE` pode ser utilizada para excluir certos padrões do resultado de uma expansão de nomes de arquivos. Por exemplo, se quisermos ignorar os nomes iniciados com o caractere `a` no padrão `*md`, basta incluir o padrão `a*` na variável:

Exemplo 4.23 – Ignorando padrões de nomes de arquivos.

```
:~$ echo *md
aula.md exemplo.md teste.md
:~$ GLOBIGNORE=a*
:~$ echo *md
exemplo.md teste.md
```

Mas, isso tem alguns efeitos colaterais. O primeiro, é que o padrão a ser ignorado é aplicado a todas as expansões de nomes de arquivo:

```
:~$ echo *
exemplo.doc exemplo.md exemplo.txt teste.doc
teste.md teste.txt
```

Outro efeito colateral é que, quando `GLOBIGNORE` está definida e não é nula, os arquivos ocultos (iniciados com ponto) passam a figurar nas expansões:

```
# Criando um arquivo oculto...
:~$ >> .oculto

:~$ echo *
exemplo.doc exemplo.md exemplo.txt .oculto teste.doc
teste.md teste.txt                    ↑
```

Nós também podemos especificar vários padrões a serem ignorados separando-os com dois pontos (`:`):

```
GLOBIGNORE=padrão1:padrão2:…
```

Para restaurar o comportamento normal da expansão de nomes de arquivos, basta destruir `GLOBIGNORE` ou atribuir um valor nulo a ela:

```
# Atribuindo um valor nulo...
:~$ GLOBIGNORE=

# Destruindo...
:~$ unset GLOBIGNORE
```

4.8.6 – 'Globs' estendidos

Os padrões estendidos para expansão de nomes de arquivos (também chamados de *globs estendidos*) nos oferecem algumas formas adicionais para formar padrões que seriam impossíveis de serem representados apenas com os três símbolos que já conhecemos. O problema, porém, é que a disponibilidade desse recurso depende de uma opção do shell estar habilitada: a opção `extglob`. O mais comum, é que ela venha habilitada para uso no modo interativo e desabilitada no modo não-interativo, mas você pode testar se este é o seu caso:

Exemplo 4.24 – Testando o estado da opção 'extglob'.

```
# Testando o estado no modo interativo...
:~$ shopt extglob
extglob            on

# Testando o estado no modo não-interativo...
:~$ bash -c 'shopt extglob'
extglob            off
```

> *O comando* `bash -c` *faz com que a string seja executada como uma linha de comando no shell não-interativo.*

Nos nossos scripts, nós podemos habilitar a opção `extglob`, mas isso só pode ser feito *antes e fora de qualquer função ou comando composto*!

Por exemplo, assim:

Exemplo 4.25 – Alterando o estado da opção 'extglob' no script.

```
#!/usr/bin/env bash

shopt -s extglob

… seu código que depende dos globs estendidos …

# Desabilitar é opcional...
shopt -u extglob
```

> *Normalmente, você não precisa (e nem deve) se preocupar em desabilitar a opção* `extglob` *no seu script. Além da mudança só afetar a sessão em curso, pode haver complicações se você precisar habilitá-la novamente.*

A tabela abaixo mostra o que podemos fazer com os globs estendidos:

Sintaxe	Descrição
`?(lista-de-padrões)`	Casa com *zero ou uma* ocorrência dos padrões.
`*(lista-de-padrões)`	Casa com *zero ou mais* ocorrências dos padrões.
`+(lista-de-padrões)`	Casa com *uma ou mais* ocorrências dos padrões.
`@(lista-de-padrões)`	Casa com um dos padrões na lista.
`!(lista-de-padrões)`	Casa com tudo menos os padrões na lista.

Aqui, `lista-de-padrões` é um ou mais padrões de nome de arquivos formados com os três símbolos básicos já vistos. Os padrões são separados entre si por uma barra vertical (`|`) que pode ser lida como a palavra "ou".

Utilizando o diretório hipotético abaixo, vejamos alguns exemplos:

Exemplo 4.26 – Exemplos de globs estendidos.

```
:~$ echo *
aaa aab aac aba abb abc aca acb acc baa bab
bac bba bbb bbc bca bcb bcc caa cab cac cba
cbb cbc cca ccb ccc

# Nomes que contenham apenas caracteres 'a' ou 'b'...
:~$ echo +(a|b)
aaa aab aba abb baa bab bba bbb

# Qualquer nome que 'não contenha apenas' 'a' ou 'b'...
:~$ echo !(+(a|b))
aac abc aca acb acc bac bbc bca bcb bcc caa
cab cac cba cbb cbc cca ccb ccc

# Nomes que contenham apenas o caractere 'c'...
:~$ echo +(c)
ccc

# Nomes que contenham apenas 'a', 'b' ou 'c'...
:~$ echo @(+(a)|+(b)|+(c))
aaa bbb ccc

# Nomes que comecem com 'aa' ou 'bb' e terminem com 'c'...
:~$ echo @(aa|bb)c
aac bbc
```

Porém, a melhor parte do uso dos globs estendidos é que eles podem ser utilizados em expressões de comparação de strings (*tópico 2.3.9*), permitindo a formação de padrões que só poderiam ser representados com expressões regulares:

Exemplo 4.27 – Globs estendidos em comparações de strings.

```
:~$ str=banana
:~$ [[ $str == ba+(na|nda)na ]]; echo $?
0
:~$ str=bandana
:~$ [[ $str == ba+(na|nda)na ]]; echo $?
0
:~$ str=bacana
:~$ [[ $str == ba+(na|nda)na ]]; echo $?
1
```

4.9 – Expansão de parâmetros

A *expansão de parâmetros*, como vimos no *Capítulo 3 – Variáveis* (e em diversos exemplos em quase todos os tópicos deste livro), é aquela que acontece quando o shell encontra um cifrão (`$`) seguido de um par de chaves contendo o identificador de um *parâmetro* (basicamente, o nome de uma *variável*).

Quando o nome do parâmetro representa um valor escalar, as chaves podem ser dispensadas, sendo utilizadas apenas para diferenciar o identificador do restante de uma string ou quando a expansão contiver algum símbolo modificador.

```
:~$ animal=gato             # um parâmetro 'escalar'
:~$ nomes=(João Maria José) # Um parâmetro 'vetorial'

# Escalar, as chaves não são obrigatórias...
:~$ echo $animal
gato

# Vetor, as chaves são obrigatórias...
:~$ echo ${nomes[@]}
João Maria José

# Diferenciando o parâmetro do restante da string...
```

```
:~$ echo Os ${animal}s são felinos
Os gatos são felinos
```

> ***Importante!*** *As modificações das expansões de parâmetros não alteram os valores no parâmetro, elas só atuam sobre a forma como eles serão expandidas, exceto quando uma expansão é utilizada para atribuir valor a um parâmetro condicionalmente.*

Também no *Capítulo 3*, nós já exploramos as expansões mais básicas dos parâmetros:

> ***3.3 – Acessando valores*:**
> `${nome}` `${nome[índice]}` A expansão de valores *escalares* e *vetoriais*;
>
> ***3.4 – Indireções:***
> `${!nome}` A expansão dos índices de um *vetor* ou de um parâmetro cujo nome é o valor de outro parâmetro;
>
> ***3.5 – Número de elementos e de caracteres*:**
> `${#nome}` O número de elementos de um vetor ou do número de caracteres de um valor escalar.

O que nos deixa bem à vontade para explorarmos as expansões mais incríveis e surpreendentes que o shell é capaz de fazer com os valores dos parâmetros!

4.9.1 – Substrings

Não é raro nós precisarmos de uma parte de uma string em vez de todo o seu conteúdo, e o shell oferece duas formas de fazermos isso com expansões: pela *posição dos caracteres* iniciais e finais da substring desejada, ou através da localização de *padrões*. Neste tópico, nós veremos apenas a primeira forma, indicando as posições dos caracteres.

Para facilitar, vamos considerar a string na variável abaixo:

```
:~$ str=abcdefghij
```

A expansão de uma substring pode ser feita da esquerda para a direita (de `a` para `j`) ou na direção contrária (de `j` para `a`). O que determina a direção da contagem é o sinal do número que indica a posição do caractere – números positivos são contados a partir do início da string e negativos são contados a partir do final.

A contagem sempre é feita a partir do zero (`0`) e deve ser imaginada como um ponto *antes* dos caracteres:

```
  .a  .b  .c  .d  .e  .f  .g  .h  .i  .j
  0   1   2   3   4   5   6   7   8   9
-10  -9  -8  -7  -6  -5  -4  -3  -2  -1
```

A expansão da substring é feita com a definição da posição inicial e da quantidade de caracteres a serem expandidos, ambas antecedidas por dois pontos (`:`). Mas, vamos com calma, vejamos como ela funciona apenas com a definição do ponto inicial:

Exemplo 4.28 – Expandindo caracteres a partir de uma posição 'até o fim'.

```
# Posições iniciais positivas...
:~$ echo ${str:0}
abcdefghij
:~$ echo ${str:1}
bcdefghij
:~$ echo ${str:2}
cdefghij

# Posições iniciais negativas (o espaço é obrigatório!)...
:~$ echo ${str: -1}
j
:~$ echo ${str: -2}
ij
```

```
:~$ echo ${str: -3}
hij
```

Como podemos ver, o número da *posição inicial* determina o ponto em que a string *começará a ser expandida*.

O segundo modificador, *se for positivo*, determina a *quantidade* de caracteres que será expandido a partir do ponto inicial:

Exemplo 4.29 – Expandindo faixas de caracteres pela quantidade.

```
:~$ echo ${str:0:3}
abc
:~$ echo ${str:4:2}
ef
:~$ echo ${str:2:5}
cdefg
:~$ echo ${str: -3:2}
hi
:~$ echo ${str: -4:3}
ghi
:~$ echo ${str: -6:4}
efgh
```

Se o segundo modificador for *negativo*, ele indicará o ponto final da expansão da substring:

Exemplo 4.30 – Expandindo faixas de caracteres pela posição final.

```
:~$ echo ${str:0:-5}
abcde
:~$ echo ${str:4:-3}
efg
:~$ echo ${str: -6:-3}
efg
:~$ echo ${str: -3:-1}
hi
:~$ echo ${str: -10:-5}
abcde
```

> ***Importante!*** *A posição final não pode ser um ponto anterior à posição inicial!*

Da mesma forma que a posição final, quando omitida (como vimos no *exemplo 4.28*), é assumida como *"até o fim"*, quando omitimos a posição inicial, ela será assumida como *"do início"*:

Exemplo 4.31 – Expandindo faixas de caracteres 'do início' até a posição final.

```
:~$ echo ${str::3}
abc
:~$ echo ${str::-3}
abcdefg
:~$ echo ${str::5}
abcde
:~$ echo ${str::-5}
abcde
:~$ echo ${str::1}
a
:~$ echo ${str::-1}
abcdefghi
```

Algumas dicas que você mesmo poderia deduzir de tudo que vimos:

- Remoção do último caractere: `${nome::-1}`
- Remoção do primeiro caractere: `${nome:1}`
- Remoção dos *"n"* últimos caracteres: `${nome::-n}`
- Remoção dos *"n"* primeiros caractere: `${nome:n}`

Uma dica extra: como todas as expansões de parâmetros acontecem na mesma etapa da avaliação da linha do comando, elas podem ser utilizadas umas dentro das outras (aninhadas):

Exemplo 4.32 – Aninhando expansões de parâmetros.

```
:~$ final=5
:~$ echo ${str:3:$final}
defgh
```

No próximo tópico, nós veremos como este mesmo modificador se comporta quando o parâmetro é um *vetor*.

4.9.2 – Faixas de elementos de arrays

Agora, vamos supor que a nossa variável seja um *vetor* (uma *array*):

```
:~$ compras=(banana laranja leite ovos presunto queijo)
```

Com um vetor criado desta forma, o shell atribuirá a cada elemento um índice numérico inteiro a partir de zero (`0`). Isso permite a expansão de um elemento a partir da indicação de seu índice, mas nós estamos interessados em expandir uma faixa de elementos. Portanto, teremos que pensar em termos de *posições*, assim como fizemos com a string do tópico anterior:

```
 .banana .laranja .leite .ovos .presunto .queijo
 0       1        2      3     4         5
-6      -5       -4     -3    -2        -1
```

Entendido esse preceito básico, tudo que vimos sobre a expansão de substrings é diretamente aplicável aos elementos de um vetor.

> ***Lembre-se:*** *para expandir todos os elementos de um vetor nós utilizamos o* `@` *ou o* `*` *no subscrito!*

Exemplo 4.33 – Expandindo faixas de elementos de um vetor.

```
# Expandindo o primeiro elemento...
:~$ echo ${compras[@]::1}
banana

# Expandindo o último elemento...
:~$ echo ${compras[@]: -1}
queijo

# Expandindo do terceiro ao quarto elemento...
:~$ echo ${compras[@]:2:3}
leite ovos presunto
```

Mas, *atenção*! Se o vetor for do tipo *associativo*, aquele que tem strings como índices (*tópico 3.2*), não existe uma definição de como será feita a ordenação dos elementos, por exemplo:

```
# Definindo o atributo de 'vetor associativo'...
:~$ declare -A carros

# Populando o vetor...
:~$ carros[vw]=fusca
:~$ carros[fiat]=147
:~$ carros[ford]=K
:~$ carros[willis]=jeep

# Expandindo os valores dos elementos...
:~$ echo ${carros[@]}
jeep fusca 147 K
```

Então, apesar de ainda podermos expandir faixas de elementos, nós não podemos contar com pressupostos como: *"o primeiro elemento foi o primeiro a entrar no vetor"* ou *"o último foi o último a ser definido"* – isso *não é verdade* e precisa ser lembrado, principalmente, quando o vetor associativo é populado de forma não-interativa, por exemplo:

```
# Definindo o vetor ‘arr’ como associativo...
declare -A arr

# Populando seus elementos em um loop ‘for’...
for i in {a..e}; do
      arr[$i]=$i
done
```

Aqui, ao fim dos ciclos, os elementos de `arr` serão:

```
1º ciclo: arr[a] → a
2º ciclo: arr[b] → b
3º ciclo: arr[c] → c
4º ciclo: arr[d] → d
5º ciclo: arr[e] → e
```

Mas sua ordem interna pode ser:

```
:~$ echo ${arr[@]}
e d c b a
```

Se tentarmos expandir o último elemento do vetor imaginando que ele é o mesmo que foi definido no último ciclo, teremos problemas:

```
:~$ echo ${arr[@]: -1}
a
```

Tirando esse detalhe, que nem tem relação com as expansões, não há nenhuma diferença na expansão de faixas de elementos de vetores indexados e associativos.

Os parâmetros especiais `@` e `*`, que expandem todos os parâmetros posicionais, comportam-se exatamente como os vetores.

4.9.3 – Remoções e substituições a partir de padrões

Se quisermos expandir uma substring a partir do início ou do fim do valor em um parâmetro, nós podemos utilizar *padrões* em vez das *posições* de seus caracteres, como visto nos tópicos anteriores.

Aqui, os *padrões* seguem os mesmos princípios da *formação de nomes de arquivos*, inclusive os *globs estendidos* (*tópico 4.8.6*), e nós só temos que definir se queremos casá-lo a partir do início ou do fim da string:

```
${nome#padrão} → Casa o padrão a partir do início
${nome%padrão} → Casa o padrão a partir do fim
```

Por exemplo, digamos que a string seja `banana da terra` e o padrão seja `?a` (um caractere qualquer seguido de `a`). Se o casamento for feito com o começo da string, a correspondência será `ba`, mas, se for com o final, será `ra`. Em ambos os casos, o padrão será removido da string na expansão:

Exemplo 4.34 – Removendo padrões casados na string.

```
# Definindo a string...
:~$ fruta='banana da terra'

# Expandindo tudo menos o padrão casado no início...
:~$ echo ${fruta#?a}
nana da terra → banana da terra

# Expandindo tudo menos o padrão casado no final...
:~$ echo ${fruta%?a}
banana da ter → banana da terra
```

> ***Dica:*** *pense no padrão como aquilo que você quer 'remover'!*

Neste tipo de expansão, o casamento do padrão é feito com a menor correspondência possível. Por exemplo, se o padrão fosse `*n` (qualquer

quantidade de caracteres e um `n`) e nós quiséssemos casá-lo com o início da string, veja o que aconteceria:

```
# String...
:~$ fruta='banana da terra'

# Expandindo tudo menos o padrão casado no início...
:~$ echo ${fruta#*n}
ana da terra → bananana da terra
```

Repare que `*n` casou apenas com os primeiros caracteres antes de um dos caracteres `n` existentes na string e a sequência `ban` foi removida.

O mesmo aconteceria se o padrão fosse `n*` (`n` seguido de qualquer quantidade de caracteres) e o casamento fosse com o final da string:

```
# String...
:~$ fruta='banana da terra'

# Expandindo tudo menos o padrão casado no final...
:~$ echo ${fruta%n*}
bana → banana da terra
```

Como podemos ver, ainda havia um `n` que poderia ser casado.

Para que o casamento aconteça com a maior quantidade possível de caracteres, nós dobramos os símbolos – `##`, para remover o padrão do início e `%%` para remover do final.

> ***Outra dica:*** *nunca se esqueça de que o padrão deve representar a sequência de caracteres que você quer remover da string original!*

Exemplo 4.35 – Removendo o máximo de caracteres casados com o padrão.

```
# Casamento máximo no início...
:~$ echo ${fruta##*n}
```

```
a da terra → banana da terra

# Casamento máximo no final...
:~$ echo ${fruta%%n*}
ba → banana da terra
```

Para que você entenda um detalhe importante sobre as expansões que removem e substituem padrões em strings, nós não podemos nos esquecer de que os parâmetros posicionais, obviamente, também são parâmetros. Portanto, tudo que vimos sobre expansões de parâmetros também se aplicam aos parâmetros especiais, inclusive `@` e `*`.

Ou seja, se o seu script receber como parâmetros `banana` e `laranja`, nós podemos expandi-los, todos de uma vez, com `$@` ou `$*` e aplicar os conceitos que estamos vendo neste tópico. Mas, como podemos exemplificar isso sem criarmos um script?

É aí que entra o comando interno `set`!

4.9.4 – Uma pausa para o comando 'set'

O comando `set` é utilizado para definir ou remover os valores das opções de execução do shell, mas também pode definir e modificar valores de parâmetros posicionais – inclusive no modo interativo!

> *O comando* `help set` *pode fornecer informações adicionais, caso você esteja curioso, mas nós vamos nos concentrar apenas na sua capacidade de manipular os parâmetros posicionais.*

Existe um argumento (talvez o mais correto fosse chamar de *operador*) do comando `set` que é o `--` (dois traços). Se algo estiver *antes* dele, será tratado como uma lista de argumentos, mas, tudo que vier *depois* será tratado como

parâmetro posicional da sessão corrente do shell. Se não houver nada após o `--`, os parâmetros posicionais da sessão corrente do shell serão destruídos.

> *Se, em algum momento, eu disse que os padrões posicionais eram de controle exclusivo do shell e que nós não poderíamos manipulá-los, eu menti – é totalmente possível fazer isso com o comando* `set --`*.*

Por exemplo:

Exemplo 4.36 – Manipulando parâmetros posicionais do shell corrente.

```
# Eliminando qualquer parâmetro posicional anterior...
:~$ set --
:~$ echo $@

:~$

# Definindo dois parâmetros posicionais...
:~$ set -- banana laranja
:~$ echo $1
banana
:~$ echo $2
laranja
echo $@
banana laranja

# Alterando os parâmetros posicionais...
:~$ set -- abacate
:~$ echo $@
abacate
```

Era exatamente disso que nós precisávamos para os exemplos dos próximos detalhes sobre as expansões de parâmetros!

4.9.5 – Voltando às remoções e substituições

Quando uma expansão de remoção ou substituição com base em padrões é feita com os nomes dos parâmetros especiais `@` e `*`, ela é aplicada individualmente a cada um dos parâmetros posicionais.

Por exemplo:

Exemplo 4.37 – Removendo o padrão do início de cada parâmetro posicional.

```
# Definindo os parâmetros posicionais da sessão corrente...
:~$ set -- abano banana dezena
:~$ echo $@
abano banana dezena

# Removendo o padrão '*n' da expansão de cada parâmetro...
echo ${@#*n}
o ana a → ~~abano~~ ~~ban~~ana ~~dezena~~
```

Os padrões também podem ser utilizados para substituir caracteres na expansão de parâmetros, e este é o modificador utilizado:

```
${nome/padrão/string} ← Troca o primeiro maior casamento
                        de padrão por string.
```

Por exemplo:

Exemplo 4.38 – Substituindo um padrão por uma string.

```
:~$ str=banana
:~$ echo ${str/na/nda}
bandana → ba~~na~~[nda]na
```

Repare que apenas o primeiro `na` foi substituído, apesar de haver dois em `banana`.

Se o padrão for precedido de `/`, todas as suas ocorrências encontradas serão substituídas:

Exemplo 4.39 – Substituindo todas as ocorrências do padrão string.

```
:~$ str=banana
:~$ echo ${str//na/ta}
batata → ba~~na~~[ta]~~na~~[ta]
```

Mas, observe que a regra do *maior casamento* prevalece neste caso:

```
:~$ str=banana
:~$ echo ${str//n*/ta}
bata → ba~~nana~~[ta]
```

Se o padrão for precedido da cerquilha (`#`), ele tem que casar com o início do valor expandido. É como na expansão com remoção utilizando duas cerquilhas (`##`), só que, em vez de ser removido, o casamento será substituído:

Exemplo 4.40 – Substituindo um padrão no início da string.

```
:~$ str=banana
:~$ echo ${str/#b??/cab}
cabana → ~~ban~~[cab]ana
```

Para fazer o mesmo com o final do valor expandido, nós utilizamos o `%`:

Exemplo 4.41 – Substituindo um padrão no final da string.

```
:~$ str=banana
:~$ echo ${str/%a??/ido}
banido → ban~~ana~~[ido]
```

> ***Para reforçar:*** *se o parâmetro for `@` ou `*`, as expansões de remoção e substituição serão aplicadas a cada um dos parâmetros posicionais, se for um vetor com `@` ou `*` no subscrito, serão aplicadas a cada um dos elementos.*

4.9.6 – Expandindo valores condicionalmente

O shell realiza quatro tipos de expansões condicionadas ao estado do parâmetro – se ele está ou não definido. É com elas que nós podemos dizer ao shell o que fazer caso uma variável esteja indefinida ou tenha um valor nulo (string vazia).

O primeiro tipo de expansão condicional (e o mais conhecido) é aquele com que nós definimos o valor padrão de um parâmetro. Caso o parâmetro seja nulo ou não esteja definido, o Bash fará a expansão de um valor padrão:

`${nome:-valor}`
Expande `valor` se `nome` for nulo ou não definido.

`${nome-valor}`
Expande `valor` *apenas* se `nome` for não definido.

Por exemplo, se o parâmetro `var` for indefinido, ambas as expansões resultarão na string `padrão`:

Exemplo 4.42 – Expandindo um valor para um parâmetro não definido.

```
:~$ unset var
:~$ echo ${var:-padrão}
padrão
:~$ echo ${var-padrão}
padrão
```

Mas, estando o parâmetro definido, se o seu valor for nulo, somente a expansão com `:` resultará na string `padrão`:

Exemplo 4.43 – Expandindo um valor padrão para um parâmetro nulo.

```
:~$ var=
:~$ echo ${var:-padrão}
padrão
:~$ echo ${var-padrão}
```

```
:~$
```

***Note:** esta expansão **não altera** o valor original do parâmetro!*

Se o parâmetro contiver algum valor, este é que será expandido (com ou sem os dois pontos):

```
:~$ var=banana
:~$ echo ${var:-padrão}
banana
:~$ echo ${var-padrão}
banana
```

Opcionalmente, em vez de apenas expandirmos, nós podemos atribuir um valor padrão ao parâmetro não definido ou com valor nulo.

`${nome:=valor}`
Expande `valor` e o atribui a `nome` se `nome` for nulo ou não definido.

`${nome=valor}`
Expande `valor` e o atribui a `nome` *apenas* se `nome` for não definido.

Observe os exemplos:

Exemplo 4.44 – Expandindo e atribuindo um valor padrão.

```
:~$ unset var ← Tornando variável não definida.
:~$ echo $var ← Conferindo...
              ← Não expande nada.

:~$ echo ${var=banana} ← Sem dois pontos...
banana                 ← Expandiu!
:~$ echo $var.
banana                 ← Atribuiu!

:~$ unset var ← Tornando não definida novamente.
```

```
:~$ echo ${var:=banana} ← Com dois pontos...
banana                  ← Expandiu!
:~$ echo $var
banana                  ← Atribuiu!

:~$ unset var ← Tornando não definida novamente.
:~$ var=      ← Atribuindo um valor nulo.

:~$ echo ${var=banana} ← Sem dois pontos...
                       ← Não expandiu!
:~$ echo $var
                       ← Não atribuiu!
:~$

:~$ echo ${var:=banana} ← Com dois pontos...
banana                  ← Expandiu!
:~$ echo $var
banana                  ← Atribuiu!
```

> ***Note:*** *esta expansão* ***altera*** *o valor original (nulo ou não definido) do parâmetro!*

Uma terceira expansão condicional (muito interessante) pode ser feita quando valor do parâmetro *não é* nulo nem indefinido.

`${nome:+valor}`
Expande `valor` apenas se `nome` *não for* nulo.

`${nome+valor}`
Expande `valor` se `nome` *não for* nulo ou não definido.

Observe que agora a opção com `:` *não expandirá* um valor padrão se o parâmetro contiver um valor nulo. Observe nos exemplos:

Exemplo 4.45 – Expandir valor se parâmetro não for nulo nem indefinido.

```
:~$ unset var ← Parâmetro 'var' não definido.

:~$ echo ${var:+banana} ← Com dois pontos.
                        ← Não expande.
:~$ echo ${var+banana}  ← Sem dois pontos.
                        ← Não expande.

:~$ var=                ← Atribuindo um valor nulo.
:~$ echo ${var:+banana} ← Com dois pontos.
                        ← Não expande.
:~$ echo ${var+banana}  ← Sem dois pontos.
banana                  ← Expande!

:~$ var=laranja         ← Atribuindo um valor não-nulo.
:~$ echo ${var:+banana} ← Com dois pontos.
banana                  ← Expande!
:~$ echo ${var+banana}  ← Sem dois pontos.
banana                  ← Expande!
```

> ***Note:** esta expansão **não altera** o valor original do parâmetro!*

A última expansão condicional nos permite gerar um erro se o valor do parâmetro for nulo ou não definido.

`${nome:?mensagem}`
Envia `mensagem` para a saída de erros se `nome` for nulo ou não definido.

`${nome?mensagem}`
Envia `mensagem` para a saída de erros *apenas* se `nome` for não definido.

Exemplo 4.46 – Mensagem de erro se o parâmetro for nulo ou indefinido.

```
:~$ unset var ← Parâmetro 'var' não definido.

:~$ echo ${var?deu erro}  ← Sem dois pontos.
bash: var: deu erro       ← Mensagem de erro!
:~$ echo ${var:?deu erro} ← Com dois pontos.
```

```
bash: var: deu erro       ← Mensagem de erro!

:~$ var= ← Parâmetro 'var' agora é nulo.

:~$ echo ${var?deu erro}  ← Sem dois pontos.
                          ← Nenhum erro!
:~$ echo ${var:?deu erro} ← Com dois pontos.
bash: var: deu erro       ← Mensagem de erro!
```

As expansões condicionais (como todas as expansões) são um grande exemplo de como o Bash pode nos ajudar a transformar várias linhas de código e estruturas lógicas complicadas em expansões e comandos simples e objetivos.

> *Há quem ache que esse tipo de recurso reduz a legibilidade do código. Mas, como eu sempre digo: "legibilidade é para quem sabe ler a linguagem em que o código foi escrito".*

4.9.7 – Alterando a caixa do texto

Nós também podemos expandir parâmetros alterando a caixa de texto de seus caracteres, tornando-os maiúsculos ou minúsculos – o que pode ser particularmente útil, por exemplo, quando temos que comparar valores recebidos do usuário com um padrão, um caractere ou uma string.

```
${nome^}  → Converte primeiro caractere para caixa alta.
${nome^^} → Converte todos os caracteres para caixa alta.

${nome,}  → Converte primeiro caractere para caixa baixa.
${nome,,} → Converte todos os caracteres para caixa baixa.

${nome~}  → Inverte a caixa do primeiro caractere.
${nome~~} → Inverte a caixa de todos os caracteres.
```

> *Este último modificador da caixa de texto (`~` e `~~`) não está documentado e eu só soube da existência dele graças ao amigo 'bashsita' Meleu, que o descobriu vasculhando o código fonte do Bash.*

O uso dessas expansões é muito simples, veja nos exemplos:

Exemplo 4.47 – Alterando a caixa de texto dos valores expandidos.

```
# Expandindo em caixa alta...
:~$ bicho=zebra
:~$ echo ${bicho^}
Zebra
:~$ echo ${bicho^^}
ZEBRA

# Expandindo em caixa baixa...
:~$ fruta=BANANA
:~$ echo ${fruta,}
bANANA
:~$ echo ${fruta,,}
banana

# Expandindo com caixa invertida
:~$ nome=MaRiA
:~$ echo ${nome~}
maRiA
:~$ echo ${nome~~}
mArIa
```

Opcionalmente, nós podemos definir um padrão que corresponda aos caracteres que queremos modificar, mas isso só tem efeito prático com os modificadores dobrados, que são os que afetam toda a string:

Exemplo 4.48 – Alterando a caixa de caracteres segundo um padrão.

```
:~$ str='casa da sogra'
:~$ echo ${str^^[cs]}
CaSa da Sogra
```

```
:~$ str='LÍNGUA DE TRAPO'
:~$ echo ${str,,[LDT]}
lÍNGUA dE tRAPO
```

Um detalhe que não tem a ver diretamente com as expansões, mas é interessante mencionar, é que existem formas de alterar a caixa de texto do valor de um parâmetro *no momento da atribuição* com o comando `declare`:

Exemplo 4.49 – Alterando a caixa do valor de um parametro.

```
:~$ declare -l str  ← Define que 'str' terá caixa baixa.
:~$ str=BOTECO
:~$ echo $str
boteco

:~$ declare -u str  ← Define que 'str' terá caixa alta.
:~$ str=música
:~$ echo $str
MÚSICA
```

Como sempre, se o parâmetro for `@` ou `*`, as expansões serão aplicadas a cada um dos parâmetros posicionais, se for um vetor com `@` ou `*` no subscrito, serão aplicadas a cada um dos elementos.

4.9.8 – Outras expansões de parâmetros

Para não deixarmos pontas soltas neste assunto, vamos fechar este tópico falando de algumas expansões de parâmetros um pouco menos conhecidas, mas que também merecem o nosso carinho.

Expansão de prefixos: `${!prefixo*}` ou `${!prefixo@}`

Expande todos os nomes das variáveis que começam com `prefixo` separados pelo primeiro caractere definido como separador padrão de palavras em `IFS`.

A diferença entre o `*` e o `@` está na forma como as palavras expandidas serão tratadas: se for utilizado o `@` e a expansão estiver entre aspas duplas, cada nome expandido será tratado como uma palavra.

Exemplo:

Exemplo 4.50 – Expansão de prefixos.

```
:~$ echo ${!P*}
PATH PIPESTATUS PPID PROMPT_COMMAND PS1 PS2 PS4 PWD
```

Expansão do operador 'Q' (aspas): `${nome@Q}`

Expande o valor do parâmetro entre aspas.

Exemplo:

Exemplo 4.51 – Expansão do valor do parâmetro entre aspas.

```
:~$ var=carroça
:~$ echo ${var@Q}
'carroça'
```

Expansão do operador 'E' (escape): `${nome@E}`

Tem o mesmo efeito da expansão de caracteres ANSI-C (`$'…'`).

Exemplo:

Exemplo 4.52 – Expansão de strings contendo caracteres de controle.

```
:~$ nl='\n'
:~$ echo ${nl@E}  ← Expande a quebra de linha (\n).

:~$
```

Expansão do operador 'P' (prompt): `${nome@P}`

Expande os caracteres de comando do prompt (*tópico 1.2.6*).

Exemplo:

Exemplo 4.53 – Expansão dos caracteres de comando do prompt.

```
:~$ prompt='\u@\h:\w\$ '
:~$ echo ${prompt@P}
blau@enterprise:~$
:~$
```

Expansão do operador 'a' (atributos): `${nome@a}`

Expande os atributos de um parâmetro.

Exemplo:

Exemplo 4.54 – Expansão dos atributos de parâmetros.

```
:~$ declare -i num  ← Define o atributo de inteiro.
:~$ echo ${num@a}
i
```

Expansão do operador 'A' (atribuição): `${nome@A}`

Expande uma string na forma de uma atribuição que, se executada, recria o parâmetro com seu valor e seus atributos.

Exemplo:

Exemplo 4.55 – Expansão de uma operação de atribuição.

```
:~$ fruta=laranja
:~$ echo ${fruta@A}
fruta='laranja'
```

4.10 – Substituição de comandos

A *substituição de comandos* é um mecanismo que permite a execução de um comando em um *subshell* (*tópico 3.6.3*) para obter sua saída na forma de uma *expansão*. No Bash, a substituição acontece quando prefixamos um agrupamento de comandos entre parêntesis com um cifrão:

```
$(comandos)
```

> *Existe uma outra sintaxe, mais antiga, limitada e considerada obsoleta por alguns, onde os comandos são circundados por um par de acentos graves:* `` `comandos` ``*, mas nós falaremos sobre ela mais adiante.*

4.10.1 – Armazenando e expandindo a saída de comandos

Quando a expansão acontece, todos os parâmetros da sessão corrente do shell são copiados para um subshell, o que nos permite expandi-los dentro do agrupamento com parêntesis, e a *saída padrão* do subshell é tratada como um valor passível de ser armazenado em uma variável:

Exemplo 4.56 – Armazenando a expansão de uma substituição de comandos.

```
:~$ var=$(grep $USER /etc/passwd)
:~$ echo $var
:~$ blau:x:1000:1000:Blau Araujo,,,:/home/blau:/bin/bash
```

Como a substituição de comandos expressa um valor, em vez de ser armazenada, ela também pode ser expandida diretamente – da mesma forma que fazemos com parâmetros:

Exemplo 4.57 – Expandindo uma substituição de comandos.

```
:~$ echo O usuário se chama $(echo Blau Araujo).
:~$ O usuário se chama Blau Araujo.
```

4.10.2 – Cuidado com o escopo das variáveis

O ponto mais importante sobre o uso de substituições de comandos é o cuidado com o escopo das variáveis. Não podemos nos esquecer de que o ambiente da sessão que iniciou a substituição é copiado para o subshell, ou seja, nós temos a impressão de que as variáveis estão disponíveis na substituição de comandos, mas *são apenas cópias com os mesmos nomes*.

Observe este exemplo:

```
:~$ echo PID mãe: $$ - PID subshell: $(echo $$)
PID mãe: 11365 - PID subshell: 11365
```

Como o PID da sessão mãe (e todo o restante do ambiente) foi copiado, o parâmetro especial `$` do subshell tem o mesmo PID da sessão mãe, o que pode nos levar a imaginar equivocadamente que se trata de uma mesma sessão, e não em um subshell (que teria seu próprio PID). Com a variável `BASHPID`, porém, que sempre é gerada no momento em que uma sessão do Bash é iniciada, seja ela um subshell ou não, nós temos como ver a diferença:

```
:~$ echo PID mãe: $BASHPID - PID subshell: $(echo $BASHPID)
PID mãe: 11365 - PID subshell: 22155
```

Consequentemente, as alterações feitas na cópia de uma variável não se refletem nos valores originais:

```
:~$ bicho=gato
:~$ echo $(echo $bicho; bicho=zebra; echo $bicho)
gato zebra
```

```
:~$ echo $bicho
gato
```

Aliás, aproveitando o exemplo, repare que a substituição de comandos produziu duas saídas a partir de dois comandos `echo`.

4.10.3 – Saída em múltiplas linhas

Sabendo que o comando `echo` inclui uma quebra de linha na string enviada para a saída padrão, é interessante notar que elas (as quebras) foram colapsadas em um espaço no processo de separação de palavras do shell. Para que fossem preservadas, nós teríamos que envolver a substituição de comandos com aspas duplas:

Exemplo 4.58 – Expandindo saídas com múltiplas linhas.

```
:~$ bicho=gato
:~$ echo "$(echo $bicho; bicho=zebra; echo $bicho)"
gato
zebra
```

Mas, como a troca das quebras de linha só acontecem no processo de separação de palavras, antes de serem exibidas, elas ainda estão lá, o que podemos comprovar, por exemplo, armazenando a expansão em uma variável:

```
:~$ bicho=gato
:~$ var=$(echo $bicho; bicho=zebra; echo $bicho)
:~$ echo $var
gato zebra
:~$ echo "$var"
gato
zebra
```

4.10.4 – Expandindo o conteúdo de arquivos

Esta capacidade da substituição de comandos de expandir valores com múltiplas linhas a torna muito útil para a visualização do conteúdo de arquivos, inclusive como uma alternativa mais rápida para o utilitário `cat`:

Exemplo 4.59 – Um 'cat' em Bash puro.

```
:~$ echo "$(< compras.txt)"
leite
pão
ovos
banana
laranja
```

No exemplo, `$(< arquivo)` se parece com um *redirecionamento* e se parece com uma substituição de comandos e se funciona como uma expansão, mas não é nada disso!

4.10.5 – Comprando 'cat' por lebre

Na verdade, trata-se de uma implementação em Bash de um operador especial do *Korn Shell* (*ksh*) para a leitura do conteúdo de arquivos. A semelhança com uma substituição de comandos, e o fato de ser apresentado como tal no manual e no código fonte do Bash, é de dar um nó na cabeça de mentes mais atentas e curiosas, pois, se fosse realmente um redirecionamento, o conteúdo do arquivo estaria sendo enviado para a *entrada padrão* (*stdin*) e nós não veríamos nada:

```
:~$ < /etc/passwd
:~$
```

Logo, também não haveria nada na *saída* da substituição de comandos para ser expandido.

4.10.6 – A sintaxe antiga

Sobre a sintaxe `` `comando` ``, ela é considerada limitada e obsoleta, e há bons motivos para isso. Para começar, a forma `$(comando)` é POSIX, é mais legível e fácil de identificar no código e permite aninhamentos com muita facilidade. Mas, o principal problema da sintaxe antiga é o *inferno* das aspas e do escape de caracteres.

Na forma antiga, a barra invertida tem valor literal, a menos que venha seguida do cifrão (`$`), do acento grave (`` ` ``) ou de outra barra invertida (`\`). Ou seja, ao estabelecer as próprias regras para lidar com as aspas e o caractere de escape, a forma antiga introduz uma complexidade totalmente desnecessária oferecendo muito pouco em troca.

> *É ótimo que você saiba que* `` `comando` `` *também é uma forma válida de substituição de comandos, mas só para entender o código dos outros. Fora isso, devemos evitar seu uso.*

4.11 – Expansões aritméticas

Uma *expansão aritmética* é o mecanismo pelo qual o Bash substitui a avaliação de uma expressão aritmética com o comando composto `((expressão))` pelo valor resultante. Para que isso aconteça, o comando composto deve ser prefixado pelo cifrão:

```
$((expressão))
```

Assim, o resultado da expansão pode ser exibido ou armazenado em uma variável, exatamente como vimos na *substituição de comandos* – a diferença, porém, é que a expansão aritmética não inicia um *subshell*.

Apenas com o comando composto `((…))`, o valor da expressão não é enviado para a saída padrão e só pode ser capturado se atribuído a uma variável:

```
:~$ ((2 + 2))
:~$
:~$ ((a = 2 + 2))
:~$ echo $a
4
```

> ***Importante!*** *Não confunda o 'estado de saída' do comando composto* `((…))` *com o valor da 'avaliação da expressão'. O estado de saída será sempre* `1` *(erro) se a expressão resultar no valor zero, e será* `0` *(sucesso) se a avaliação resultar em qualquer valor diferente de zero.*

Com a expansão aritmética nós temos acesso ao valor da expressão:

Exemplo 4.60 – Expandindo o valor de uma expressão aritmética.

```
:~$ echo $((2 + 2))
4

:~$ echo O resultado da multiplicação é $((4 * 5)).
O resultado da multiplicação é 20.

:~$ div=$((15 / 3))
:~$ echo 15 / 3 = $div
15 / 3 = 5
```

> *A expansão em si é muito simples e há pouco a ser dito sobre ela, mas nós a utilizaremos bastante quando falarmos de operações aritméticas no Bash.*

4.12 – Substituição de processos

Uma expansão pouco documentada é a *substituição de processos*, cuja finalidade básica é fazer com que uma lista de comandos seja tratada como um arquivo:

```
<(lista) → Substitui um arquivo para leitura.
>(lista) → Substitui um arquivo para escrita.
```

> ***Importante!*** *Não há espaços entre os símbolos* `<` *e* `>` *e o restante da expansão! Apesar da semelhança, essa expansão não tem (quase) nada a ver com redirecionamentos.*

A expansão é feita trocando a ocorrência da substituição de processos pelo nome de um arquivo criado pelo Bash para uso temporário. Com isso, nós podemos utilizar as substituições de processos em comandos e programas que exigem nomes de arquivos como argumentos, tanto para a leitura quanto para a escrita de dados.

A lista de comandos geralmente é formada pelo encadeamento de comandos através de *pipes* (`|`), que é quando a substituição de processos mostra todo o seu poder.

> *Nós falaremos de pipes e redirecionamentos nos próximos capítulos, mas, de forma resumida, quando uma lista de comandos está encadeada por pipes, cada comando após o operador* `|` *é executado em um subshell, utilizando como entrada de dados a saída do comando anterior.*

Infelizmente, nem todos os sistemas têm o shell configurado para permitir esse tipo de expansão, não é um recurso compatível com as normas POSIX, e só funcionam no Bash se a opção `posix` estiver desabilitada:

```
set -o posix → Ativa e restringe alguns recursos do Bash.
set +o posix → Desativa o modo POSIX.
```

Na versão 10 do Debian GNU/Linux, o modo POSIX do Bash vem *desativado* por padrão nos modos interativo e não-interativo, o que permite o uso de substituições de processos:

Exemplo 4.61 – Verificando a configuração do modo POSIX do Bash.

```
:~$ shopt -o posix
posix           off
:~$ bash -c 'shopt -o posix'
posix           off
```

Um experimento interessante é tentar descobrir o nome do arquivo que o shell cria para expandir uma substituição de processos, observe:

```
# Expandindo um nome de arquivo para leitura.
:~$ echo <(:)
/dev/fd/63

# Expandindo um nome de arquivo para escrita.
:~$ echo >(:)
/dev/fd/63
```

O comando nulo (`:`) equivale ao nosso velho conhecido comando `true`: ele não faz nada e encerra com estado de saída de sucesso. Ele só está no exemplo para que possamos fazer uma substituição de processos e forçar o shell a expandir o nome do arquivo que ele vai utilizar – no caso `/dev/fd/63`.

Como vimos no *tópico 1.9.5*, o diretório `/dev/fd` contém os chamados *descritores de arquivos* (entre eles, os descritores de arquivos `0`, `1` e `2`, que recebem os fluxos de entrada e saída de dados: *stdin*, *stdout* e *stderr*). Portanto, o nosso experimento demonstrou que o shell criou um descritor de arquivo de nome `63`, tanto para o nome de arquivo de leitura quanto para o de escrita.

Caso ainda não esteja claro, quando um nome de arquivo é expandido *para leitura*, ele pode ser utilizado como *fonte de dados* para um comando ou um programa, enquanto que um nome de arquivo expandido *para escrita*, será utilizado para *receber os dados* enviados por um comando ou do programa.

Isso quer dizer que, se o comando esperar mais de um arquivo nos seus argumentos (o que é muito comum, por exemplo, em programas de conversão de mídia, onde temos que informar o arquivo original e o arquivo de destino), nós podemos utilizar várias substituições de processos:

```
# Expandindo nomes de arquivos para leitura e escrita.
:~$ echo <(:) >(:)
/dev/fd/63 /dev/fd/62
```

Note que o nome inicial (o número) continuou o mesmo: o arquivo `63`, mas a numeração do nome seguinte caiu para `62`, e continuaria caindo se expandíssemos mais nomes.

> ***Importantíssimo!*** *A substituição de processos em si não tem nada de complicado – ela apenas gera um nome de arquivo e pronto. O que realmente causa muita confusão são os símbolos* `<` *e* `>` *da sua sintaxe, porque eles quase sempre causam a impressão de serem operadores de redirecionamento. Eles não são! Não existe uma relação entre os nomes de arquivos expandidos, eles não trocam dados entre si (a menos que o comando de quem eles são argumentos faça isso), eles são apenas 'parâmetros de um comando'.*

Por falar em redirecionamentos...

5 – Fluxos de dados e redirecionamentos

Após algumas referências em outros capítulos, chegou o momento de conhecermos um pouco melhor os mecanismos pelos quais o Bash nos permite trabalhar com fluxos de entrada e saída de dados, descritores de arquivos e as formas de passar os dados que foram processados em um comando para outros comandos.

Neste *pequeno manual*, a ideia não é esgotar o assunto, mas apresentar os principais conceitos e mecanismos destes recursos que, em última análise, estão na essência da própria *Filosofia Unix*. O tópico em si não é nada do outro mundo – muito pelo contrário, é tudo muito simples. Mas são recursos tão flexíveis, no que tange às suas aplicações práticas, que a tentativa de abordar todas as suas nuances e peculiaridades em apenas um capítulo seria um esforço quase inglório.

5.1 – Fluxos de entrada e saída de dados

Quase todos os comandos que executamos no shell precisará receber informações sobre como deverá ser executado, os dados que deve processar, e produzirá algum tipo de informação a ser enviada para exibição no terminal ou para registro em um arquivo.

No sentido da *recepção*, que nós chamamos de *entrada* (ou *leitura*) *de dados*, as informações passadas para um comando ou um programa de várias formas: pelos argumentos da linha do comando (registrados nos *parâmetros posicionais*), através das variáveis de ambiente herdadas do processo mãe, de arquivos, e de tudo que possa ser apontado por um *descritor de arquivos*.

No sentido oposto, do *envio*, que nós também podemos chamar de *saída* (ou *escrita*) *de dados*, as informações produzidas por um comando ou um

programa podem ir parar em muitos lugares diferentes: um arquivo, uma variável de ambiente, o terminal, ou em todo lugar para o qual um *descritor de arquivos* possa apontar.

Como você pode ver, os descritores de arquivos têm um papel-chave nos processos que envolvem os fluxos de entrada e saída de dados, e é sobre isso que precisamos conversar.

5.2 – Os descritores de arquivos

Quando um comando ou um arquivo precisa lidar com arquivos, isso nunca é feito diretamente – os arquivos em si, são dados registrados em algum dispositivo de *hardware*, e isto pertence ao domínio das funções do *kernel*. O que é feito, em vez disso, é a solicitação de um *recurso* que funcionará como um ponteiro para as fontes de dados ou um local onde os dados deverão ser escritos. Dependendo do contexto, este *recurso* pode ser chamado de *ponteiro de arquivo*, *manipulador* (ou *handler*). Em sistemas *unix-like*, onde tudo tem uma representação na forma de um arquivo, inclusive o hardware, estes recursos são os *descritores de arquivos* (em inglês, *"file descriptors"* ou *"FD"*).

> *Sim, existem diferenças entre os conceitos por detrás desses nomes, mas todos eles têm isso em comum: são 'recursos' disponibilizados pelo sistema operacional para que os fluxos de dados sejam manipulados por um programa.*

Os *descritores de arquivos*, portanto, são referências a arquivos ou a outros recursos que atuam como arquivos, como os *pipes*, dispositivos de hardware, e até os *fluxos de dados*!

5.3 – Os fluxos padrão (stdin, stdout e stderr)

Por padrão, todo novo processo, seja o shell ou outro programa qualquer, é iniciado com acesso a três descritores de arquivos:

- Entrada padrão (stdin): descritor de arquivos 0;
- Saída padrão (stdout): descritor de arquivos 1;
- Saída padrão de erros (stderr): descritor de arquivos 2.

> *Dependendo do que estivermos fazendo, os fluxos padrão podem ser referenciados pela sua numeração POSIX, pelos seus nomes ou, dependendo do operador em uso, podem até ser omitidos.*

Se estivermos trabalhando no terminal, a *entrada padrão* (stdin) estará associada aos dados digitados nos nossos teclados. No sentido oposto, tudo que é exibido no terminal vem do fluxo de dados na *saída padrão* (stdout).

As mensagens de erro, por sua vez, são enviadas para um descritor de arquivos especialmente disponibilizado para esse tipo de fluxo de dados, a *saída padrão de erros* (stderr), que geralmente é replicada na *saída padrão* para que possamos ver o que deu errado.

> *Programas para a interface gráfica (GUI) também têm acesso a esses descritores de arquivo, mas geralmente só fazem uso da saída padrão de erros (stderr).*

5.4 – Lendo a entrada padrão

Até aqui, nós só vimos a passagem de dados para um programa ou um comando através de *parâmetros posicionais*, mas isso também pode ser feito com a leitura dos dados em *stdin*, por exemplo, com o comando interno `read`:

Exemplo 5.1 – Lendo 'stdin' com o comando 'read'.

```
:~$ read -p 'Digite seu nome: ' nome
Digite seu nome: Blau Araujo
:~$ echo $nome
Blau Araujo
```

Quando executado, o comando `read` lê os dados digitados (ou seja, em *stdin*) e os armazena em uma variável – que pode ser uma ou mais variáveis definidas por nós ou, se nenhuma for especificada, a variável especial `REPLY`. No exemplo, nós definimos que os dados lidos de *stdin* seriam armazenados na variável `nome`. Antes disso, com a opção `-p`, nós especificamos uma string que seria exibida como um prompt.

```
read -p 'Digite seu nome: ' nome
     └──────────┬──────────┘  ↑
              prompt       variável
```

Da forma que o comando foi montado, enquanto eu digitava o meu nome, todos os caracteres eram enviados para a entrada padrão até que o caractere da quebra de linha fosse lido, o que aconteceu quando eu teclei `Enter`. Porém, com a opção `-d`, nós poderíamos determinar o fim da leitura assim que o caractere da nossa escolha fosse digitado.

Sem a especificação da variável `nome`, todos os caracteres lidos iriam para a variável especial `REPLY`:

```
:~$ read -p 'Digite seu nome: '
Digite seu nome: Blau Araujo
:~$ echo $REPLY
Blau Araujo
```

Se mais de uma variável de destino fosse definida, cada uma delas se comportaria mais ou menos como um *parâmetro posicional nomeado:*

```
# Duas variáveis e duas palavras...
:~$ read -p 'Digite seu nome: ' nome sobrenome
Digite seu nome: Blau Araujo
:~$ echo $nome
Blau
:~$ echo $sobrenome
Araujo

# Duas variáveis e três palavras...
Digite seu nome: João Paulo Teixeira
:~$ echo $nome
João
:~$ echo $sobrenome
Paulo Teixeira

# Aspas são tratadas literalmente...
Digite seu nome: 'João Paulo' Teixeira
:~$ echo $nome
'João
:~$ echo $sobrenome
Paulo' Teixeira

# Mas podemos escapar espaços...
 Digite seu nome: João\ Paulo Teixeira
:~$ echo $nome
João Paulo
:~$ echo $sobrenome
Teixeira
```

Alternativamente, a variável pode ser um *vetor indexado*:

```
:~$ read -p 'Digite seu nome: ' -a nome
Digite seu nome: Luis Carlos Lopes
:~$ echo ${nome[0]}
Luis
:~$ echo ${nome[1]}
Carlos
:~$ echo ${nome[2]}
Lopes
:~$ echo ${nome[@]}
Luis Carlos Lopes
```

Mas, em todos esses exemplos, ocorreu uma coisa que nós geralmente deixamos passar despercebidamente: *nós conseguimos ver tudo que foi digitado!* Isso quer dizer que, enquanto o comando estava sendo executado, a *entrada padrão* estava sendo *ecoada* na *saída padrão*.

Claro que estou mencionando isso porque estamos falando de fluxos de entrada e saída de dados, mas também porque existe uma opção do comando `read` que muda esse comportamento – a opção `-s`:

```
:~$ read -s -p 'Digite seu nome: ' nome
Digite seu nome: :~$
:~$ echo $nome
Blau
```

Repare que nem a quebra de linha gerada pelo `Enter` foi ecoada na saída, o que fez com que o prompt do shell fosse exibido na mesma linha. Aliás, sobre a quebra de linha no final, mesmo quando ela é ecoada na saída, ela é descartada da string recebida.

5.5 – Enviando dados para a entrada padrão

O teclado, contudo, não é a única forma fazermos os dados chegarem à entrada padrão. Na verdade, a maior beleza dos fluxos de dados é que eles podem ser desviados com o propósito de chegarem aonde são necessários.

Para isso, o Bash oferece uma série de operadores, entre eles: o operador de *redirecionamento do conteúdo de arquivos para a entrada padrão* (`<`), o operador *here-document* (`<<`) e o operador *here-string* (`<<<`), que nós veremos nos próximos tópicos.

5.5.1 – Redirecionamento de arquivos para stdin

Também conhecido como *redirecionamento de entrada*, o operador < faz com que um arquivo seja aberto para leitura no descritor de arquivos especificado ou, por padrão, no descritor de arquivos `0` (stdin).

A sintaxe geral da operação é:

```
[n]< nome
```

Onde `n` é o número de um descritor de arquivos (opcional) e `nome` é a expansão do nome do arquivo que será lido.

Se `n` não for especificado, o conteúdo do arquivo irá para a *entrada padrão* (*stdin*, descritor de arquivos `0`) – e é isso que nos interessa agora. Como o comando `read` lê a *entrada padrão*, nós podemos tentar enviar o conteúdo de um arquivo para ele.

Tomando como exemplo um arquivo chamado `compras.txt`, vamos ler seu conteúdo com a *substituição de comandos* que aprendemos no capítulo anterior:

```
:~$ echo $(< compras.txt)
Pão
Leite
Ovos
Banana
Laranja
```

Nós já vimos que esta construção não é nem um redirecionamento para a entrada e nem uma substituição de comandos exatamente, então, vamos tentar um redirecionamento de fato desta vez:

```
:~$ read < compras.txt
:~$ echo $REPLY
Pão
```

Funcionou, mas... O que houve com as outras linhas?!

Exatamente, parabéns!

Como cada linha termina com um caractere de quebra de linha, e este é o caractere delimitador padrão do comando, o `read` interrompe a leitura assim que o encontra. Então, para lermos o arquivo inteiro com ele, nós temos a alternativa de fazermos um *loop*. O problema é que nós só conhecemos o *loop* `for`, mas precisamos de um *loop* que tenha seu fim limitado a não haver mais nada para ser lido no arquivo com o comando `read`. Para nossa sorte, porém, isso é facílimo de resolver com o *loop* `while` e aproveitando o fato do comando `read` sair com estado de erro encontra o *fim do arquivo* (*EOF*).

> *O fim do arquivo (EOF) não é um caractere, como muitos pensam, mas um sinal que o sistema operacional emite para o processo indicando o fim de um fluxo de dados.*

O *loop* `while`, que é um *comando composto*, monitora o estado de saída de um dado comando para *"decidir"* se irá ou não executar um bloco de comandos. Vejamos na prática:

Exemplo 5.2 – Lendo um arquivo com o comando 'read' e o loop 'while'.

```
:~$ while read; do echo $REPLY; done < compras.txt
Pão
Leite
Ovos
```

```
Banana
Laranja
```

Um detalhe muito importante fará com que você jamais se confunda com esse tipo de estrutura ou com o que realmente acontece quando usamos o operador `<`.

Tente responder: para onde vai o conteúdo do arquivo?

Para o `done`?

Para o `read`?

Para o `while`?

Para toda a estrutura do *loop*?

Exato! Você está prestando atenção!

O redirecionamento de entrada manda os dados do arquivo para o *descritor de arquivos* `0`, a *entrada padrão*, stdin, e é isso que o comando `read` tenta ler!

Outra coisa interessante é que, a cada ciclo do `while`, o ponteiro interno do descritor de arquivos muda de posição indicando até onde aconteceu a última leitura e, assim, o `read` pode continuar lendo de onde parou até encontrar o fim do arquivo (EOF).

No exemplo, nós usamos o descritor de arquivos `0`, que é o padrão do operador, mas nós poderíamos ter utilizado outros, por exemplo:

```
:~$ while read -u 33; do echo $REPLY; done 33< compras.txt
Pão
Leite
Ovos
Banana
Laranja
```

> *A opção* `-u` *do comando* `read` *serve para especificarmos o descritor de arquivo que será lido.*

O descritor de arquivo `33` não existia antes do redirecionamento solicitar um *recurso* e não existirá depois de terminada a leitura – e o exemplo abaixo demonstra isso:

```
:~$ while read -u 33; echo /dev/fd; done 33< compras.txt
0  1  2  3  33  ← Linha Pão
0  1  2  3  33  ← Linha Leite
0  1  2  3  33  ← Linha Ovos
0  1  2  3  33  ← Linha Banana
0  1  2  3  33  ← Linha Laranja
:~$ ls /dev/fd  ← Leitura terminada, listando /dev/fd...
0  1  2  3      ← FD 33 não existe mais!
```

> ***Atenção!*** *Note que, neste caso, o operador cuidou tanto da abertura quanto do fechamento do recurso, o que pode não acontecer em todas as situações e com todos os operadores.*

Mas... Você não vai me perguntar nada?

Pois é, boa pergunta!

Eu disse que todo processo inicia com os descritores de arquivos `0`, `1` e `2` à disposição, por isso eles estão lá em `/dev/fd`, mas o que é aquele descritor de arquivos `3`?!

Acontece que eu também disse que *nenhum comando ou programa manipula arquivos diretamente*, que eles precisam solicitar um recurso ao sistema operacional. Então, o utilitário `ls`, que faz a listagem de *arquivos*, também precisa de um *recurso*, e o sistema fornece aquele que geralmente é o menor número disponível – no caso, `3`.

Mas você já sabia disso, é claro, só estava querendo me testar, não é?

5.5.2 – Eis o here-document

Com o operador `<<`, chamado de *here-document* (ou *here-doc*, pelos íntimos), nós podemos fazer exatamente a mesma coisa que fizemos com o redirecionamento de entrada, só que utilizando uma string com várias linhas (um *documento*, daí *here-doc*) como fonte de dados – tudo que precisamos fazer é dizer onde começam e onde terminam as linhas.

> *Como os exemplos ficarão mais longos, nós vamos utilizar as nossas convenções para exemplos 'em scripts'.*

Esta é a sintaxe geral de uso de um *here-doc*:

```
[n]<< DELIMITADOR
linha 1
linha 2
…
DELIMITADOR
```

O espaço depois do operador `<<` é opcional, assim como o descritor de arquivo `n`, que receberá as linhas do documento.

> ***Importante!*** *Não pode haver espaços antes do delimitador que faz o fechamento do bloco!*

O delimitador inicial pode ser qualquer palavra com ou sem aspas, e isso é o que vai determinar se haverá ou não expansões no texto do documento. Observe, os exemplos abaixo. Neles, nós utilizaremos o utilitário `cat` para ler o conteúdo na *entrada padrão*:

Exemplo 5.3 – Lendo um 'here-doc' com o utilitário 'cat'.

```
# Sem aspas no delimitador...
```

```
cat << AQUI
linha 1
$USER
AQUI

# A saída será...

linha 1
blau

# Com aspas (simples ou duplas) no delimitador...

cat << "AQUI"
linha 1
$USER
AQUI

# A saída será...

linha 1
$USER
```

Para facilitar a leitura do código em scripts, nós podemos utilizar a variante `<<-` do operador, que remove *tabulações reais* no início das linhas do documento:

```
# No script...

cat <<- AQUI
      linha 1
      $USER
AQUI

# A saída ainda seria...

linha 1
blau
```

5.5.3 – Aqui está a here-string

Para ser bem franco, eu nunca achei as *here-docs* muito interessantes nos meus scripts, mas sou fã de carteirinha das *here-strings*! O mais legal de uma *here-string*, é que ela manda *qualquer string* para a entrada padrão – sem delimitadores, sem regras especiais para aspas, e com quantas linhas eu quiser!

Aqui está a sintaxe geral:

```
[n]<<< string
```

Simples assim. A string pode ser uma string literal ou o resultado de uma expansão. Isso quer dizer que podemos armazenar o "documento" numa variável e utilizar a *here-string* para mandá-lo para a entrada padrão:

Exemplo 5.4 – Lendo uma 'here-string' com o utilitário 'cat'.

```
# Enviando uma string para stdin...

:~$ cat <<< moleza
moleza

# Enviando a expansão de uma string para stdin...

:~$ var="banana
> laranja
> abacate"
:~$ cat <<< $var
banana
laranja
abacate
```

5.6 – Redirecionamento das saídas

Da mesma forma que o fluxo da *entrada padrão* (*stdin*) não precisa vir apenas de um teclado, a *saída padrão* (*stdout*) e da *saída padrão de erros* (*stderr*) não precisam necessariamente ser exibidos no terminal.

Para controlar o destino dos fluxos de *saída*, o Bash oferece dois operadores que já são nossos velhos conhecidos: `>` e `>>`, ambos para o envio de dados para *arquivos*. Nos tópicos *1.9.5*, *1.9.6* e *2.3.8*, nós vimos, de uma forma até que bem completa, o que são e alguns usos desses dois operadores. Agora, nossa tarefa é apenas completar essa informação com a apresentação das suas sintaxes e opções.

> *Novamente, o que nos interessa neste capítulo são apenas os redirecionamentos envolvendo os três fluxos de dados padrão, mas os conceitos são os mesmos para qualquer outro descritor de arquivos.*

5.6.1 – Redirecionamento da saída para arquivos

O operador de redirecionamento da saída para arquivos (`>`) faz com que os dados em um descritor de arquivos sejam escritos em um arquivo. Neste processo, se o arquivo de destino não existir, ele será criado – caso contrário, ele será *"truncado para o tamanho de zero bytes"* (uma forma chique de dizer *"apagado"*).

A sintaxe geral é:

```
[n]>[|] arquivo
```

Onde `n` é o número do descritor de arquivos (o padrão é `1`, se ele for omitido), e `arquivo` é o nome do arquivo de destino. Para entender a forma opcional

`>|`, porém, nós precisamos conhecer uma das opções de configuração mais sem sentido do shell: a opção `noclobber`.

Todo mundo erra, e pode ser trágico quando um redirecionamento para um arquivo é feito sem os devidos cuidados. Por isso, existe a opção `noclobber`, para que o comportamento destrutivo do `>` seja inibido e cause um erro se o arquivo de destino existir. Normalmente ela vem desabilitada, mas pode ser ativada e desativada com os comandos:

```
set -o noclobber ← Ativa
set +o noclobber ← Desativa
```

Mas, eu vou logo avisando: é uma péssima ideia mexer com isso!

Com a opção ativada, o operador `>` deixa de ser destrutivo, mas nós ainda podemos forçar a truncagem do arquivo de destino com sua forma alternativa `>|`.

Meu problema com essa opção, aliás, começa aí: por que alterar o comportamento do shell e oferecer uma forma de burlar a alteração feita se, sem alteração nenhuma, existem meios de garantir que a coisa seja feita de forma segura?

Ou pior: por que dar a opção do usuário ou do programador se tornar um desleixado? Sim, porque, uma coisa é o usuário cometer erros. Mas, quando se oferece um dispositivo para ofuscar o perigo, isso é um incentivo ao desleixo. Não seria melhor aprender a fazer algo assim?

```
[[ -e arquivo ]] && echo erro! || comando > arquivo
```

Enfim, a opção existe, mas eu recomendo fortemente que ela seja mantida como está, desativada, e que todos os cuidados normais com operações arriscadas sejam tomadas.

Voltando ao uso racional do operador >, com ele, também é possível redirecionar a saída de erros para arquivos (o que é muito útil para inspecionar o último erro ocorrido na execução de um comando).

Para isso, basta indicar que queremos desviar o fluxo no descritor de arquivos `2`:

```
comando 2> arquivo
```

Quando queremos que as duas saídas vão para o arquivo, basta utilizar a forma:

```
comando &> arquivo
```

Onde o & significa *"1 e 2"*. Eventualmente, você pode encontrar uma das formas abaixo sendo utilizadas com o mesmo propósito:

```
comando >& arquivo
comando > arquivo 2>&1
```

Mas a primeira forma (`&>`) ainda é a preferível – tanto por legibilidade quanto por compatibilidade.

5.6.2 – Redirecionamento para *append* em arquivos

Se, em vez de truncar o conteúdo do arquivo, você quer incluir novas linhas a partir do que vier das saídas, o operador `>>` é o seu amigo:

```
[n]>> arquivo
```

O funcionamento é idêntico ao do operador `>`, inclusive no fato dele criar o arquivo se ele não existir. A única diferença é que, com `>>`, os dados são incluídos no final arquivo sem alterar o que já estiver nele.

Para enviar tanto a saída padrão quanto a de erros para o final de um arquivo (o que, desta vez, é ideal para a criação de arquivos de *log* ou de históricos), a sintaxe também é parecida:

```
comando &>> arquivo
```

5.7 – Pipes

Os pipes são outro exemplo de mecanismos do shell para direcionar fluxos de dados. Com eles, os dados na saída de um comando são redirecionados para a entrada de outro. Isso nos permite criar uma cadeia de processamento onde cada comando cuida do tratamento dos dados conforme a sua especialidade.

Observe:

```
$ echo 'banana
> laranja
> abacate
> morango
> morango
tangerina' | sort
abacate
banana
laranja
morango
morango
tangerina
```

Aqui, a saída do comando `echo` foi enviada por *pipe* para a entrada do utilitário `sort`, que exibiu as linhas em ordem alfabética. Mas, nós poderíamos encadear comandos:

```
$ echo 'banana
> laranja
> abacate
```

```
> morango
> morango
tangerina' | sort | uniq
abacate
banana
laranja
morango
tangerina
```

Então, tendo recebido a saída do `sort` em sua entrada, o utilitário `uniq` removeu a ocorrência duplicada da linha `morango`.

Isso é só uma pequena amostra do potencial embutido na simplicidade dos *pipes*, mas existem dois pontos que eu acho que merecem a nossa atenção. O primeiro deles é que, cada novo comando encadeado em um pipe será executado em um subshell (*tópico 3.6.3*), o que não é um problema em si, mas exige a nossa atenção quanto ao escopo das variáveis em uso.

O segundo, é que não é uma boa prática exagerar na quantidade de comandos encadeados. Geralmente, quando caímos na tentação de criar pipes infinitos, é porque tem alguma coisa errada na nossa forma de abordar o problema. Talvez, numa linha de comandos, essa prática seja até aceitável (nós queremos uma solução rápida e prática para ser utilizada apenas naquela vez), mas um programa em Bash exige e permite mais planejamento.

Apenas para fechar este tópico sobre *pipes*, que é essencialmente introdutório, é interessante saber que não é só a saída padrão que pode ser enviada para a entrada de comandos no *pipe* – a saída de erros também pode ir junto com o uso do operador `|&`.

> ***Importante!*** *Uma sequência de comandos encadeados por 'pipe' é chamada de 'pipeline' e é tratada como uma unidade na formação de uma 'lista de comandos' (tópicos 2.1.3, 2.1.4 e 2.1.5) .*

6 – Operadores e expressões

O Bash oferece operadores para diversas finalidades e, de acordo com o contexto, um mesmo símbolo pode assumir significados totalmente diferentes. Por exemplo, o operador `|` pode ser tanto um *pipe*, onde seria um *operador de controle*, quanto o operador lógico bit-a-bit *ou* – o que determina seu papel é o contexto de uso. De modo geral, porém, nós podemos pensar em dois grandes contextos: os *comandos* e as *expressões*.

Contexto dos comandos:

- Operadores de controle
- Operadores de redirecionamento

Contexto de expressões:

- Operadores de atribuição
- Operadores aritméticos
- Operadores de comparação numérica
- Operadores de comparação de strings
- Operadores lógicos
- Operadores lógicos bit-a-bit
- Operadores de arquivos

Uma forma de identificar o contexto com bastante chances de acerto, é observando se o operador aparece numa situação que resulta em um *valor* – se aparecer o contexto é de uma *expressão*, caso contrário, é bem provável que ele esteja ali apenas para controlar o encadeamento de comandos ou executar algum redirecionamento.

> *Neste capítulo, nosso objetivo não é demonstrar o uso de cada operador (não neste "pequeno manual" – em um "grande manual", talvez?), mas tão somente apresentar seus principais conceitos e os mecanismos em que eles se encaixam.*

6.1 – Operadores do contexto de comandos

Independente de seu tipo, todo operador no contexto dos comandos é formado de um ou mais *metacaracteres*.

> ***Metacaracteres** são os caracteres que o Bash utiliza para separar as 'palavras' de um comando. Eles podem ser um espaço, uma tabulação, uma quebra de linha ou os caracteres* `|`, `&`, `;`, `(`, `)`, `<` *ou* `>`.

Os operadores utilizados no contexto dos comandos possuem uma característica em comum: todos eles são *separadores de palavras* e, portanto, a menos que possam ser confundidos com as *palavras* que eles separam, não existe a obrigatoriedade de se deixar um espaço antes ou depois deles.

Por exemplo:

```
:~$ [[ -f log.txt ]]&&echo bom||echo ruim
bom
:~$ [[ -f xxx.txt ]]&&echo bom||echo ruim
ruim
```

> ***Só para reforçar:** espaços, tabulações e quebras de linha também são 'metacaracteres' e, obviamente, 'separadores de palavras'.*

6.1.1 – Operadores de controle

Operador	Função
`&&`	Encadeamento condicional de comandos. Executa o comando seguinte se o anterior terminar com estado de "sucesso" (saída `0`).
`||`	Encadeamento condicional de comandos. Executa o comando seguinte se o anterior terminar com estado de "erro" (saída *diferente* de `0`).
`|`	*Pipe* (*tópico 5.7*). Cria uma *pipeline*, onde cada comando envia a sua saída padrão (*stdout*) para a entrada padrão (*stdin*) do comando seguinte.
`|&`	Um pipe que, além da saída padrão, envia a saída padrão de erros (*stderr*) para o comando seguinte.
`;`	Indica o fim de cada comando ou *pipeline* em uma lista de comandos escrita em uma mesma linha. Quando a lista é escrita em várias linhas, o `;` é dispensável, tendo sua função substituída pela ocorrência de uma ou mais quebras de linha.
`&`	Executa comandos em *segundo plano*. Quando aparece delimitando o fim de um comando, faz com ele seja executado *assincronamente* em um *subshell*. Desta forma, o restante da lista de comandos, se houver, continua sendo executada de forma totalmente independente do comando que foi para o *subshell*.
`(lista...)`	Agrupa uma *lista de comandos*. Com os parêntesis, toda a lista será executada em um *subshell* e será tratada como se fosse apenas um comando para efeito de redirecionamentos e de estado de saída.
`;;`	Delimita o fim de uma lista de comandos em uma cláusula do comando composto `case`, causando o término dos testes

	comparativos dos padrões (equivale à instrução *break* de outras linguagens).
`;&`	Delimita o fim de uma lista de comandos em uma cláusula do comando composto `case`, mas indica que a lista de comandos da cláusula seguinte também deve ser executada (incondicionalmente).
`;;&`	Delimita o fim de uma lista de comandos em uma cláusula do comando composto `case`, mas indica que os padrões de todas as cláusulas subsequentes também devem ser testados (faz o `case` do shell se comportar de forma equivale ao padrão de uma estrutura *switch-case* de outras linguagens).

6.1.2 – Operadores de redirecionamento

Operador	Função
`[n]>`	Escreve o fluxo de dados no descritor de arquivos `n` em um arquivo. Se `n` for omitido, os dados na *saída padrão* (descritor de arquivos `1`, *stdout*) serão escritos no arquivo. Se o arquivo existir, seu conteúdo será apagado antes da escrita. Se não existir, o arquivo será criado.
`[n]>>`	Escreve o fluxo de dados no descritor de arquivos `n` no final de um arquivo (*append*). Se `n` for omitido, os dados na *saída padrão* (*stdout*) serão escritos no arquivo. Se o arquivo existir, seu conteúdo será mantido e os dados serão escritos após o conteúdo original. Se não existir, o arquivo será criado.
`&>`	Escreve o fluxo de dados na saída padrão (*stdout*) e na saída de erros (*stderr*) em um arquivo. Se o arquivo existir, seu conteúdo será apagado antes da escrita. Se não existir, o arquivo será criado.
`&>>`	Escreve o fluxo de dados na saída padrão (*stdout*) e na saída

	de erros (*stderr*) no final de um arquivo (*append*). Se o arquivo existir, seu conteúdo será mantido e os dados serão escritos após o conteúdo original. Se não existir, o arquivo será criado.
`[n]<`	Abre um arquivo para leitura no descritor de arquivos `n` ou `0` (*stdin*, entrada padrão), se `n` for omitido.
`<<`	*Here-document* (ou *here-doc*). Envia uma string iniciada e terminada com uma palavra delimitadora para a entrada padrão (*stdin*).
`<<<`	*Here-string*. Envia uma string para a entrada padrão (*stdin*)
`[n]>&-`	Fecha o descritor de arquivos `n`.
`>&[n]`	Escreve no descritor de arquivos `n`.
`<&[n]`	Lê o descritor de arquivos `n`.
`[n]<>`	Abre um arquivo para *leitura e escrita* no descritor de arquivos `n`. Se `n` não for especificado, utiliza o descritor de arquivos `0`.
`[n]>&[m]`	Copia o descritor de arquivos `m` no descritor de arquivos `n`.
`[n]>&[m]-`	Copia o descritor de arquivos `m` no descritor de arquivos `n` e fecha o descritor de arquivos `m` (*move* `m` para `n`).

É possível realizar múltiplos redirecionamentos em um mesmo comando, bastando prestar atenção na ordem em que eles serão executados – da esquerda para a direita.

6.2 – Operadores do contexto de expressões

Por definição, uma expressão é algo que *expressa um valor*. No Bash, porém, o valor de uma expressão só pode ser obtido através de um comando que seja

capaz de avaliá-la. Sendo assim, um comando de atribuição, por exemplo, não expressa um valor por si só. Mas, uma atribuição avaliada pelo comando composto `(( ))`, também por exemplo, receberia um valor – logo, seria uma *expressão*.

Observe:

```
:~$ a=10
:~$
```

Nesta atribuição, um novo *parâmetro* foi criado com o valor `10` e com o identificador (nome) `a`. O parâmetro tem um valor, mas o comando em si não expressa nada, como você pode ver abaixo:

```
:~$ a=10
:~$ echo $a
10
:~$ b=a=10
:~$ echo $b
a=10
```

Se `a=10` tivesse algum valor, a expansão do parâmetro `b` seria `10`, mas o shell viu apenas uma string representando a atribuição.

Agora, veja o que acontece quando a atribuição é feita no interior do comando composto `(( ))`:

```
:~$ ((b=a=15))
:~$ echo $a
15
:~$ echo $b
15
```

Em expressões aritméticas não precisamos utilizar o `$` antes do nome da variável para acessar o seu valor.

Decompondo a sequência dos acontecimentos, `15` é a *expressão* literal do *valor* inteiro 15. Ele foi avaliado pelo comando `(( ))` e seu valor foi atribuído à variável `a`, que também passou a ser uma *expressão* com *valor* 15. Por último, uma segunda atribuição fez com que `b` também se tornasse uma *expressão* do *valor* 15, desta vez, porque recebeu o *valor* da variável `a`. Então, no exemplo acima, o valor 15 é expresso de várias formas, o que podemos ver *expandindo* o comando `(( ))`:

```
:~$ echo $((b=a=15))
15
:~$ echo $((a=15))
15
:~$ echo $((b=a))
15
:~$ echo $((b))
15
:~$ echo $((a))
15
:~$ echo $((15))
15
```

Isso é fundamental para o entendimento dos comandos que avaliam expressões de uma das duas formas: quanto à sua *verdade* ou quanto ao *valor numérico* que representam. Essas duas formas de avaliação são feitas por dois grupos de comandos internos do Bash:

> **Comandos:** `test`, `[ expressão ]` e `[[ expressão ]]`
> Avaliam expressões na forma de proposições assertivas.
>
> **Comandos:** `declare -i`, `let` e `(( ))`
> Avaliam expressões aritméticas.

No primeiro grupo, os operadores entram compondo uma expressão que representa uma afirmação, como: o *"arquivo tal existe"*, *"a string x é igual a string y"*, e por aí vai. Como resultado, a expressão é avaliada como *verdadeira*

ou *falsa*, fazendo o comando terminar com um estado de saída de *sucesso* ou de *erro* – o que podemos capturar com a variável especial `?`.

No segundo grupo, os operadores servem para formar expressões que serão avaliadas como números inteiros em operações de atribuição, de incremento, de comparação, aritméticas, lógicas e condicionais.

6.3 – Operadores de expressões afirmativas

No *tópico 2.2.9*, eu explico com detalhes por que me recuso a chamar as expressões formadas por este grupo de operadores de "expressões condicionais". Mas, se não estiver curioso ao ponto de voltar até lá, aqui está a versão curta da minha "bronca": as expressões nos comandos `test`, `[ ]` e `[[ ]]` não são *condicionais.* Elas apresentam uma *afirmação* que será testada *quanto à sua verdade* – e isso não tem nada de *condicional*, é apenas a *avaliação lógica de uma "proposição"*.

Uma expressão realmente condicional seria, por exemplo, aquela formada por um *operador ternário*, onde o valor de uma expressão estivesse condicionado à avaliação de, pelo menos, duas outras expressões, mas só os comandos relacionados com as operações aritméticas possuem esse tipo de operador, então…

Voltemos aos nossos operadores de expressões afirmativas!

> ***Atenção!*** *Nem todos os operadores válidos para o comando composto* `[[ ]]` *(exclusivo do Bash) são válidos para os comandos* `test` *e* `[ ]`*.*

6.3.1 – Operadores unários de arquivos

Operador	Função

`-a ARQUIVO`	Afirma que `ARQUIVO` existe.
`-e ARQUIVO`	Afirma que `ARQUIVO` existe (igual a `-a`).
`-d ARQUIVO`	Afirma que `ARQUIVO` é um *diretório*.
`-f ARQUIVO`	Afirma que `ARQUIVO` é um *arquivo comum*.
`-s ARQUIVO`	Afirma que `ARQUIVO` não está vazio.
`-h ARQUIVO`	Afirma que `ARQUIVO` é um *link simbólico*.
`-L ARQUIVO`	Afirma que `ARQUIVO` é um *link simbólico* (igual a `-h`).
`-N ARQUIVO`	Afirma que `ARQUIVO` foi alterado desde a última leitura.
`-r ARQUIVO`	Afirma que `ARQUIVO` pode ser lido pelo usuário logado.
`-w ARQUIVO`	Afirma que `ARQUIVO` pode ser escrito pelo usuário logado.
`-x ARQUIVO`	Afirma que `ARQUIVO` pode ser executado pelo usuário logado.
`-O ARQUIVO`	Afirma que `ARQUIVO` pertence ao usuário logado.
`-G ARQUIVO`	Afirma que `ARQUIVO` pertence ao grupo do usuário logado.
`-g ARQUIVO`	Afirma que `ARQUIVO` está associado à identificação de um grupo (*group id*).
`-e FD`	Afirma que o descritor de arquivos `FD` foi aberto em um terminal.
`-p ARQUIVO`	Afirma que `ARQUIVO` é um *pipe nomeado*.
`-k ARQUIVO`	Afirma que `ARQUIVO` está com o bit de *fixado* ativo (só o *usuário* ou o *root* podem alterá-lo).
`-c ARQUIVO`	Afirma que `ARQUIVO` é um *arquivo especial de*

	caractere.
`-b ARQUIVO`	Afirma que `ARQUIVO` é um *arquivo especial de bloco*.

Em sistemas *unix-like*, existem os chamados "*arquivos especiais*", que são aqueles relacionados com a representação de diversos dispositivos no sistema de arquivos. Em alguns deles, o acesso aos dados se dá em *"blocos"*, por isso são chamados de "*arquivos especiais de bloco*". Já em outros, o acesso é feito *"um caractere por vez"*, daí a terminologia: "*arquivos especiais de caractere*".

6.3.2 – Operadores binários de arquivos

Operador	Função
`ARQ1 -nt ARQ2`	Afirma que `ARQ1` é mais novo do que `ARQ2`.
`ARQ1 -ot ARQ2`	Afirma que `ARQ1` é mais antigo do que `ARQ2`.
`ARQ1 -ef ARQ2`	Afirma que `ARQ1` é uma *ligação física* (*hard link*) para `ARQ2`.

> ***Relembrando:*** *uma "ligação física" (hard link) é quando um arquivo é apenas um segundo nome para outro arquivo.*

6.3.3 – Operadores de strings

Operador	Função
`-z STRING`	Afirma que `STRING` é uma string vazia.
`-n STRING`	Afirma que `STRING` *não é* uma string vazia. Nos testes lógicos, este é o operador padrão e pode ser omitido.

`STR1 = STR2`	Afirma que `STR1` é igual a `STR2`.
`STR1 != STR2`	Afirma que `STR1` é igual a `STR2` que, no caso do comando `[[ ]]`, representa um *padrão*.
`STR1 == PADRÃO`	No comando `[[ ]]`, afirma que `STR1` casa com `PADRÃO`.
`STR1 =~ REGEX`	No comando `[[ ]]`, afirma que `STR1` casa com a `REGEX`.
`STR1 < STR2`	Afirma que `STR1` é ordenada *antes* de `STR2`.
`STR1 > STR2`	Afirma que `STR1` é ordenada *depois* de `STR2`.

A comparação feita pelos operadores `<` e `>` é lexicográfica, ou seja, segue a ordem alfabética natural (a ordem dos dicionários).

6.3.4 – Operadores de comparação numérica

Operador	Função
`NUM1 -eq NUM2`	Afirma que `NUM1` é igual a `NUM2`.
`NUM1 -ne NUM2`	Afirma que `NUM1` é diferente de `NUM2`.
`NUM1 -lt NUM2`	Afirma que `NUM1` é menor que `NUM2`.
`NUM1 -gt NUM2`	Afirma que `NUM1` é maior que `NUM2`.
`NUM1 -le NUM2`	Afirma que `NUM1` é menor ou igual a `NUM2`.
`NUM1 -ge NUM2`	Afirma que `NUM1` é maior ou igual a `NUM2`.

6.3.5 – Operadores para configurações e variáveis

Operador	Função
`-o OPÇÃO`	Afirma que a opção do shell `OPÇÃO` está habilitada.
`-v VARIÁVEL`	Afirma que `VARIÁVEL` está definida.
`-R VARIÁVEL`	Afirma que `VARIÁVEL` está definida e é uma *referência* para um nome (definida com o comando `declare -n`).

6.3.6 – Operadores lógicos

Operador	Função
`EXP1 -a EXP2`	Afirma que ambas as expressões são verdadeiras.
`EXP1 -o EXP2`	Afirma que uma ou ambas as expressões são verdadeiras.
`EXP1 && EXP2`	No comando `[[ ]]`, afirma que ambas as expressões são verdadeiras.
`EXP1 || EXP2`	No comando `[[ ]]`, afirma que uma ou ambas as expressões são verdadeiras.
`! EXPRESSÃO`	Afirma que `EXPRESSÃO` é falsa.
`( EXPRESSÃO )`	Avalia `EXPRESSÃO` antes das demais expressões.

Os operadores `&&` e `||` fazem com que o comando não avalie a expressão da direita se a expressão da esquerda for suficiente para determinar o resultado avaliado: `true || tanto faz = true` ou `false && tanto faz = false`.

6.4 – Operadores de expressões aritméticas

As expressões aritméticas são avaliadas pelo comando interno `let`, pelo comando composto `(( ))` , pelo comando `declare -i`, e pela *expansão aritmética*. Apenas valores inteiros podem participar dessas expressões, mas as variáveis não precisam ter o atributo de *inteiro* ligado, basta que elas expandam valores inteiros.

Se o valor for literal, ele pode ser expresso em bases de numeração diferentes da base 10. Se o número começar com `0`, ele será considerado um inteiro *octal* (base 8), se começar com `0x` ou `0X`, ele será considerado um inteiro hexadecimal (base 16), fora essas convenções, nós podemos representar o valor em qualquer base numérica com o prefixo `base#`, onde `base` é qualquer número decimal entre 2 e 64.

Aliás, em Bash, nós não precisamos escrever códigos complicados para converter um número escrito em qualquer base para a base 10 – basta fazer uma expansão aritmética do valor de acordo com a notação da sua base:

```
# Base 2 para base 10
:~$ echo $((2#0110))
6

# Base 16 para base 10
:~$ echo $((0x1f4))
500

# Base 8 para base 10
:~$ echo $((0123))
83
```

Mas é sempre bom tomar cuidado com a forma que os valores são escritos ou expandidos! É muito comum um número decimal ser coletado de uma fonte de dados qualquer com zeros a esquerda, o que poderia gerar confusões como esta:

```
# Aqui, os zeros à esquerda denotam valores em base 8!
:~$ echo $(( 120 + 066 + 014 ))
186

# O resultado é bem diferente com os valores na base 10...
:~$ echo $(( 120 + 66 + 14 ))
200
```

Outra coisa que precisa ficar bem clara antes das nossas tabelas de operadores, é que os comandos `let` e `(( ))` não enviam valores para a saída padrão! Quando isso é necessário, nós utilizamos a *expansão aritmética*.

Quanto aos estados de saída, eles sempre serão `1` (*erro*), se a avaliação da expressão resultar em `0`, ou `0` (*sucesso*), se qualquer outro valor for avaliado.

> *O comando `(( ))` permite espaços entre os operadores e operandos, a separação de múltiplas expressões com vírgulas e a alteração da precedência dos operadores com o uso de parêntesis. Com o comando `let`, isso só é possível com as expressões entre aspas.*

6.4.1 – Operadores de atribuição

Operador	Função
`NOME = VALOR`	Atribui `VALOR` à variável `NOME`.
`NOME += VALOR`	O valor em `NOME` é o seu valor atual acrescido de `VALOR`.
`NOME -= VALOR`	O valor em `NOME` é o seu valor atual menos `VALOR`.
`NOME *= VALOR`	O valor em `NOME` é o seu valor atual multiplicado por `VALOR`.
`NOME /= VALOR`	O valor em `NOME` é o seu valor atual dividido por `VALOR`.

`NOME %= VALOR`	O valor em `NOME` é o resto da divisão de seu valor atual por `VALOR`.
`NOME <<= N`	O valor em `NOME` é o seu valor atual em binário com seus bits deslocados `N` vezes para a esquerda.
`NOME >>= N`	O valor em `NOME` é o seu valor atual em binário com seus bits deslocados `N` vezes para a direita.
`NOME &= VALOR`	O valor em `NOME` é o resultado da operação lógica E (*and*) de seu valor atual em binário comparado bit-a-bit com o valor binário em `VALOR`.
`NOME |= VALOR`	O valor em `NOME` é o resultado da operação lógica OU (*or*) de seu valor atual em binário comparado bit-a-bit com o valor binário em `VALOR`.
`NOME ^= VALOR`	O valor em `NOME` é o resultado da operação lógica OU EXCLUSIVA (*xor*) de seu valor atual em binário comparado bit-a-bit com o valor binário em `VALOR`.

6.4.2 – Operadores aritméticos

Operador	Função
`A + B`	Soma `A` com `B`.
`A - B`	Subtrai `B` de `A`.
`A * B`	Multiplica `A` por `B`.
`A / B`	Divide `A` por `B`.
`A % B`	Módulo (resto) da divisão de `A` por `B`.
`A ** B`	Eleva `A` à potência `B`.

6.4.3 – Operadores de incremento e decremento

Operador	Função
`VAR++`	Pós-incremento: avalia o valor na variável VAR e soma 1.
`++VAR`	Pré-incremento: soma 1 ao valor de VAR e avalia o resultado.
`VAR--`	Pós-decremento: avalia o valor na variável VAR e subtrai 1.
`--VAR`	Pré-decremento: subtrai 1 do valor de VAR e avalia o resultado.

6.4.4 – Operadores bit-a-bit

Operador	Função
`A & B`	Realiza uma operação lógica E (*and*) avaliando bit-a-bit os valores binários de `A` e `B`.
`A | B`	Realiza uma operação lógica OU (*or*) avaliando bit-a-bit os valores binários de `A` e `B`.
`A ^ B`	Realiza uma operação lógica OU EXCLUSIVA (x*or*) avaliando bit-a-bit os valores binários de `A` e `B`.
`~VALOR`	Inverte cada um dos bits de `VALOR`.
`VALOR << N`	Desloca os bits de `VALOR` para a esquerda `N` vezes.
`VALOR >> N`	Desloca os bits de `VALOR` para a direita `N` vezes.

6.4.5 – Operadores lógicos

Operador	Função
`EXP1 && EXP2`	Avalia como verdadeiro se `EXP1` E `EXP2` forem avaliadas como verdadeiras.
`EXP1 || EXP2`	Avalia como verdadeiro se `EXP1` OU `EXP2` forem avaliadas como verdadeiras.
`! EXPRESSÃO`	Nega (inverte) a avaliação lógica de `EXPRESSÃO`.

6.4.6 – Operadores de comparação

Operador	Função
`A == B`	Avalia como verdadeiro se o valor `A` for igual ao valor `B`.
`A != B`	Avalia como verdadeiro se o valor `A` for diferente do valor `B`.
`A > B`	Avalia como verdadeiro se o valor `A` for maior do que o valor `B`.
`A >= B`	Avalia como verdadeiro se o valor `A` for maior ou igual ao valor `B`.
`A < B`	Avalia como verdadeiro se o valor `A` for menor do que o valor `B`.
`A <= B`	Avalia como verdadeiro se o valor `A` for menor ou igual ao valor `B`.

6.4.7 – Operador condicional

```
EXP1 ? EXP2 : EXP3
```

O operador ternário condicional faz com que toda a expressão tenha o valor da expressão `EXP2` se `EXP1` for *verdadeira*. Se `EXP1` for avaliada como *falsa*, o valor será o da expressão `EXP3`.

6.4.8 – Operadores unários de sinal

São os operadores que definem o sinal de um valor:

```
+VALOR
-VALOR
```

6.5 – Precedência de operadores

Todos os operadores vistos neste capítulo podem ser utilizados para formar múltiplas expressões que serão avaliadas como um todo em um mesmo comando. Nestes casos, é importante saber a ordem em que elas serão avaliadas, o que se dá pela propriedade da *precedência dos operadores*. Por exemplo:

```
:~$ echo $((3 + 4 * 5))
23
```

O resultado foi 23 porque, mesmo aparecendo depois, a multiplicação (`*`) tem precedência sobre a soma (`+`). Se não houver nada alterando a precedência natural, o que geralmente se faz com o uso de parêntesis, todos os operadores do contexto de comandos e das avaliações de expressões afirmativas serão interpretados da esquerda para a direita, mas os operadores

utilizados nas expressões aritméticas possuem uma ordem bem específica de precedência, o que podemos ver no diagrama abaixo:

```
MAIOR PRECEDÊNCIA

VAR++, VAR--
++VAR, --VAR
-VALOR, +VALOR
! EXPRESSÃO , ~VALOR
**
*, /, %
+, -
<<, >>
<=, >=, <, >
==, !=
&
^
|
&&
||
EXP1 ? EXP2 : EXP3
=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=

MENOR PRECEDÊNCIA
```

Para alterar a precedência nas expressões aritméticas, nós utilizamos parêntesis. Por exemplo, se quiséssemos que a soma fosse efetuada antes da multiplicação...

```
:~$ echo $(( (3 + 4) * 5 ))
35
```

Outro ponto a que devemos ficar atentos, é que cada comando tem uma ordem de avaliação dos seus termos (*operandos*), que é definida por uma propriedade conhecida como *associabilidade*.

Por exemplo, na atribuição abaixo, os termos à direita dos operadores `=` são avaliados antes de cada atribuição acontecer:

```
:~$ ((a=b=15)))
```

Observe no diagrama abaixo como isso funciona para os operadores lógicos e aritméticos:

```
DIREITA PARA A ESQUERDA

VAR++, VAR--, ++VAR, --VAR
-VALOR, +VALOR
! EXPRESSÃO , ~VALOR
=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=
**

ESQUERDA PARA A DIREITA

*, /, %, +, -
<<, >>
&, ^, |
&&, ||
EXP1 ? EXP2 : EXP3

NÃO SE ASSOCIAM

<=, >=, <, >, ==, !=
```

7 – Comandos compostos

Os *comandos compostos* são estruturas do Bash que permitem o agrupamento de vários comandos diferentes em blocos, a fim de que eles atuem e sejam tratados como um único comando. Existem quatro categorias de comandos compostos, cada uma delas classificada pela forma de agrupamento dos comandos e pelo seu contexto léxico:

Agrupamentos de comandos

- Agrupamento em *chaves*
- Agrupamento em *parêntesis*

Estruturas de repetição (*loops*)

- Loop `for`
- Loops `while` e `until`
- Menu `select`

Estruturas de decisão

- Bloco `if`
- Bloco `case`

Estruturas de avaliação de expressões

- Avaliação de expressões afirmativas: `[[ expressão ]]`
- Avaliação de expressões aritméticas: `((expressão))`

Neste capítulo, nós veremos cada um desses comandos compostos, exceto as estruturas de *avaliação de expressões*, que foram o assunto do capítulo anterior.

7.1 – Agrupando comandos com chaves e parêntesis

A principal utilidade de um agrupamento está na possibilidade de capturarmos a saída de cada comando em um único fluxo de dados.

Observe o exemplo:

```
:~$ { echo banana; echo laranja; } > frutas.txt
:~$ echo "$(< frutas.txt)"
banana
laranja
```

Se você pensar bem, com o operador `>`, o arquivo `fruta.txt` seria apagado a cada redirecionamento da saída individual dos comandos `echo`:

```
:~$ echo banana > frutas.txt; echo laranja > frutas.txt
:~$ echo "$(< frutas.txt)"
laranja
```

Isso não aconteceu no primeiro exemplo porque, estando os comandos agrupados, existe apenas um fluxo de dados e ele só se fecha ao final da execução do último comando.

O mesmo poderia ser feito com parêntesis:

```
:~$ (echo fusca; echo corcel) > carros.txt
:~$ echo "$(< carros.txt)"
fusca
corcel
```

Contudo, existem diferenças importantes entre essas duas formas de agrupamento de comandos:

Agrupamento com chaves:

- Os comandos são executados na mesma sessão do shell;

- As chaves são *palavras reservadas* do shell e, por isso, *exigem* espaços entre elas e os comandos da lista;
- Também é obrigatório haver algum tipo de operador de controle (*tópico 6.1.1*) terminando o último comando, geralmente `;`, `&` ou uma *quebra de linha*.

Agrupamento com parêntesis:

- Os comandos são executados em um subshell;
- Os parêntesis são *metacaracteres* (*tópico 6.1*), por isso não precisamos de espaços nem da terminação do último comando;
- Se prefixado com `$`, um agrupamento com parêntesis será tratado como uma expansão, a *substituição de comandos*, e suas saídas poderão ser armazenadas em variáveis (*tópico 4.10*).

> *O estado de saída dos agrupamentos será o estado de saída do último comando executado.*

7.2 – Estruturas de decisão

As estruturas de decisão são aquelas em que blocos de comandos são executados condicionalmente. No Bash, nós podemos dizer que existem dois tipos de condições: o *estado de saída* de um comando e o casamento de uma string com um dado *padrão*.

Quando um estado de saída é a condição, nós podemos utilizar os operadores de controle condicional (`&&` e `||`, *tópicos 2.1.5 e 6.1.1*) ou o comando composto `if`.

Se a condição vier do casamento de um padrão, nós ainda podemos utilizar os operadores de controle condicional e o comando `if` testando a saída de qualquer comando que avalie uma *comparação de strings* (comandos `test`,

`[ ]` e `[[ ]]`), mas a forma mais direta e comum de se fazer isso é com o comando composto `case`.

7.2.1 – Grupos e operadores de controle condicional

Naturalmente, os operadores de controle condicional não são comandos compostos, mas, *com alguns cuidados*, eles podem trabalhar em parceria com os agrupamentos (com *chaves* ou *parêntesis*) para condicionar a execução de blocos de comandos.

Observe o exemplo:

Exemplo 7.1 – Controlando a execução de comandos com '&&' e '||'.

```
COMANDO && {
      COMANDOS SE SUCESSO (0)
      [true|:]
} || {
      COMANDOS SE ERRO (!=0)
}

# Ou então...

COMANDO || {
      COMANDOS SE ERRO (!=0)
      [false]
} && {
      COMANDOS SE SUCESSO (0)
}
```

Essa estrutura é bastante flexível, prática e legível, mas exige um cuidado: nós temos que garantir que o último comando de um agrupamento não saia com o estado necessário para permitir a execução do agrupamento seguinte. Ou seja, se o operador de controle seguinte for um `||` e o último comando do grupo sair com *erro*, todo o bloco seguinte será executado – e vice-versa. Se existir essa possibilidade, nós podemos utilizar os comandos `true`, `:` (que

sempre saem com *sucesso*) ou `false` (que sempre sai com *erro*) para garantir o estado de saída desejado.

Apesar de ser uma solução engenhosa, o melhor mesmo é ficar atento ao que pode acontecer com o último comando do grupo e decidir se vale mesmo a pena utilizar operadores de controle condicional – afinal, comandos sem um papel definido no algorítimo são sempre uma péssima ideia.

Porém, esse tipo de estrutura geralmente é utilizada com blocos que terminam em comandos como `echo` ou em atribuições de valores a variáveis, o que não coloca em risco a execução de blocos *condicionados a um estado de saída de erro*, ou com um comando `exit`, que faz com que a sessão do shell (e, obviamente, a execução do script) seja encerrada.

7.2.2 – O comando composto 'if'

Embora não tenha a flexibilidade dos operadores de controle condicional, o comando composto `if` oferece uma opção bem mais segura quanto ao que será ou não executado. A sua sintaxe não poderia ser mais simples:

`if COMANDO-TESTADO;`

> A palavra reservada `if` indica o início da estrutura condicionada ao estado de saída de `COMANDO-TESTADO`. A única estrutura obrigatória após o `if` é um bloco de comandos iniciado pela palavra reservada `then`.

`then LISTA-DE-COMANDOS;`

> A palavra reservada `then` indica o início de um bloco de comandos que será executado caso o comando testado por um `if` ou um `elif` saia com estado de *sucesso*. Todo bloco `then` precisa ser obrigatoriamente seguido de um `elif`, um bloco `else` ou do terminador `fi`.

`elif OUTRO-COMANDO-TESTADO;`

Opcionalmente, caso o comando testado pelo `if` saia com *erro*, nós podemos testar outros comandos com a palavra reservada `elif` (*else if*) associada obrigatoriamente a outro bloco `then`.

`else LISTA-DE-COMANDOS;`

A palavra reservada `else` inicia um bloco opcional de comandos que será executado caso o teste feito com o `if` e todos os testes feitos com a cláusula `elif`, se houver algum, resultarem em *erro*. O bloco else precisa ser *obrigatoriamente* antecedido por um bloco `then` e seguido pelo terminador `fi`.

`fi`

A palavra reservada `fi` (*if* ao contrário) delimita o fim do comando composto iniciado pelo `if`.

O exemplo abaixo mostra algumas combinações possíveis dos elementos do comando composto `if`:

Exemplo 7.2 – Condicionando a execução de blocos com 'if'.

```
# Não faz nada no caso do comando testado sair com 'erro'...
if COMANDO-TESTADO; then
      COMANDOS SE SUCESSO
fi

# Oferece uma alternativa se o comando testado sair com 'erro'...
if COMANDO-TESTADO; then
      COMANDOS SE SUCESSO
else
      COMANDOS SE ERRO
fi

# Testa outro comando de o comando testado sair com 'erro'...
if COMANDO-TESTADO; then
      COMANDOS SE SUCESSO
elif OUTRO-COMANDO-TESTADO; then
      OUTROS COMANDOS SE SUCESSO
else
```

```
     COMANDOS SE ERRO
fi
```

O `if` e o `elif` podem testar qualquer comando, mas é muito comum que eles sejam vistos testando a saída dos comandos `(( ))`, `[[ ]]` ou `[ ]`, o que sempre confunde quem está começando no Bash, especialmente aqueles que possuem alguma experiência com outras linguagens de programação e acham que esses comandos fazem parte da estrutura condicional. Então, não custa nada repetir:

> *A estrutura `if` testa a saída de "comandos", qualquer comando, inclusive dos comandos `(( ))`, `[[ ]]` e `[ ]`, que testam expressões.*

Uma outra situação muito comum é precisarmos apenas de um bloco que seja executado caso o comando testado saia com estado de *erro*. Como o bloco `else` precisa ser antecedido por um bloco `then`, isso não seria possível:

```
:~$ if false; else echo erro; fi
bash: erro de sintaxe próximo ao token inesperado `else'
```

Nesses casos, a solução é inverter o resultado do teste com uma exclamação (`!`) e utilizar um bloco `then` normalmente:

```
:~$ if ! false; then echo erro; fi
erro
```

Nós também podemos testar o estado de saída de comandos agrupados ou de uma lista:

```
# Agrupamento com parêntesis...
:~$ if (true; false); then echo sucesso; else echo erro; fi
erro
```

```
# Agrupamento com chaves...
:~$ if { false; true; } then echo sucesso; else echo erro; fi
sucesso

# Uma lista de comandos...
:~$ if false; true; then echo sucesso; else echo erro; fi
sucesso
```

No caso das listas de comandos desagrupados, a negação só tem efeito quando aplicada ao último comando:

```
:~$ if ! true; false; then echo sucesso; else echo erro; fi
erro
:~$ if true; ! false; then echo sucesso; else echo erro; fi
sucesso
```

Já nos agrupamentos, a negação é aplicada à saída do grupo:

```
:~$ if ! (true; false); then echo sucesso; else echo erro; fi
sucesso
:~$ if ! { false; true; } then echo sucesso; else echo erro; fi
erro
```

Uma dica com base numa impressão pessoal: *evite utilizar cláusulas* `elif`.

Pense bem, quando o teste de `if` resulta em *erro* e não existe um bloco `else`, nada é executado e a estrutura condicional é terminada. Na minha opinião, se outro comando precisar ser testado, isso já caracteriza outro contexto e, portanto, merece sua própria estrutura condicional.

Não há nada que proíba o aninhamento de cláusulas `elif`, e eles até fazem algum sentido quando os comandos testados são avaliações de expressões que possuam algo em comum que as relacione. Fora isso, o aninhamento é uma opção que eu nem chego a considerar nos meus scripts.

> *Eventualmente, se a expressão avaliada for, por exemplo, uma comparação de strings, pode ser muito mais interessante recorrer à nossa próxima estrutura condicional do que aninhar cláusulas* `elif`.

7.2.3 – O comando composto 'case'

Imagine a situação abaixo:

```
if [[ ${str,,} == banana ]]; then
     echo Não gosto de $str.
elif [[ ${str,,} == laranja ]]; then
     echo Esta $str é muito azeda!
elif [[ ${str,,} == abacate ]]; then
     echo ${str^} é a minha fruta favorita!
else
     echo Não encontrei $str no mercado.
fi
```

Foi desse tipo de aninhamento que eu falei no tópico anterior. Veja que estou comparando a string em `str` com três outras strings. Aliás, sendo mais exato, esta nem é uma comparação de strings. No comando `[[ ]]`, o operador `==` compara uma string com um *padrão*. Apenas por acaso os padrões são strings literais, mas é isso que o operador faz.

Agora observe se isso não lhe parece muito mais racional e legível:

Exemplo 7.3 – Condicionando a execução de blocos com 'case'.

```
case ${str,,} in
    laranja)
        echo Esta $str é muito azeda!
        ;;
    banana)
        echo Não gosto de $str.
        ;;
    abacate)
```

```
        echo ${str^} é a minha fruta favorita!
        ;;
    *)
        echo Não encontrei $str no mercado.
        ;;
esac
```

O comando `case` compara uma string com vários *padrões* e executa o bloco de comandos relativo ao padrão casado. Os padrões seguem as mesmas regras da formação de nomes de arquivos e podem ser, inclusive, *globs estendidos*, se a opção `extglob` estiver habilitada.

Se precisarmos condicionar um mesmo bloco ao casamento com vários padrões diferentes, nós podemos separá-los com uma barra vertical (`|`), que aqui terá o significado de *"ou"*:

```
case $opt in
    a|A)
        echo Boa escolha!
        ;;
    b|B)
        echo Podia ser melhor...
        ;;
    c|C)
        echo Argh!
        ;;
    *)
        echo Melhor não escolher nada mesmo...
        ;;
esac
```

O exemplo serve para demonstrar o uso do separador de padrões que, aqui, deverão casar com os caracteres indicados em caixa alta ou em caixa baixa. Porém, na prática, é muito melhor fazer como nos primeiros exemplos, onde utilizamos uma expansão para tornar minúscula a string na variável de referência.

Observe também que o último padrão (`*`) casa com qualquer valor na variável de referência. Ele está ali para permitir a execução de um bloco de comandos no caso de não haver casamento com os padrões anteriores, mas isso não é obrigatório se não houver uma ação padrão a ser executada. De qualquer forma, é importante saber que a comparação é feita na ordem em que os padrões são apresentados, ou seja, se o padrão `*` aparecer antes das outras opções, seu bloco de comandos será sempre executado.

Para indicar o fim de um bloco de comandos, nós utilizamos o operador de controle `;;`, que encerra o case após a execução do bloco de comandos do padrão encontrado. Mas dependendo da situação, nós poderíamos ter optado pelos operadores `;&` ou `;;&`.

Para relembrar:

`;;`	Delimita o fim de uma lista de comandos em uma cláusula do comando composto `case`, causando o término dos testes comparativos dos padrões (equivale à instrução *break* de outras linguagens).
`;&`	Delimita o fim de uma lista de comandos em uma cláusula do comando composto `case`, mas indica que a lista de comandos da cláusula seguinte também deve ser executad (incondicionalmente).
`;;&`	Delimita o fim de uma lista de comandos em uma cláusula do comando composto `case`, mas indica que os padrões de todas as cláusulas subsequentes também devem ser testados (faz o `case` do shell se comportar de forma equivale ao padrão de uma estrutura *switch-case* de outras linguagens).

> *O bloco condicionado ao último padrão não precisa ser terminado com um operador de controle, mas é uma boa prática utilizar o `;;` mesmo assim.*

Finalmente, como você já deve ter notado, toda estrutura `case` precisa ser terminada com a palavra reservada `esac` (que é *case* ao contrário).

7.3 – Estruturas de repetição

O último tipo de comandos compostos é o que implementa estruturas capazes de iterar repetidamente um bloco de comandos condicionado aos elementos de uma lista de *palavras* ou ao estado de saída de comandos ou à avaliação de uma expressão aritmética ternária (como na linguagem C).

7.3.1 – O loop 'for'

Velho conhecido dos nossos exemplos nos capítulos anteriores, o comando composto `for` é uma estrutura de repetição geralmente preferida para as iterações com quantidades de ciclos bem definidas pelos elementos de uma lista de palavras ou pela avaliação de uma expressão aritmética ternária.

O uso mais comum do *loop* `for` obedece à seguinte sintaxe:

```
for VAR in PALAVRAS; do
      BLOCO DE COMANDOS
done
```

Em uma linha...

```
for VAR in PALAVRAS; do BLOCO DE COMANDOS; done
```

Cada palavra em `PALAVRAS` é atribuída temporariamente à variável `VAR` e causa um ciclo de execução do `BLOCO DE COMANDOS` iniciado pela palavra reservada `do` e terminado pela palavra reservada `done`.

Por exemplo:

Exemplo 7.4 – Percorrendo listas de palavras com o loop 'for'.

```
:~$ for bicho in burro cão gato galo; do echo $bicho; done
burro
cão
```

```
gato
galo
```

A lista de palavras também pode ser o resultado de uma expansão do shell:

Exemplo 7.5 – Percorrendo listas de palavras expandidas.

```
# Expandindo as palavras em uma string...
:~$ palavras='café pão leite'
:~$ for p in $palavras; do echo $p; done
café
pão
leite

# Expandindo as palavras em um vetor...
:~$ frutas=(pera uva 'banana da terra')
:~$ for f in "${frutas[@]}"; do echo $f; done
pera
uva
banana da terra

# Percorrendo as strings geradas por uma expansão de chaves...
for n in {1..5}; do echo $n; done
1
2
3
4
5
```

Quando a lista de palavras corresponde a todos os parâmetros posicionais passados para o script, nós podemos omitir a parte do `in PALAVRAS`. Para demonstrar isso, nós utilizaremos o comando interno `set --` para definir os parâmetros posicionais da sessão corrente do shell (*tópico 4.9.4*):

Exemplo 7.6 – Percorrendo todos os parâmetros posicionais com o loop 'for'.

```
:~$ set -- João Maria 'Luis Carlos'
:~$ for nome; do echo $nome; done
João
```

```
Maria
Luis Carlos
```

Isso é equivalente a `for VAR in "$@"...`

Em vez da lista de palavras, o *loop* `for` também pode ser controlado por uma *expressão aritmética ternária*, o que é conhecido como *"estilo C"*. Neste caso, a sintaxe da estrutura muda para:

```
for (( EXP1; EXP2; EXP3 )); do
      BLOCO DE COMANDOS
done
```

Ou, opcionalmente...

```
for (( EXP1; EXP2; EXP3 )) {
      BLOCO DE COMANDOS
}
```

Para entender com essa estrutura funciona, vamos pensar em etapas:

Primeira etapa:

> A expressão `EXP1`, que geralmente é a atribuição de um valor inicial a uma variável que será utilizada como referência, é avaliada. Esta avaliação ocorre apenas uma vez.

Segunda etapa:

> A expressão `EXP2` é avaliada quanto à verdade da sua afirmação. Geralmente, ela é uma comparação entre o valor na variável de referência e um valor inteiro utilizado como limite. Se `EXP2` for avaliada como *verdadeira* (qualquer valor diferente de `0`), haverá um ciclo de

execução do `BLOCO DE COMANDOS`. Caso seja avaliada como *falsa* (valor `0`), nenhum ciclo é executado e o *loop* `for` é encerrado.

Terceira etapa:

Após executar um ciclo do `BLOCO DE COMANDOS`, a expressão em `EXP3`, que costuma ser alguma alteração no valor da variável de referência, será avaliada.

Quarta etapa:

Completadas as etapas anteriores, `EXP2` é novamente avaliada para confirmar se a sua afirmação ainda é verdadeira. Se for, acontece um novo ciclo e uma nova avaliação de `EXP3`. Caso contrário, nenhum novo ciclo é executado e o *loop* `for` é encerrado.

Exemplo 7.7 – Controlando os ciclos de um loop 'for' estilo C.

```
# Delimitando o bloco de comandos com 'do...; done':
:~$ for (( n=0 ; n < 5; n++)) do echo $n; done
0
1
2
3
4

# Delimitando o bloco de comandos com chaves:
:~$ for (( n=0 ; n < 5; n++)) { echo $n; }
0
1
2
3
4
```

Perceba: *a decisão de utilizar a estrutura típica do shell ou o estilo C não tem nada a ver com melhor ou pior, mais difícil ou mais fácil, ou qualquer outro aspecto qualitativo. Ambas as formas são corretas e necessárias para resolver diferentes tipos de problemas. Já o uso das chaves, que é totalmente opcional*

> *no estilo C, deve ser evitado para não confundir com outros agrupamentos de comandos.*

Uma coisa curiosa, de que poucas pessoas se dão conta, é que nós podemos fazer um loop infinito com o `for`:

Exemplo 7.8 – Um loop 'for' infinito.

```
for (( ;1; )); do BLOCO DE COMANDOS; done
```

Como vimos, a expressão `EXP2` é que determina a continuação dos ciclos: se ela for receber qualquer valor diferente de zero (0), os ciclos continuam. Então, uma expressão literal (como o valor inteiro `1`, do exemplo) diferente de zero garantirá a continuidade infinita dos ciclos.

> ***Lembre-se:*** *as expressões aritméticas sempre saem com 'erro' se a avaliação resultar no valor `0`, e com estado de 'sucesso', se qualquer outro valor for avaliado!*

O mesmo resultado pode ser conseguido omitindo `EXP2`:

```
for (( ; ; )); do BLOCO DE COMANDOS; done
```

Todo *loop infinito*, porém, precisa de um comando que encerre os ciclos em algum momento. Há vários meios de se fazer isso, mas todas elas envolvem alguma forma de execução condicional do comando `break`, como veremos no próximo tópico.

7.3.2 – Os loops 'while' e 'until'

Os loops `while` e `until` condicionam a execução de um bloco de comandos ao estado de saída de um comando testado (como no `if`). A diferença entre

eles é que o `while` permite a execução do *loop* quando o comando testado sai com *sucesso*, ao passo que o `until` só permite a execução do *loop* se o estado de saída for *erro:*

```
while COMANDOS-TESTADOS; do
      BLOCO DE COMANDOS
done

until COMANDOS-TESTADOS; do
      BLOCO DE COMANDOS
done
```

Em uma linha...

```
while COMANDOS-TESTADOS; do BLOCO DE COMANDOS; done
until COMANDOS-TESTADOS; do BLOCO DE COMANDOS; done
```

Aqui, `COMANDOS-TESTADOS` podem ser comandos simples, listas de comandos ou agrupamentos, exatamente como vimos no *tópico 7.2.2*, sobre o comando composto `if`. Mas, de longe, os comandos mais utilizados são os testes de expressões, com `[ ]` ou `[[ ]]`, para testar algo que provoque o fim dos ciclos, e os comandos, `true`, `false` ou `:`, quando queremos um *loop infinito*.

Aqui estão alguns exemplos:

Exemplo 7.9 – Loops condicionais com 'while' e 'until'.

```
# Utilizando 'while' com uma expressão aritmética...
a=0; while ((a < 5)); do echo $((++a)); done
1
2
3
4
5

# O mesmo efeito com 'until'...
a=0; until ((a == 5)); do echo $((++a)); done
1
```

```
2
3
4
5
```

Como podemos observar, o *loop* `while` "*continuou enquanto*" o comando `(( ))` saiu com *sucesso*. O *loop* `until`, por sua vez, "*só parou quando*" o comando `(( ))` saiu *sucesso*. Ou seja...

> *O **sucesso** determina a 'continuidade' do `while` e a 'parada' do `until`.*

Eu acho melhor pensar assim para não me confundir, mas há outra forma de encarar o funcionamento dessas estruturas:

> *O **erro** determina a 'fim' do `while` e o 'começo' do `until`.*

7.3.3 – Cadê o loop 'do-while'?

Se você vem de alguma linguagem com sintaxes derivadas do C, é provável que esteja se perguntando sobre o *loop do-while* – mas só se ainda não notou um detalhe importante sobre as duas estruturas existentes no Bash:

> *No `while`, nós especificamos uma condição de continuidade: "faça enquanto sucesso". Já no `until`, nós especificamos uma condição de parada: "faça até que sucesso".*

Portanto, semanticamente, nós elaboramos a lógica do *loop* `while` de modo a que uma condição *impeça* um próximo ciclo, mas a lógica do *loop* `until` é elaborada para executar *pelo menos um ciclo* antes de encontrar a condição de parada – exatamente como um loop *do-while*.

7.3.4 – Loops infinitos com 'while' e 'until'

Quando falamos do *loop* `for`, nós vimos como podemos fazer *loops infinitos* com ele, mas isso não é muito comum. Até semanticamente, o *loop* `while` é uma estrutura muito mais apropriada para esta finalidade:

Exemplo 7.10 – Loops infinitos com 'while'.

```
while :; do
      BLOCO DE COMANDOS
done
```

> *O comando nulo* `:` *poderia ser trocado pelo comando* `true`.

O que não nos impede de utilizarmos o *loop* `until`:

Exemplo 7.11 – Loops infinitos com 'while'.

```
while false; do
      BLOCO DE COMANDOS
done
```

Porém, como já dissemos, um loop infinito exige alguma forma de parada, e é aí que entra em cena o comando interno `break`.

7.3.5 – Saindo do loop com 'break'

O comando interno `break` provoca a saída imediata de qualquer uma das estruturas de repetição do Bash.

Por exemplo:

Exemplo 7.12 – Saindo de um loop infinito com 'break'.

```
:~$ a=0; while :; do echo $((++a)); [[ $a -eq 5 ]] && break; done
1
2
3
4
5
```

No caso de loops aninhados, o `break` *pode receber um número inteiro como argumento informando qual nível de loop deve ser interrompido. O inteiro 1 corresponde ao loop de nível mais interno (o próprio loop onde o* `break` *é executado), e os inteiros maiores que 1 são os níveis mais externos.*

7.3.6 – Pulando o restante da iteração com 'continue'

Também utilizado em qualquer uma das estruturas de repetição do Bash, o comando interno continue serve para que o restante da iteração por um bloco de comandos seja interrompida e o próximo ciclo seja executado.

Por exemplo:

Exemplo 7.12 – Saltando uma iteração com 'continue'.

```
:~$ for c in {A..E}; do [[ $c == C ]] && continue; echo $c; done
A
B
D
E
```

Repare que o comando `echo` não foi executado quando o valor na variável `c` era o caractere `C`.

Tal como o `break`*, o comando* `continue` *também pode receber um número inteiro como argumento informando qual nível de iteração deve ser saltado. O*

> *inteiro 1 corresponde ao loop de nível mais interno (o próprio loop onde o* `continue` *é executado), e os inteiros maiores que 1 são os níveis mais externos.*

7.3.7 – Criando menus com loops infinitos

Apesar de o Bash oferecer uma estrutura de repetição específica para isso, devido à flexibilidade que oferece, o loop infinito com o `while` é a construção mais utilizada para a criação de menus.

As formas de implementar menus com o loop while variam muito de caso para caso, mas o exemplo abaixo apresenta uma ideia geral de como isso poderia ser feito:

Exemplo 7.13 – O script 'menu.sh'.

```
#!/usr/bin/env bash

# Define a string do menu...
menu='CARDÁPIO DE PIZZAS

a - Calabresa
b - Muçarela
c - Vegana

s - sair
'

# Loop infinito...
while :; do
    # Limpa o terminal...
    clear

    # Exibe a string do menu...
    echo "$menu"
```

```
    # Aguarda a opção do utilizador...
    read -n1 -p 'Escolha um sabor: ' opt

    # Apenas quebra uma linha...
    echo

    # Decide o que fazer com a escolha...
    case ${opt,,} in
        a) echo 'Você escolheu calabresa';;
        b) echo 'Você escolheu muçarela';;
        c) echo 'Você escolheu vegana';;
        s) break;;
        *) continue;;
    esac

    # Aguarda a digitação de um caractere para seguir...
    read -n1 -p 'Tecle algo para continuar...'
done

# Sai do script...
exit 0
```

Quando executado, este é o resultado:

```
CARDÁPIO DE PIZZAS

a - Calabresa
b - Muçarela
c - Vegana

s - sair

Escolha um sabor: _
```

Agora, tente analisar cada linha do código, experimente fazer modificações e pense em alguma aplicação para um menu deste tipo – a melhor forma de aprender a programar em Bash é programando!

7.3.8 – O menu 'select'

O comando composto `select` (que não faz parte do shell padrão POSIX) é uma estrutura especializada na criação de menus rápidos a partir de listas de palavras. Sua sintaxe é idêntica à do *loop* `for`:

```
select VAR [in PALAVRAS]; do BLOCO DE COMANDOS; done
```

Neste aspecto, aliás, tudo que se aplica ao *loop* `for` é aplicável ao `select`, o que muda é o funcionamento da estrutura como um todo. Para começar, o `select` se comporta exatamente como o *loop infinito* do script `menu.sh`, no tópico anterior, só que ele espera uma entrada do usuário sem a necessidade de um comando `read`. A opção do utilizador é armazenada em duas variáveis: a string correspondente à `PALAVRA` vai para a variável `VAR`, e o *número da opção* escolhida vai para a variável interna `REPLY`. Deste modo, tudo que temos que fazer é escrever a lógica para tratar as opções digitadas pelo utilizador. Veja no script abaixo:

Exemplo 7.14 – O script 'menu-select.sh'.

```
#!/usr/bin/env bash

titulo='CARDÁPIO DE PIZZAS
------------------'

# Usar uma array é uma boa ideia...
menu=(Calabresa Muçarela Vegana Sair)

# Limpa o terminal...
clear

# Exibe o título...
echo "$titulo"

# Monta menu com os elementos da array...
select opt in ${menu[@]}; do
```

```
    # Sai do loop se escolher a opção igual ao
    # número de elementos da array 'menu'...
    [[ $REPLY == ${#menu[@]} ]] && break

    # Qualquer opção maior do que o número de
    # elementos da array é inválida...
    [[ $REPLY -gt ${#menu[@]} ]] && {
        # Informa que a opção é inválida...
        echo 'Opção inválida!'
    } || {
        # Exibe o número e a string da escolha...
        echo "Você escolheu $REPLY: ${opt,,}"
    }

    # Aguarda a digitação de um caractere para seguir...
    read -n1 -p 'Tecle algo para continuar...'

    # Limpa o terminal e exibe o título novamente...
    clear
    echo "$titulo"
done

# Sai do script...
exit 0
```

O script só ficou longo por conta dos comentários, mas ele exemplifica bem p uso típico do menu `select`.

Quando executado, este será o resultado:

```
CARDÁPIO DE PIZZAS
------------------
1) Calabresa
2) Muçarela
3) Vegana
4) Sair
#? _
```

Como você pode ver, o próprio `select` cuidou de numerar as opções, e foi por isso que nós tomamos o cuidado de criar a lista de palavras em um vetor onde o último elemento era a opção `Sair`. Deste modo, o número associado a ele pelo select corresponde ao número total de elementos do vetor, o que nos permite testar facilmente quando o utilizador escolheu sair do menu:

```
# Sai do loop se escolher a opção igual ao
# número de elementos da array 'menu'...
[[ $REPLY == ${#menu[@]} ]] && break
```

Se você não entendeu o que significa `${#menu[@]}`*, leia novamente o tópico 3.5 – Número de elementos e de caracteres.*

7.3.9 – O prompt PS3

Se você executou o script `menu-select.sh`, deve ter ficado intrigado com os caracteres `#?` exibidos no prompt do menu select, que é um dos quatro prompts do shell, cada um com a sua respectiva variável de ambiente:

- A variável `PS1` contém a string do prompt da linha de comandos.
- A variável `PS2` contém o caractere `>`, que é exibido quando digitamos comandos em várias linhas no terminal.
- A variável `PS4`, por sua vez, contém o caractere `+`, que aparece quando estamos *"debugando"* a execução do shell (comando `set -x`).
- E, finalmente, a variável `PS3` contém o prompt do menu `select` que, por padrão é `#?`.

Então, se quisermos um prompt mais amigável no nosso script, basta incluir uma linha definindo a string do prompt `PS3` em qualquer ponto *antes* da estrutura do menu:

```
# Define o prompt PS3...
PS3='Escolha a sua opção: '

# Monta menu com os elementos da array...
select opt in ${menu[@]}; do
```

E o resultado será...

```
CARDÁPIO DE PIZZAS
------------------
1) Calabresa
2) Muçarela
3) Vegana
4) Sair
Escolha a sua opção: _
```

8 – Funções

Se estivéssemos falando de qualquer outra linguagem de programação, uma função seria basicamente um bloco de instruções agrupadas e identificadas por um nome pelo qual pude ser chamada de outras partes no código – mas o Bash consegue nos surpreender até nisso!

A ideia básica é a mesma, ainda é um bloco de comandos nomeado, só que este "bloco de comandos", no Bash, é *qualquer um dos comandos compostos* que nós vimos no capítulo anterior! Logo, a definição de função no Bash é:

> *Qualquer 'comando composto' que recebe um nome pelo qual possa ser chamado de qualquer parte do código.*

Nós podemos criar funções com a sintaxe:

```
NOME() COMANDO-COMPOSTO
```

Alternativamente, pode ser utilizada a sintaxe abaixo com a palavra reservada `function`:

```
function NOME { LISTA DE COMANDOS; }
```

> ***Relembrando:*** *as funções devem ser definidas no script em um ponto anterior ao ponto onde são chamadas.* ***Agora, a novidade:*** *as funções só são executadas quando chamadas, logo, elas podem chamar outras funções definidas antes ou depois de suas próprias definições.*

8.1 – Criando funções com outros comandos compostos

É possível transformar qualquer comando composto em uma função, mas o agrupamento com chaves é a opção mais comum:

```
diga_oi() { echo oi; }
```

Aliás, muitos dos conceitos sobre funções que aprendemos por aí são apenas consequências do uso do agrupamento com chaves no corpo da função, como o escopo das variáveis e o fato da lista de comandos ser executada na mesma sessão do shell. Uma função criada com um *agrupamento com parêntesis*, por exemplo, é perfeitamente válida, mas a sua lista de comandos seria executada em um subshell e exigiria um tratamento totalmente diferente.

De qualquer forma, aqui estão alguns exemplos de uso de outros comandos compostos como funções – alguns são mais úteis do que outros, mas todos são muito interessantes.

8.1.1 – Funções com agrupamentos em parêntesis

Observe...

```
:~$ minha_func() (x=banana; echo $x)
:~$ minha_func
banana
:~$ echo $x

:~$
```

Repare que a função adota o comportamento do comando composto utilizado, e o agrupamento com parêntesis tem a característica de executar a lista de comandos *em um subshell*. Logo, as variáveis em seu interior estarão sempre limitadas ao escopo do subshell e não poderão ser acessadas pela sessão do

shell do script. Contudo, ainda podemos passar valores da sessão corrente do terminal ou do nosso script para os parêntesis:

```
:~$ minha_func() (echo $x)
:~$ x=elefante
:~$ minha_func
elefante
```

> *O subshell recebe uma cópia de todo o ambiente da sessão mãe, lembra?*

8.1.2 – Testes nomeados

```
:~$ teste() [[ ${1,,} == a* ]]
:~$ teste abacate; echo $?
0
:~$ teste melão; echo $?
1
```

Aqui, nós estamos testando se a string passada como primeiro argumento na chamada da função começa com a letra "a", o que é feito utilizando o operador de comparação de strings com padrões.

> *Nós falaremos mais sobre isso, mas todo argumento passado na chamada de uma função é recebida por ela como um parâmetro posicional (como se a função fosse um script dentro do nosso script).*

8.1.3 – Expressões aritméticas nomeadas

```
:~$ igual() (($1 == $2))
:~$ igual 3 4 && echo igual || echo desigual
desigual
```

```
:~$ igual 4 4 && echo igual || echo desigual
igual
```

Neste caso, estamos passando dois argumentos para a função, onde eles serão comparados com o operador de igualdade. Se forem iguais, o comando `(( ))` sairá com *sucesso*, e esta será a saída da função. Se forem diferentes, ela saíra com *erro*.

8.1.4 – Um loop for nomeado

```
:~$ lista( ) for n; do echo $n; done
:~$ lista {1..5}
1
2
3
4
5
:~$ lista *txt
alunos.txt
compras.txt
notas.txt
```

Se este não é o exemplo mais útil de uma função com outros comandos compostos, no mínimo, é o mais divertido! Repare que nós omitimos a parte do `in LISTA` da sintaxe do `for` para que ele capturasse todos os argumentos passados para a função.

8.1.5 – Um menu select nomeado

Já pensou em ter uma função geradora de menus no seu script?

```
#!/usr/bin/env bash

menu( ) select opt in "$@" Sair; do
```

```
 [[ $REPLY -eq $(($# + 1)) ]] && break
 [[ $REPLY -gt $(($# + 1)) ]] && {
 echo 'Opção inválida!'
 } || {
 echo $REPLY: $opt
 break
 }
 read -n1 -p 'Tecle algo para continuar...'
 clear
done
```

Para chamar a função no seu código, você faria assim, por exemplo:

```
PS3='Tecle a sua opção: '
menu banana laranja abacate
```

Quando o script fosse executado, este seria o resultado...

```
1) banana
2) laranja
3) abacate
4) Sair
Tecle a sua opção:
```

Como você pode ver, são muitas as possibilidades! Estes foram apenas alguns exemplos da flexibilidade e do poder que temos à disposição com as funções no Bash. Mas, está na hora de voltarmos à boa e velha função com comandos agrupados com chaves, porque esta é a forma mais genérica e mais flexível de se construir uma função.

8.2 – Sobre os nomes das funções

As regras para o `NOME` da função são as mesmas que utilizamos para criar nomes de variáveis, mas consideram-se boas práticas:

- Utilizar apenas caracteres minúsculos e o sublinhado.
- Separar nomes compostos com o sublinhado.
- Utilizar o sublinhado como primeiro caractere do `NOME` quando houver risco de confusão entre a chamada de uma função e um comando do shell.

> *Boas práticas são **recomendações** para a escrita de códigos legíveis sem o sacrifício do uso dos recursos que a linguagem nos oferece, mas elas não são absolutas e nem mandatórias.*

Por exemplo:

```
# Separando nomes compostos com sublinhado...
minha_func() { echo Olá, mundo!; }

# Iniciando o nome com o sublinhado...
_exit() { exit $1; }
```

Além disso, para facilitar a leitura do código, é conveniente manter um padrão ao longo de todo o script: se uma função tiver seu nome iniciado por um caractere sublinhado, todos os outros nomes também deverão ser iniciados com um sublinhado.

8.3 – Passagem de argumentos para funções

Como vimos no *tópico 2.3.15*, o shell substitui temporariamente o conteúdo dos parâmetros posicionais da sessão pelos valores dos argumentos passados na chamada de uma função, de modo que, no interior da função nós podemos utilizar qualquer uma de suas expansões. Quando a execução da função termina, os parâmetros posicionais da sessão são restaurados.

Acompanhe este exemplo com atenção e analise o que acontece:

```
:~$ set -- banana laranja
:~$ echo $1 $2
banana laranja
:~$ soma() { echo $1 + $2 = $(($1 + $2)); }
:~$ soma 10 15
10 + 15 = 25
:~$ echo $1 $2
banana laranja
```

Reparou que $1 e $2 continuaram expandindo seus valores originais na sessão do shell, mas assumiram outros valores durante a execução da função? Esse é o ponto mais importante, porque, como todas variáveis têm escopo global numa mesma sessão do shell, o esperado seria que os parâmetros posicionais tivessem o mesmo valor da sessão mãe, mas este é um comportamento exclusivo de funções. Se fosse apenas um agrupamento de comandos entre chaves, por exemplo, isso não aconteceria:

```
:~$ set -- banana laranja
:~$ { echo $1 + $2; }
banana + laranja
```

Os parâmetros especiais `*`, `@` e `#` também passam a conter todos valores e a quantidade (`#`) dos parâmetros posicionais passados para a função, mas o parâmetro especial `0` continua contendo o nome do shell ou do script.

8.4 – Escopo de variáveis em funções

Como vimos, fora o caso especial dos parâmetros posicionais, o escopo das variáveis dependerá da estrutura utilizada na formação da função. Ou seja, se o comando composto não abrir um subshell (como um agrupamento com parêntesis, por exemplo), nós podemos generalizar e dizer que a função tem

acesso a todos os parâmetros da sessão do shell e a sessão enxerga todas as variáveis na função – mas, isso pode ser mudado.

Eventualmente, pode não ser interessante criar situações em que uma variável na função possa ser acessada fora dela. Nesses casos, nós podemos contar com o comando interno `local`, que torna restrito ao contexto da função os escopos das variáveis que forem definidas com ele.

Observe e analise:

```
:~$ a=10; b=20
:~$ soma() { echo $((a + b)); }
:~$ soma
30
:~$ soma() { local a=15 b=35; echo $((a + b)); }
:~$ soma
50
:~$ echo $a $b
10 20
```

O comando interno `local`, que só funciona no corpo de funções, recebe nomes de variáveis e as atribuições como argumentos.

8.5 – Destruindo funções

A definição de uma função pode ser desfeita com a opção `-f` do comando interno `unset`:

```
:~$ soma() { echo $1 + $2 = $(( $1 + $2 )); }
:~$ soma 15 30
15 + 30 = 45
:~$ unset -f soma
:~$ soma 10 20
bash: soma: comando não encontrado
```

8.6 – Retorno de funções no Bash

Um dos pontos mais polêmicos sobre o como as funções funcionam no Bash, é o fato de que elas não possuem um conceito de retorno tal como nas outras linguagens de programação. A parte de *"devolver o controle da execução para o fluxo principal do programa"*, que consta desses outros conceitos, tudo bem, ela está lá. A dificuldade começa quando tentamos descobrir como podemos obter valores através do retorno de funções.

Mas a resposta é simples: *não podemos*.

Todos os dados que as funções processam, ou estão em variáveis globais, na saída padrão, ou em algum descritor de arquivos se as saídas dos comandos no corpo da função forem redirecionadas (*tópico 7.1*), o que sempre nos levará, ou a uma expansão, ou à exibição desses dados.

Por exemplo:

Exemplo 8.1 – Expandindo a saída de uma função.

```
:~$ quadrado() { echo $(($1**2)); }
:~$ echo O quadrado de 8 é $(quadrado 8)
O quadrado de 8 é 64
```

Para aumentar a confusão, existe ainda o comando interno `return`, que é quase um `exit`, só que para funções. Quando utilizado, seu papel é interromper a função e devolver o controle de execução para o comando seguinte no script.

Opcionalmente, nós podemos passar um valor inteiro entre 0 e 255 como argumento do `return` para especificar o estado de saída da função – se não for passado nenhum valor, a função sairá com o mesmo estado de saída do último comando executado.

8.7 – Obtendo o nome da função

Quando o shell "empresta" os parâmetros posicionais para uma função, o único que não muda é o parâmetro especial `0`, que continua contendo o nome do executável do shell ou o nome do script. Mas existe uma variável interna que armazena o nome de todas as funções em execução – o vetor `FUNCNAME`.

Sempre que uma função é executada, seu nome é armazenado no elemento de índice `0` de `FUNCNAME`. Os outros elementos contém os nomes das demais funções em execução em ordem inversa de chamada. Sendo assim, o maior índice de `FUNCNAME` em um script sempre terá o nome da função `main`., o que podemos comprovar com o pequeno script abaixo:

Exemplo 8.2 – Listando as funções em execução (funcname.sh).

```
#!/usr/bin/env bash

teste() {  echo ${FUNCNAME[@]}; }

teste
```

Quando executado...

```
:~$ ./funcname.sh
teste main
```

Isso possibilita algumas brincadeiras interessantes...

Vamos mudar um pouco o nosso script:

```
#!/usr/bin/env bash

goku() {  echo Oi, eu sou ${FUNCNAME[0]^}!; }

vegeta() { echo Eu sou o grande ${FUNCNAME[0]^}!; }
```

```
goku
vegeta
```

Executando...

```
:~$ ./funcname.sh
Oi, eu sou Goku!
Eu sou o grande Vegeta!
```

Repare que nós expandimos os nomes das funções com o caractere inicial em caixa alta.

Mas, nós podemos ir além e tentar descobrir quem chamou quem – afinal, funções também podem chamar funções:

```
#!/usr/bin/env bash

goku() {
      echo Oi, eu sou ${FUNCNAME[0]^}!
      echo Fala aí, Vegeta!
      vegeta
}

vegeta() {
      echo ${FUNCNAME[1]^}?!
      echo O que você quer com o grande ${FUNCNAME[0]^}?!
}

goku
```

Observe que, quando a função `vegeta` foi chamada, a função `goku` ainda estava em execução, logo, na função `vegeta`, ela passou a ser o elemento de índice `1` de `FUNCNAME`.

Vamos conferir na execução...

```
:~$ ./funcname.sh
Oi, eu sou Goku!
Fala aí, Vegeta!
Goku?!
O que você quer com o grande Vegeta?!
```

8.8 – Variáveis, aliases e funções

Na customização da linha de comandos, é muito comum nós criarmos aliases para abreviar a digitação ou padronizar as opções de um comando. Isso pode ser feito tanto com o comando `alias` (*tópico 4.1*) quanto com variáveis. Na prática, a única diferença é que um *alias* pode se executado sem um `$` na frente.

Repare:

```
# Criando o alias 'hh'
:~$ alias hh='help -d'
:~$ hh test
test - Evaluate conditional expression.

# Destruindo o alias 'hh'...
:~$ unalias hh

# Criando a variável 'hh'...
:~$ hh='help -d'
:~$ $hh test
test - Evaluate conditional expression.

# Destruindo a variável 'hh'...
:~$ unset hh
```

Como podemos ver, tanto o alias quanto a variável expandem para a string do comando. Mas falta uma coisa que pode ser bem mais útil na linha de comandos, que é a capacidade de receber argumentos – e isso só é possível com scripts ou funções. Como não vale a pena passar por todo o processo de

criação de scripts e gerenciamento de arquivos para fazer essas pequenas customizações, nós acabamos recorrendo às funções.

Então, se eu quiser um "comando" para calcular o quadrado de um inteiro qualquer, basta criar uma função `quadrado` e salvá-la no final do meu arquivo `~/.bashrc`. Assim, ela estará sempre disponível para quando eu precisar.

No arquivo `~/.bashrc`:

```
quadrado() { echo $(($1**2)); }
```

Executando (depois do shell recarregado)...

```
:~$ quadrado 6
36
```

9 – Uma mensagem (quase) final

Como você deve ter notado, são quase 300 páginas e nós mal arranhamos as possibilidades que o Bash nos oferece como linguagem. Em termos de conteúdo, o que está nesta primeira edição do nosso *Pequeno Manual* é mais do que suficiente para você começar a desenvolver programas e scripts profissionais em Bash. Contudo, eu espero que a minha maior contribuição para seu desenvolvimento como programador *bashista* tenha sido o incentivo à curiosidade e à pesquisa.

Há muito mais a ser descoberto, existem coisas dificílimas de encontrar – tanto nos manuais "oficiais" quanto nos tutoriais e outros conteúdos na internet, mas o esforço vale a pena. O Bash é um shell, um interpretador de comandos, e é uma linguagem de programação que permite o exercício da lógica e dos princípios da Filosofia Unix de desenvolvimento como nenhuma outra. É justo dizer, inclusive, que o Bash é uma das formas mais democráticas de aprender a programar e de fazer o seu sistema operacional trabalhar do jeito que você quer e precisa que ele trabalhe – não o contrário.

Esse é o tipo de liberdade que mais me encanta no Bash e nas ferramentas do sistema operacional GNU. Neste momento (quase) final, eu só espero ter conseguido passar um pouco dessa paixão para você.

Muito obrigado!

Blau Araujo
Osasco, 16 de novembro de 2020

Índice de Exemplos

Se você achou que encontraria um manual com todas as informações e exemplos de uso de todos os comandos e recursos do Bash, você entendeu errado o nome deste livro. Este é um manual do programador que utiliza os conceitos da Filosofia Unix, conforme implementada pelo Projeto GNU, para programar em Bash.

Com toda humildade de quem reconhece a sua própria estatura diante de gigantes, é isso que eu me proponho a fazer neste livro. Mais do que oferecer mais uma fonte de consulta para o seu arsenal de programador, o que eu realmente quero é convidá-lo para uma jornada de descobertas, onde cada novo conceito, cada comando, cada linha de código seja uma surpresa e um motivo a mais para você se apaixonar pela possibilidade de programar em Bash.