Curso Intensivo de Programação do Bash

Guia de estudos

Blau Araujo

Índice

Semana 1

Semana 2

Semana 3

Semana 4 (final)

Semana 1

1 – Sobre o curso

Este é um curso livre

- Livre em termos de conteúdo e de proposta.
- Licença: Creative Commons BY-SA 4.0.
- Proposta: que você seja livre para continuar aprendendo.
- Você pagou para que todos possam aprender gratuitamente.

Este não é um curso básico

- Pense em básico como em “base sólida”.
- Nada pode ser feito no Bash sem o que veremos.
- Nós veremos tudo que há para ser visto no Bash.
- Além disso, só desdobramentos e aplicações.

Comunidade de aprendizado contínuo

- Agora você faz parte de uma comunidade.
- Nossa plataforma estará sempre disponível para os inscritos.
- Não existem perguntas “básicas demais”.
- Restou alguma dúvida, continue perguntando.
- Não entendeu nada, faça o curso novamente.

Material de apoio

- Playlist do intensivão do Bash (Youtube).
- Playlist do Curso Básico de Programação do Bash (Youtube)
- Pequeno Manual do Programador GNU/Bash (livro em PDF)
- Curso Shell GNU (texto e vídeos)

2 – Sobre o shell

Interface padrão de sistemas unix-like

- Interface entre o usuário e o sistema operacional.
- Interpretador programável de comandos.
- Modos: interativo e não-interativo.

Modo interativo

- Sessão iniciada em um terminal.
- Comandos escritos no “prompt”.

Modo não-interativo

- Sessão iniciada em uma instância do shell.
- Comandos escritos em arquivos.
- Comandos passados como argumentos do shell.

O shell do GNU/Linux

- O Bash é o shell padrão do sistema operacional GNU.
- É um shell POSIX, mas vai além das especificações.
- Recursos de programação (linguagem?).

O Bash como linguagem

- Procedural, imperativa e estruturada.
- Tudo são comandos!
- Executada “*de cima para baixo*”
- Parâmetros (dados).
- Comandos compostos (estruturas).
- Expansões em vez de avaliações.

3 – O que há em um ‘olá mundo’?

Para criar um script

- Criar o arquivo;
- Editar e salvar o arquivo;
- Tornar o arquivo executável.

Criar o arquivo

O arquivo pode ser criado:

- Por um editor “*raw text*”;
- Diretamente na linha de comando (`touch`, `>` ou `>>`);

Editar e salvar o arquivo

Pode ser com qualquer editor *raw text* (texto plano):

- No terminal: `nano`, `vi`/`vim`/`neovim`, `micro`, etc.
- Na interface gráfica: Geany, Emacs, etc.

Tornar o arquivo executável

Utilitário `chmod`:

```
:~$ chmod +x nome-do-arquivo
```

O conteúdo do arquivo

Hashbang

A linha do interpretador de comandos (*shebang* ou *hashbang*) define qual será o programa utilizado para interpretar o conteúdo do arquivo. Sem ela, o shell em uso correntemente tentará interpretar o conteúdo do arquivo.

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

Ou…

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

Onde `#!`, quando acontece no exato início de um arquivo executável, é um comando para que o *kernel* execute o interpretador definido no caminho que aparecer em seguida.

> **Importante!** *Hashbangs* não são de uso exclusivo para scripts em shell/Bash. Várias linguagens interpretadas utilizam o mesmo recurso para invocar seus respectivos interpretadores.

Comandos

Tudo que vier nas linhas depois da *hashbang* será visto como uma **lista de comandos** a serem executados:

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

echo 'Olá, mundo!'
who
uptime -p
```

Executando o script:

```
:~$ ./meu-script
Olá, mundo!
blau     tty1         2022-08-01 09:21
up 4 days, 15 hours, 43 minutes
```

Comandos internos e externos

- **Comandos internos:** estão disponíveis como funcionalidades do próprio shell (*builtins*).
- **Comandos externos:** comandos que contém a invocação de outros programas.

Ajuda para comandos internos

```
:~$ help COMANDO
```

Exemplo:

```
:~$ help echo
```

Ajuda para outros programas

```
:~$ man PROGRAMA
```

Exemplo:

```
:~$ man bash
```

Parâmetros

Todos os dados disponíveis em uma sessão do shell são chamados de **parâmetros**, mas existe uma diferença de conceitos definida, essencialmente, de acordo com a forma como esses dados são identificados:

Nome: se o identificador contiver exclusivamente letras não acentuadas maiúsculas e minúsculas, sublinhado (`_`) e números (desde que não seja o primeiro caractere), o parâmetro será, conceitualmente, uma **variável**.

Números e símbolos gráficos: se o identificador contiver apenas um caractere numérico ou um símbolo gráfico, o parâmetro será chamado de **parâmetro especial**.

Resumindo

- Variáveis são parâmetros identificados por nomes.
- Parâmetros especiais são identificados por um dígito ou um símbolo gráfico.

Acesso

Em termos de acesso, os dados associados a ambos os tipos de parâmetros são obtidos através do mecanismo da **expansão**.

Expansões

No que diz respeito à manipulação de dados, o mecanismo da **expansão** é o mais importante (e impressionante) do shell/Bash.

Resumidamente, na expansão, o shell troca certas construções digitadas na linha do comando por outros dados **antes do seu comando ser executado**!

Passando dados para o script

Quando invocamos o script seguido de argumentos, todas as palavras na linha do comando serão registradas na sessão do script como parâmetros especiais.

```
:~$ ./meu-script banana laranja abacate
         ↑         ↑       ↑       ↑
         0         1       2       3
```

No script, esses parâmetros podem ser expandidos prefixando o cifrão (`$`) aos seus

identificadores:

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

echo $0 $1 $2 $3
```

Executando:

```
:~$ ./meu-script banana laranja abacate
./meu-script banana laranja abacate
```

Exercícios de fixação

> **Atenção:** pode haver várias respostas corretas.

1 – O que o shell não é?

☐ Uma interface.
☐ Uma linguagem de programação.
☐ Um compilador de programas.
☐ Um interpretador de comandos.

2 – Qual é o shell padrão do sistema operacional GNU?

☐ ash
☐ dash
☐ bash
☐ sh

3 – O sistema operacional gerencia programas em execução…

☐ Através da memória.
☐ Pela linha de comandos.
☐ Através de processos.
☐ Pelo programa `top`.

4 – O processo relativo à execução do shell define…

☐ Um script.
☐ Um programa.
☐ Um processo.
☐ Uma sessão.

5 – Uma nova sessão do shell pode ser inciada…

☐ Pela abertura de um terminal.
☐ Por *pipes* e agrupamentos com parêntesis.
☐ Pela invocação do executável do shell.
☐ Pela execução de um script.

6 – Um script é…

☐ Um programa.
☐ Uma lista de comandos previamente escritos em um arquivo.
☐ Executado na sessão do shell.
☐ Executado em uma nova sessão do shell.

7 – Parâmetros são…

☐ Dados associados a um script.
☐ Dados associados a uma sessão do shell.
☐ Identificados por *nomes*.
☐ Avaliados prefixando seus identificadores com `$`.

8 – Qual o jeito correto de criar um arquivo com recursos do shell?

☐ Utilitário `touch`.
☐ Redirecionamentos de escrita `>` e `>>`.
☐ Com um editor de textos (`nano`, por exemplo).
☐ Com o utilitário `cat`.

9 – Utilizando o comando `help` para descobrir, quais das listas abaixo contém apenas comandos internos (*builtins*)?

☐ `touch`, `ls`, `echo` e `cp`
☐ `read`, `echo`, `printf` e `ls`
☐ `type`, `echo`, `help` e `test`
☐ `read`, `echo`, `printf` e `mkdir`

10 – No shell, os dados associados e variáveis e parâametros podem ser acessados…

☐ Por expansão.
☐ Por referência.
☐ Por valor.
☐ Por avaliação.

11 – Um script só pode ser executado se…

☐ O arquivo tiver uma *hashbang* na primeira linha e permissão para execução.
☐ For um argumento na invocação de seu interpretador na linha de comandos.
☐ O arquivo tiver permissão para execução.
☐ O arquivo tiver uma *hashbang* na primeira linha.

12 – Descreva o que fazem os comandos abaixo e diga se são externos ou internos:

- `echo 'Salve, simpatia'`
- `read linha < arquivo.txt`
- `cat arquivo.txt`
- `help test`
- `touch arq1.txt arq2.txt`
- `>> arq3.txt`

13 – Pesquise e descubra quais ferramentas utilizar para:

- Listar os processos associados à sessão corrente do shell.
- Descobrir se o nome de um arquivo refere-se um arquivo comum ou um diretório.
- Listar os arquivos existentes no diretório corrente.
- Definir uma variável a partir da leitura da entrada padrão.

14 – Sabendo que a estrutura `$((VALOR1 + VALOR2))` expande a soma dos dois valores, complete o script abaixo para que ele receba dois argumentos na linha de comando da sua invocação e exiba sua soma.

Exemplo:

```
#!/bin/bash

FERRAMENTA-USADA $((VALOR1 + VALOR2))
```

15 – Explique por que os comandos abaixo produziram erros.

Comando 1

```
:~ $ var=casa amarela
bash: amarela: comando não encontrado
```

Comando 2

```
:~ $ 2var=banana
bash: 2var=banana: comando não encontrado
```

Comando 3

```
:~ $ $((3 + 4))
bash: 7: comando não encontrado
```

Semana 2

4 – Recebendo dados pela entrada padrão

Fluxos de dados padrão

Todo processo recebe três *descritores de arquivos* (*file descriptors*, FD) por padrão:

- FD `0` - Entrada padrão (stdin)
- FD `1` - Saída padrão (stdout)
- FD `2` - Saída padrão de erros (stderr)

> Em sistemas operacionais unix-*like*, tudo possui uma representação na forma de arquivo.

Redirecionamentos e pipes

Por padrão, os três fluxos de dados padrão estão associados a um dispositivo de terminal, o que pode ser alterado com os mecanismos de *redirecionamento* e *pipe*.

Redirecionamentos

O fluxo de dados é desviado de um **arquivo** para um processo (leitura) ou de um processo para um **arquivo** (escrita).

Lendo a primeira linha de um arquivo:

```
:~$ read linha < arquivo.txt
:~$ echo $linha
Primeira linha de arquivo.txt
```

Lendo os dados digitados pelo teclado:

```
:~$ read -p 'Digite seu nome: ' nome
Digite seu nome: Fulano da Silva
:~$ echo $nome
Fulano da Silva
```

> O builtin `read` lê uma linha da entrada padrão e a atribui a uma variável. Se nenhum nome de variável for informado, a linha será atribuída à variável do shell `REPLY`.

Escrevendo a saída padrão em um arquivo:

```
:~$ echo 'Uma linha de texto.' > arquivo.txt
:~$ cat arquivo.txt
Uma linha de texto.
```

> Com o operador de redirecionamento `>`, o conteúdo de arquivo é zerado antes da escrita.

Adicionando uma nova linha ao arquivo:

```
:~$ echo 'Outra linha de texto.' >> arquivo.txt
:~$ cat arquivo.txt
Uma linha de texto.
Outra linha de texto.
```

> O operador de redirecionamento `>>` adiciona o texto na saída ao final do arquivo (*append*).

Escrevendo a saída de erro em um arquivo:

```
:~$ ls banana 2> arquivo.txt
:~$ cat arquivo.txt
ls: não foi possível acessar 'banana': Arquivo ou diretório inexistente
```

> Quando o número do descritor de arquivos é omitido, o shell assume que serão `0` (entrada padrão) ou `1` (saída padrão). Por isso, quando queremos redirecionar a saída padrão de erros (stderr), nós informamos o número `2`, que é o seu descritor de arquivos.

Escrevendo ambas as saídas padrão no arquivo:

```
:~$ ls arquivo.txt banana &> arquivo.txt
:~$ cat arquivo.txt
ls: não foi possível acessar 'banana': Arquivo ou diretório inexistente
arquivo.txt
```

> No **Bash**, nós podemos utilizar os operadores `&>` e `&>>` para redirecionar ambas as saídas para um arquivo.

Pipes

Com os *pipes* (“canos”), um arquivo especial do tipo FIFO (*first in, first out*) é providenciado pelo sistema para *canalizar* o fluxo de dados entre processos:

```
:~$ echo 'Casa amarela' | grep -o 'am.*'
amarela
```

No exemplo, a saída do *builtin* `echo` foi canalizada para a entrada padrão do utilitário `grep`. **No Bash**, todos os comandos de uma pipeline são executados em seus respectivos subprocessos. Em outras palavras: eles não são executados na mesma sessão corrente do shell.

Here-docs e here-strings

O Bash oferece mais dois tipos de pipe: *here document* e *here string*.

Escrevendo várias linhas em um arquivo:

```
:~$ cat << FIM > arquivo.txt
> banana
> laranja
> abacate
> FIM
:~$ cat arquivo.txt
banana
laranja
abacate
```

O operador de **here-doc** (`<<`) permite a entrada de linhas de texto até que a palavra utilizada como delimitadora (no exemplo, `FIM`) seja encontrada. Neste momento, o `cat` (ainda no exemplo) é executado e imprime as linhas digitadas. Como a saída do `cat` está redirecionada para a escrita no arquivo `arquivo.txt` (operador `>`), ele é quem receberá as linhas de texto.

Acessando variáveis definidas em *pipes*:

```
:~$ echo zebra | read linha
:~$ echo $linha

:~$
```

No Bash, como todos os comandos de uma *pipeline* são executados em subprocessos, as variáveis ali criadas não poderão ser acessadas pela sessão corrente do shell.

```
:~$ read linha <<< 'zebra'
:~$ echo $linha
zebra
```

Em uma **here-string** (operador `<<<`), os comandos internos do shell são executados na sessão corrente e, portanto, todas as variáveis ali definidas estarão disponíveis.

Fluxos de dados e scripts

Os fluxos de dados, através dos redirecionamentos, *here-docs*, *here-strings* e *pipes*, podem ser utilizados em scripts para diversas finalidades e de várias formas:

- Criar, ler e escrever arquivos.
- Solicitar a entrada de dados.
- Receber dados provenientes de outros processos.

Exceto pela solicitação da entrada de dados, que só pode ser feita durante a execução do script, todo restante pode ser feito tanto na invocação quanto durante a execução do script.

Recebendo dados na invocação do script

Para receber dados (leitura) de outros processos na invocação do script, nós podemos utilizar os seguintes mecanismos na linha do comando:

```
# Lendo o conteúdo de um arquivo...
./meu-script < ARQUIVO

# Recebendo a saída de comandos...
COMANDO | ./meu-script

# Recebendo strings...
./meu-script <<< 'string literal ou expansão'

# Recebendo linhas de texto...
./meu-script << FIM
> ...
> FIM
```

Em todos esses casos, a entrada padrão do nosso script estará associada a um arquivo de pipe em vez do dispositivo de terminal.

Demonstração

Considere o script abaixo (`teste.sh`):

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

[[ -t 0 ]] && echo terminal || echo pipe ou redirecionamento
echo ---
ls -l /proc/$$/fd
```

A estrutura entre colchetes duplos (`[[`) é um **comando composto do Bash** que testa expressões assertivas: se a afirmação entre os colchetes for verdadeira, o comando terminará com **sucesso**; caso contrário, terminará com **erro**.

> Para mais informações sobre o que podemos testar e quais os operadores disponíveis, leia `help test` e `help [[`.

No exemplo, estamos utilizando o operador `-t` para **afirmar** que o descritor de arquivos `0` (entrada padrão) está associado a um terminal. Se o script for invocado sem pipes ou

redirecionamentos, o teste sairá com estado de **sucesso**, o que permitirá a impressão da string `terminal`. Depois dessa impressão, nós veremos a listagem do diretório `/proc/$$/fd`, que é o diretório dos descritores de arquivos associados ao processo do nosso script: nele, devemos observar o descritor de arquivos `0`.

Invocando o script sem pipes ou redirecionamentos:

```
:~$ ./teste.sh                                  dispositivo
terminal                                        de terminal
---                                                 |
total 0                                             V
lrwx------ 1 blau blau 64 ago 13 08:05 0 -> /dev/pts/0
lrwx------ 1 blau blau 64 ago 13 08:05 1 -> /dev/pts/0
lrwx------ 1 blau blau 64 ago 13 08:05 2 -> /dev/pts/0
lr-x------ 1 blau blau 64 ago 13 08:05 255 -> /home/blau/teste.sh
```

Se houver um redirecionamento de leitura na invocação, o resultado será esse:

```
:~$ ./teste.sh < arquivo.txt
pipe ou redirecionamento                          arquivo
---                                                  |
total 0                                              V
lr-x------ 1 blau blau 64 ago 13 08:09 0 -> /home/blau/arquivo.txt
lrwx------ 1 blau blau 64 ago 13 08:09 1 -> /dev/pts/0
lrwx------ 1 blau blau 64 ago 13 08:09 2 -> /dev/pts/0
lr-x------ 1 blau blau 64 ago 13 08:09 255 -> /home/blau/teste.sh
```

Repare que, desta vez, o descritor de arquivos `0` não está mais associado a um terminal, mas ao arquivo `arquivo.txt`.

Por último, se o script for invocado em uma *pipeline*…

```
:~$ echo banana | ./teste.sh              arquivo FIFO
pipe ou redirecionamento                   (um pipe)
---                                            |
total 0                                        V
lr-x------ 1 blau blau 64 ago 13 08:13 0 -> 'pipe:[74147]'
lrwx------ 1 blau blau 64 ago 13 08:13 1 -> /dev/pts/0
lrwx------ 1 blau blau 64 ago 13 08:13 2 -> /dev/pts/0
lr-x------ 1 blau blau 64 ago 13 08:09 255 -> /home/blau/teste.sh
```

> O resultado seria semelhante com *here-strings* ou *here-docs*.

No caso específico de *pipes*, nós podemos fazer o teste com o operador `-p` informando o dispositivo testado:

```
# Terminará com sucesso se for um 'pipe'...
[[ -p /dev/stdin ]]
```

Solicitando dados ao usuário

Nós também podemos solicitar dados pela entrada padrão em qualquer momento durante a execução do nosso script com o *builtin* `read`:

```
read -p 'Uma mensagem: ' VAR
```

> Mais opções do comando `read`, tal como implementado no bash, em `help read`.

5 – Estruturas condicionais

A lógica condicional do shell

- Condiciona a execução de comandos.
- Toda lógica condicional do shell é baseada em **estados de saída** ou pela comparação de expansões com padrões de texto (`case`).
- Na linha de um comando, não existem avaliações de valores boolianos *verdadeiro* ou *falso*.

Estados de saída

- **Sucesso:** comando termina com retorno zero (`0`).
- **Erro:** comando termina com qualquer retorno diferente de zero.
- Os estados de saída podem ser expandidos pelo parâmetro especial `?`.

Comandos internos

- `if` - Testa o estado de saída de comandos.
- `case` - Compara expansões e strings com padrões de texto.
- `while` - Estrutura de repetição condicionada ao estado de saída de um comando testado (continua se sucesso).
- `until` - Estrutura de repetição condicionada ao estado de saída de um comando testado (continua se erro).
- `[[ EXP ]]` - Avalia expressões assertivas.
- `test` e `[ EXP ]` - Avaliam expressões assertivas (POSIX).

> Excetuando-se os comandos `test` e `[ EXP ]`, os demais são **comandos compostos**.

Comando composto 'if'

Sintaxe:

```
if COMANDO_TESTADO_1; then
    BLOCO DE COMANDOS SE SUCESSO
elif COMANDO_TESTADO_2; then
    BLOCO DE COMANDOS SE SUCESSO
...
else
    BLOCO DE COMANDOS SE ERRO
fi
```

- As palavras reservadas `if` e `elif` avaliam o estado de saída de comandos.
- Todo `if` e `elif` deve ser seguido de um bloco iniciado pela palavra reservada `then`.
- Podemos utilizar tantos blocos `elif/then` quanto forem necessários.
- Apenas o bloco `if/then` é obrigatório na estrutura.
- Só pode haver um bloco `else` e ele é opcional.

Comando composto 'case'

Sintaxe:

```
case STRING in
    PADRÃO_1) BLOCO DE COMANDOS;;
    PADRÃO_2) BLOCO DE COMANDOS;;
    ...
    PADRÃO_N) BLOCO DE COMANDOS;;
esac
```

- Compara uma string, literal ou resultante de uma expansão, com os padrões das cláusulas.
- Os padrões podem ser strings literais, descrições de strings com os mesmos *curingas* utilizados para descrever nomes de arquivos ou globs estendidos.
- Com o operador de controle `;;`, caso o padrão corresponda à descrição da string testada, seu respectivo bloco de comandos será executado e o case terminará.

Operadores de controle do `case`:

- `;;` - Delimita o fim de um bloco de comandos, causando o término das comparações de padrões.
- `;&` - Delimita o fim de um bloco de comandos, mas indica que o bloco de comandos da cláusula seguinte também deve ser executado incondicionalmente.
- `;;&` - Delimita o fim de um bloco de comandos, mas indica que os padrões de todas as cláusulas subsequentes também devem ser testados.

Operadores de encadeamento condicional

- `&&` - Permite a execução do comando seguinte se o último comando executado sair com sucesso.
- `||` - Permite a execução do comando seguinte se o último comando executado sair com erro.

Exercícios de fixação

1 – Tendo como base o script `calc.sh` (na página final destes exercícios), descreva as linhas abaixo:

```
[[ $# -eq 3 ]] || die 1
```

Descrição:

```
[[ $1 =~ ^(0|[1-9][0-9]*)$ && $3 =~ ^(0|[1-9][0-9]*)$ ]] || die 2
```

Descrição:

```
[[ $2 = '/' && $3 -eq 0 ]] && die 3
```

Descrição:

```
[[ $2 == [+-%/x^] ]] || die 4
```

Descrição:

2 – As linhas descritas fazem referência à função `die`: o que justificou sua criação?

Resposta:

3 – Ainda observando a função `die`, descreva o seu funcionamento.

```
die() {
    echo ${msg[$1]}
    echo ${msg[0]}
    exit $1
}
```

Resposta:

4 – A função `die` utiliza a expansão de uma variável de nome `msg`: como podemos classificar esse tipo de variável?

Resposta:

5 – Por que precisamos converter os operadores de multiplicação (`*`) e potenciação (`**`) na estrutura `case` abaixo?

```
case $2 in
    x) operador='*' ;;
    ^) operador='**' ;;
    *) operador=$2 ;;
esac
```

Resposta:

6 – Crie um script chamado `calc2.sh` que, em vez de receber a expressão na invocação do script, solicite interativamente cada um dos termos da operação na seguinte ordem:

- Operador
- Operando 1
- Operando 2

7 – Crie um script chamado `calc3.sh` que, em vez de receber 3 argumentos na sua invocação, receba apenas um, correspondendo à operação a ser efetuada no seguinte formato:

```
:~$ ./calc3.sh 'OPERADOR OPERANDO1 OPERANDO2'
```

8 – Torne o script `calc3.sh` capaz de receber, opcionalmente, a string da operação através de pipes e redirecionamentos, conforme os exemplos abaixo:

```
:~$ ./calc3.sh '** 12 2'
144

:~$ ./calc3.sh <<< '+ 3 7'
7

:~$ echo '* 5 10' | ./calc3.sh
50

:~$ cat arquivo.txt
/ 24 3
:~$ ./calc3.sh < arquivo.txt
8
```

Script `calc.sh`:

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

# Definição de variáveis -----------------------------------------------------

msg[0]='Uso: calculadora VALOR1 OPERADOR VALOR2'
msg[1]='Número incorreto de argumentos (3).'
msg[2]='Operandos inválidos (apenas inteiros).'
msg[3]='Divisão por zero.'
msg[4]='Operador inválido.'

# Definição de funções -------------------------------------------------------

die() {
    echo ${msg[$1]}
    echo ${msg[0]}
    exit $1
}

# Seção principal (main) -----------------------------------------------------

# Validações...

[[ $# -eq 3 ]] || die 1

[[ $1 =~ ^(0|[1-9][0-9]*)$ && $3 =~ ^(0|[1-9][0-9]*)$ ]] || die 2

[[ $2 = '/' && $3 -eq 0 ]] && die 3

[[ $2 == [+%/x^-] ]] || die 4

# Conversão de operadores...

case $2 in
    x) operador='*' ;;
    ^) operador='**' ;;
    *) operador=$2 ;;
esac

# Impressão da saída...

echo $(($1 $operador $3))
```

> Este é um script didático desenvolvido colaborativamente pelos alunos da turma de agosto/2022 do *Curso Intensivo de Programação do Bash*.

Semana 3

6 – Parâmetros especiais

- Parâmetros são todos os dados em uma sessão do shell.
- Quando esses dados recebem *nomes*, eles são chamados de **variáveis**.
- Quando são identificados por números ou símbolos gráficos, são chamados de **parâmetros especiais**.
- Parâmetros especiais podem expandir dados sobre argumentos passados para scripts e funções, informações sobre o sistema e até o estado de saída de comandos.

Manipulação de argumentos

Parte dos parâmetros especiais expandem dados e informações sobre os argumentos passados para scripts e funções.

Parâmetro	Nome	Descrição
`$0`...`$n`	Parâmetros posicionais	Expandem cada uma das palavras passadas como argumentos em uma linha de comando segundo sua ordem de aparição. O primeiro parâmetro, `$0`, sempre receberá o nome do executável que deu início à sessão do shell.
`$#`	Quantidade de argumentos	Expande um inteiro correspondente à quantidade de argumentos passados para a sessão do shell, o que **não inclui** na contagem o parâmetro `$0`.
`$*`	Todos os argumentos	Expande uma lista de todos os argumentos passados para a sessão do shell. Entre aspas, expande tudo como uma palavra só com os argumentos separados pelo primeiro caractere definido na variável `IFS`.
`$@`	Todos os argumentos	Sem aspas, não há diferença da expansão `$*`. Entre aspas, porém, a expansão de `$@` resultará em todos os argumentos separados segundo as regras de citação aplicadas.
`$_`	Último argumento do comando anterior	Expande o último parâmetro posicional do último comando executado: que pode ser, inclusive, o valor em `$0`.

Informações sobre a sessão do shell

Também existem os parâmetros especiais que expandem informações sobre a sessão do shell e o estado de saída de comandos.

Parâmetro	Nome	Descrição
`$-`	Parâmetros do shell	Expande todas as opções ativas (flags) passadas na inicialização do shell.
`$$`	PID da sessão	Expande o identificador do processo (PID) da sessão corrente do shell.
`$!`	PID do job	Expande o PID do último processo executado em segundo plano na sessão.
`$?`	Estado de saída	Expande o estado de saída do último comando executado.

7 – Expansões

As expansões são o mecanismo de transformação da linha de comando digitada a partir da ocorrência de certas palavras e símbolos (*tokens*).

Expansões do shell

- Apelidos (*aliases*)
- Chaves
- O til (`~`)
- Parâmetros (inclusive as variáveis)
- Substituições de comandos
- Expansões aritméticas
- Substituições de processos
- Caracteres de controle ANSI-C
- Padrões de nomes de arquivos

Expansão de apelidos

- Ocorre na primeira etapa de processamento da linha do comando.
- São criados e exibidos com o *builtin* `alias`.
- São listados com o comando `alias -p`.
- São destruídos com o *builtin* `unalias` ou com o fim da sessão.
- São nomes associados a strings que expressam comandos completos.

Expansão de chaves

- É a primeira expansão da terceira etapa de processamento da linha do comando.
- Expande uma lista de palavras opcionalmente acrescidas de um prefixo e/ou um sufixo.
- As palavras expandidas podem vir de uma lista ou da expressão de uma sequência.
- A construção da expansão de chaves deve ser feita numa única palavra.

Sintaxe geral

```
prefixo{palavra1,palavra2,...,palavraN}sufixo

prefixo{inicio..término..salto}sufixo
```

Expandindo listas de palavras

```
:~$ echo {banana,laranja,abacate}
banana laranja abacate
:~$ echo {banana,laranja,abacate}.txt
banana.txt laranja.txt abacate.txt
:~$ echo arquivo-{banana,laranja,abacate}.txt
arquivo-banana.txt arquivo-laranja.txt arquivo-abacate.txt
```

Expandindo sequências numéricas

```
:~$ echo {1..5}
1 2 3 4 5
:~$ echo a{1..5}
a1 a2 a3 a4 a5
:~$ echo a{1..5}b
a1b a2b a3b a4b a5b
```

Expandindo sequências de caracteres (ASCII)

```
:~$ echo {a..e}
a b c d e
```

Combinando sequências e listas

```
:~$ echo {1..5}{a..c}
1a 1b 1c 2a 2b 2c 3a 3b 3c 4a 4b 4c 5a 5b 5c
:~$ echo {1..5}{a,z}
1a 1z 2a 2z 3a 3z 4a 4z 5a 5z
```

Expansão do til

Expande o caminho completo do diretório de um usuário.

Sintaxe geral

```
~        # Expande o caminho do diretório do usuário corrente.
~USUÁRIO # Expande o caminho do diretório de USUÁRIO.
~+       # Expande o caminho atual (PWD).
~-       # Expande o último diretório visitado (OLDPWD).
```

Expandindo o caminho do diretório do usuário corrente

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

Expandindo o caminho do diretório de outro usuário

```
:~$ echo ~helena
/home/helena
```

Expansão de parâmetros

- Expande o dado associado ao identificador de um parâmetro.
- Podem ser parâmetros especiais, variáveis ou vetores.
- Permite diversos tipos de modificações dos dados expandidos.

Expansões básicas

Expansões que não implicam em alterações nos dados expandidos.

Expandindo variáveis

```
$NOME ou ${NOME} # Expande o dado atribuído à variável NOME.
```

Exemplo:

```
:~$ var=banana
:~$ echo $var
banana
```

Expandindo elementos de vetores

```
${NOME[N]} # Expande o elemento N do vetor NOME.
```

Exemplo:

```
:~$ vetor=(zebra tigre elefante)
:~$ echo ${vetor[1]}
tigre
```

Expandindo todos os elementos de vetores

```
${NOME[@]} ou ${NOME[*]} # Expande todos os elementos do vetor NOME.
```

Exemplo:

```
:~$ vetor=(zebra tigre elefante)
:~$ echo ${vetor[@]}
```

```
zebra tigre elefante
```

Expandindo os índices de vetores

```
${!NOME[@]} ou ${!NOME[*]} # Expande todos os índices do vetor NOME.
```

Exemplo:

```
:~$ vetor=(zebra tigre elefante)
:~$ echo ${!vetor[@]}
0 1 2
```

Expandindo indireções

```
${!NOME} # Expande a variável identificada
         # com a expansão da variável NOME.
```

Exemplo:

```
:~$ bicho=cavalo
:~$ cavalo=equino
:~$ echo ${!bicho}
equino
```

Expandindo nomes de variáveis

```
${!PREF@} ou ${!PREF*} # Expande todos os nomes de variáveis
                       # iniciadas com o prefixo PREF.
```

Exemplo:

```
:~$ echo ${!BASH*}
BASH BASHOPTS BASHPID BASH_ALIASES BASH_ARGC BASH_ARGV BASH_ARGV0
BASH_CMDS BASH_COMMAND BASH_COMPLETION_VERSINFO BASH_LINENO
BASH_REMATCH BASH_SOURCE BASH_SUBSHELL BASH_VERSINFO BASH_VERSION
```

Expandindo quantidades de caracteres

```
${#NOME} # Expande o total de caracteres do dado
         # atribuído à variável NOME.
```

Exemplo:

```
:~$ var=abacate
:~$ echo ${#var}
7
```

Expandindo a quantidade de elementos

```
${#NOME[@]} ou ${NOME[*]} # Expande a quantidade de elementos
                          # do vetor NOME.
```

Exemplo:

```
:~$ vetor=(zebra tigre elefante)
:~$ echo ${#vetor[@]}
3
```

Expandindo a quantidade de caracteres de um elemento

```
${#NOME[N]} # Expande a quantidade de caracteres
            # do elemento N do vetor NOME.
```

Exemplo:

```
:~$ vetor=(zebra tigre elefante)
:~$ echo ${#vetor[2]}
8
```

Alterando a caixa de texto

Transformação em maiúsculos

```
${NOME^}  # Torna maiúsculo o primeiro caractere expandido.
${NOME^^} # Torna maiúsculos todos os caracteres expandidos.
```

Exemplo:

```
:~$ var=banana
:~$ echo ${var^}
Banana
:~$ echo ${var^^}
BANANA
```

Transformação em minúsculos

```
${NOME,}  # Torna minúsculo o primeiro caractere expandido.
${NOME,,} # Torna minúsculos todos os caracteres expandidos.
```

Exemplo:

```
:~$ var=ZEBRA
:~$ echo ${var,}
zEBRA
```

```
:~$ echo ${var,,}
zebra
```

Alternando a caixa de texto

```
${NOME~}  # Inverte a caixa do primeiro caractere expandido.
${NOME~~} # Inverte a caixa de todos os caracteres expandidos.
```

Exemplo:

```
:~$ var=ZeBrA
:~$ echo ${var~}
zeBrA
:~$ echo ${var~}
zEbRa
```

Expansões condicionais

Expandem valores de forma condicionada à definição e/ou ao dado atribuído a um parâmetro.

Expandindo um valor padrão

```
${NOME:-STRING} # Expande STRING se NOME for nulo ou indefinido.
${NOME-STRING}  # Expande STRING apenas se NOME for indefinido.
```

Exemplo:

```
:~$ echo ${var:-banana}
banana
:~$ var=laranja
:~$ echo ${var:-banana}
laranja
```

Expandindo um valor alternativo

```
${NOME:+STRING} # Expande STRING se NOME não for nulo nem indefinido.
${NOME+STRING}  # Expande STRING apenas se NOME não for indefinido.
```

Exemplo:

```
:~$ echo ${var:+banana}

:~$ var=laranja
:~$ echo ${var:-banana}
banana
```

Atribuindo um valor padrão

```
${NOME:=STRING} # Expande e atribui STRING se NOME for nulo ou indefinido.
${NOME=STRING}  # Expande e atribui STRING apenas se NOME for indefinido.
```

Exemplo:

```
:~$ echo $var

:~$ echo ${var:=banana}
banana
:~$ echo $var
banana
```

Expandindo uma mensagem de erro

```
${NOME:?STRING} # Causa um erro e emite a mensagem em STRING
                # se NOME for nulo ou indefinido.
${NOME?STRING}  # Causa um erro e emite a mensagem em STRING
                # apenas se NOME for indefinido.
```

Exemplo:

```
:~$ echo $var

:~$ echo ${var:?Valor nulo ou variável indefinida}
bash: var: Valor nulo ou variável indefinida
```

Aparando início ou fim da expansão

A partir da descrição de um padrão, nós podemos remover o início ou o fim de uma expansão.

Aparando o início da expansão

```
${NOME#PADRÃO}  # Remove a menor ocorrência de PADRÃO do início da expansão.
${NOME##PADRÃO} # Remove a maior ocorrência de PADRÃO do início da expansão.
```

Exemplos:

```
:~$ var='https://blauaraujo.com'
:~$ echo ${var#*/}
/blauaraujo.com
:~$ echo ${var##*/}
blauaraujo.com
```

Aparando o fim da expansão

```
${NOME%PADRÃO}  # Remove a menor ocorrência de PADRÃO do fim da expansão.
${NOME%%PADRÃO} # Remove a maior ocorrência de PADRÃO do fim da expansão.
```

Exemplos:

```
:~$ var='https://blauaraujo.com/baixar'
:~$ echo ${var%/*}
https://blauaraujo.com
:~$ echo ${var%%/*}
https:
```

Expandindo substrings e faixas de elementos

Se considerarmos os dados associados a parâmetros e variáveis como cadeias de caracteres, nós podemos expandir apenas o trecho especificado pelo caractere inicial até uma quantidade de caracteres. Se a variável for um vetor, também poderemos expandir apenas os elementos de um índice inicial até uma dada quantidade de elementos.

Expandindo substrings

```
${VAR:INÍCIO:QUANTIDADE}  # Expande caracteres de INÍCIO até QUANTIDADE.
${VAR:-INÍCIO:QUANTIDADE} # Expande a QUANTIDADE de caracteres a partir
                          # de um INÍCIO contado a partir do fim da expansão.
${VAR:INÍCIO:-FIM}        # Expande caracteres entre INÍCIO e um FIM
                          # contado a partir do final da expansão.
```

Exemplos:

```
:~$ var=abcdefghji
:~$ echo ${var:2:4}
cdef
:~$ echo ${var:0:3}
abc
:~$ echo ${var::3}
abc
:~$ echo ${var:3}
defghij
:~$ echo ${var:(-5):3}
fgh
:~$ echo ${var:5:-2}
fgh
```

Expandindo faixas de elementos

```
${VAR[@]:INÍCIO:QUANTIDADE} # Expande elementos de INÍCIO até QUANTIDADE.
```

Exemplos:

```
:~$ vetor=(a b c d e f g h j i)
:~$ echo ${vetor[@]:2:4}
c d e f
:~$ echo ${vetor[@]:0:3}
a b c
:~$ echo ${vetor[@]::3}
a b c
:~$ echo ${vetor[@]:(-5):3}
f g h
:~$ echo ${vetor[@]:5:-2}
bash: (-2): expressão de substring < 0
```

Busca e substituição

As expansões também possibilitam a busca e substituição de padrões de texto nos dados expandidos.

Substituindo a primeira ocorrência do padrão

```
${VAR/PADRÃO/STRING} # Substitui a primeira ocorrência de PADRÃO por STRING.
```

Exemplo:

```
:~$ var=banana
:~$ echo ${var/na/ca}
bacana
```

Substituindo todas as ocorrências do padrão

```
${VAR//PADRÃO/STRING} # Substitui todas as ocorrência de PADRÃO por STRING.
```

Exemplo:

```
:~$ var=banana
:~$ echo ${var/na/ta}
batata
```

Ancorando o padrão da busca

```
${VAR/#PADRÃO/STRING} # Substitui PADRÃO encontrado no início da expansão.
${VAR/%PADRÃO/STRING} # Substitui PADRÃO encontrado no final da expansão.
```

Exemplo:

```
:~$ var=cavalo
:~$ echo ${var/#???/rob}
robalo
```

```
:~$ echo ${var/%???/eira}
caveira
```

Adicionando prefixos e sufixos

Como efeito da ancoragem de um padrão vazio, nós podemos incluir prefixos e sufixos aos dados expandidos.

```
${VAR/#/STRING} # Adiciona STRING ao início da expansão.
${VAR/%/STRING} # Adiciona STRING ao final da expansão.
```

Exemplo:

```
:~$ var=cavalo
:~$ echo ${var/#/meu }
meu cavalo
:~$ echo ${var/%/ dado}
cavalo dado
```

Substituição de comandos

- Ocorre na mesma etapa das expansões de parâmetros.
- Possibilita expandir as saídas produzidas por uma lista de comandos.
- A lista de comandos é executada em um subshell.

Sintaxe:

```
$(LISTA DE COMANDOS) # Sintaxe recomendada!
`LISTA DE COMANDOS`  # Sintaxe obsoleta!
```

Exemplo:

```
:~$ echo Meu nome de usuário é $(whoami).
Meu nome de usuário é blau.
```

Expansões aritméticas

- Ocorre na mesma etapa das expansões de parâmetros e substituições de comandos.
- Possibilita a expansão da avaliação de operações lógicas e aritméticas.
- É a forma POSIX de efetuar operações lógicas e aritméticas.
- Pode conter várias expressões separadas por vírgula, mas apenas a última avaliação será expandida.
- Mesmo que não sejam expandidas, as demais avaliações são computadas na sessão do shell.
- Só trabalha com dados numéricos inteiros.

- Em expansões aritméticas e no comando composto `((EXP))`, ao contrário de uma linha de comando, todos os dados literais e identificadores expressam valores.
- Nomes de variáveis não precedidos por `$` expressam o valor de seus dados.
- Strings expandidas a partir de parâmetros e variáveis são tratadas como nomes de variáveis.
- Se uma string não corresponder a uma variável definida ou se o dado associado não for um número inteiro, sua avaliação será zero (`0`).
- Variáveis ou dados literais que resultem em números com ponto flutuante causarão erros.

Sintaxe:

```
$((EXP1[, EXP2, ...]))
```

Exemplos:

```
:~$ echo 2 + 2 = $((v = 3 * 5, 2 + 2))
2 + 2 = 4
:~$ echo $v
15
:~$ echo $((++v))
16
:~$ var=numero
:~$ numero=25
:~$ echo $((var * 2))
50
:~$ numero=banana
:~$ echo $((var * 2))
0
```

Substituição de processos

- Ocorre na mesma etapa das expansões de parâmetros, expansões aritméticas e substituições de comandos.
- Faz com que as saídas do uma lista de comandos seja enviada para um descritor de arquivos pelo qual poderão ser lidas ou fornece um fluxo de entrada para a lista de comandos.
- Expande apenas um descritor de arquivos.
- A lista de comandos é executada em um subshell.

Sintaxe:

```
<(LISTA DE COMANDOS) # Expande o descritor de arquivos para leitura.
>(LISTA DE COMANDOS) # Expande o descritor de arquivos para escrita.
```

Expandindo o descritor de arquivos

```
:~$ echo <(echo Salve simpatia)
/dev/fd/63
```

Lendo os dados do descritor de arquivos

```
:~$ cat <(echo Salve simpatia)
Salve simpatia
```

Escrevendo na entrada padrão da lista de comandos

```
:~$ echo Salve simpatia > >(cat)
Salve simpatia
```

Expansão de caracteres ANSI-C

- Expande caracteres de controle expressos como sequências de escape.
- Excetuando-se as sequências de escape, tudo mais será expandido literalmente.
- Faz parte do conjunto de regras de citação, mas não é processada na primeira etapa de processamento de comandos.
- Na primeira etapa, será vista pelo shell apenas como aspas simples.

Sintaxe:

```
$'STRING CONTENDO SEQUÊNCIAS DE ESCAPE'
```

Exemplo:

```
echo $'banana\nlaranja'
banana
laranja
```

Expansão de nomes de arquivos

A ocorrência dos caracteres curinga `*` e `?` ou de uma lista de caracteres válidos `[...]` faz com que o shell tente expandir os nomes de arquivos que correspondam ao padrão descrito.

- Acontece depois das expansões de parâmetros.
- Mecanismo também conhecido como *glob*.
- Os curingas perdem seus significados especiais entre aspas.
- Quando utilizados em estruturas `case` e `[[`, ou em expansões de parâmetros, não expandem arquivos e passam a expressar padrões de texto.
- Quando curingas resultam de expansões de parâmetros, acontece a expansão de nomes de arquivos.
- Quando não há arquivos com nomes que correspondam ao padrão, o shell expande o próprio padrão.

Curingas

- `*` - Casa com zero ou mais caracteres quaisquer.
- `?` - Casa com exatamente um caractere qualquer.
- `[...]` - Casa com exatamente um dos caracteres da lista.
- `[!...]` - Casa com qualquer caractere, menos os listados.

Exemplos:

```
:~/docs$ ls
notas  arquivo.txt  planilha.ods  livro.pdf
:~/docs$ echo *
notas arquivo.txt planilha.ods livro.pdf
```

Expandindo apenas diretórios

Quando o padrão é seguido de uma barra (`/`), ele passa a descrever um diretório:

```
:~/docs$ echo */
notas/
```

8 – Ordem de processamento de comandos

- Antes de ser executada a linha de comando passa por cinco estágios de processamento.
- As expansões acontecem no terceiro estágio e também são processadas em uma ordem fixa.
- Ambas as ordens de processamento afetam o que funciona ou não numa linha de comando.

Etapas de processamento

- Decomposição da linha em palavras e operadores.
- Classificação de comandos em simples, complexos e compostos.
- Expansões
- Preparação dos redirecionamentos.
- Monitoração de estados de saída.

Etapa 1: decomposição da linha em palavras e operadores

Depois de expandir eventuais apelidos, o shell percorre todos os caracteres da linha do comando resultante. Quando um **operador** é encontrado, ele é registrado e todos os caracteres anteriores são registrados como uma **palavra**.

- Palavras são sequências de caracteres delimitadas por operadores.
- Operadores são os elementos da sintaxe compostos por um ou mais **metacarecteres**.
- Metacaracteres são caracteres com algum significado especial para o shell.
- São metacarecteres do shell: espaços, tabulações, quebras de linha, `|`, `&`, `;`, `(`, `)`, `<` e `>`.

Operadores de controle e de redirecionamento

O shell implementa dois tipos de operadores no contexto da linha do comando: **operadores de controle** e **operadores de redirecionamento**.

Operadores de controle

Operador	Descrição
Quebra de linha e `;`	Delimitam o fim de uma linha de comando e indicam que ela deve ser executada incondicionalmente.
`&&`	Encadeamento condicional se sucesso. Permite a execução do comando seguinte apenas se o último comando executado terminar com sucesso.
`||`	Encadeamento condicional se erro. Permite a execução do comando seguinte apenas se o último comando executado terminar com erro.

`&`	Execução assíncrona. Faz com que a linha do comando seja executada em segundo plano (background) em um job.
`|` e `|&`	Operadores de pipe. Fazem com que os dados na saída de um comando sejam passados para a entrada do comando seguinte. No Bash, o operador `
`;;`, `;&` e `;;&`	Operadores de controle das cláusulas do comando composto `case`.
`(` e `)`	Operadores de agrupamento. Agrupam uma lista de comandos e fazem com ela seja executada em um subshell.

Operadores de redirecionamento

Operador	Descrição
`[n]>`	Escrita. Escreve o fluxo de dados oriundo do 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 truncado antes da escrita; se não existir, o arquivo será criado.
`[n]>>`	Append. Escreve o fluxo de dados oriundo do 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 os fluxos de dados oriundos da saída padrão (stdout) e da saída de erros (* stderr*) em um arquivo. Comporta-se como o operador de escrita `>`.
`&>>`	Escreve os fluxos de dados oriundos da saída padrão (stdout) e da saída de erros (stderr) no final de um arquivo (append). Comporta-se como o operador de append.
`[n]<`	Leitura. Abre um arquivo para leitura no descritor de arquivos `n`. Se `n`for omitido, lê o descritor de arquivos `0` (stdin, entrada padrão).
`<<`	Here-document (ou here-doc). Envia uma ou mais linhas de texto delimitadas por uma palavra para a entrada padrão (stdin) de um processo.
`<<<`	Here-string. Envia uma string para a entrada padrão (stdin) de um processo.
`[n]>&-`	Fecha o descritor de arquivos `n`.
`>&[n]`	Faz com que a saída padrão seja uma cópia do descritor de arquivos `n`.
`<&[n]`	Faz com que a entrada padrão seja uma cópia do descritor de arquivos `n`.
`[n]<>`	Abre um arquivo para leitura e escrita no descritor de arquivos `n`.

Operador	Descrição
	Se `n` não for especificado, utiliza o descritor de arquivos 0 (stdin).
`[n]>&[m]`	Torna o descritor de arquivos `n` uma cópia do descritor de arquivos `m`.
`[n]>&[m]-`	Torna o descritor de arquivos `n` uma cópia do descritor de arquivos `m` e fecha o descritor de arquivos `m` (move `m` para `n`).

Regras de citação

- A separação de palavras obedece às regras de citação.
- O caractere `\` remove o significado especial do caractere seguinte.
- Todos os caracteres entre aspas simples (`'...'`) têm seus significados especiais removidos.
- Todos os caracteres entre aspas duplas (`"..."`) têm seus significados especiais removidos, exceto o acento grave (`` ` ``), o cifrão (`$`) e, de acordo com algumas condições, a exclamação (`!`) e a contrabarra (`\`).
- Tudo que vier depois de uma cerquilha (`#`) é ignorado pelo shell (comentário).

Expansão de apelidos

- Os apelidos (*aliases*) são expandidos na primeira etapa de processamento.
- Quando um apelido tem o mesmo nome de um comando ou de um programa, nós podemos evitar a expansão do apelido precedendo-o de uma contrabarra, o que fará com que o comando original seja executado.

Etapa 2: classificação de comandos

Tendo registrado as palavras e operadores encontrados, o shell classifica os comandos de três formas:

- **Comando simples:** de modo geral, a invocação de um comando ou de um programa, seus argumentos.
- **Comandos complexos:** são vários comandos simples organizados em listas encadeadas e/ou *pipelines*.
- **Comandos compostos:** são comandos delimitados ou iniciados por *palavras reservadas*.

Todos as classes de comandos podem participar de pipes e redirecionamentos.

> **Importante!** É neste estágio, que os operadores no contexto de comandos são identificados como tal, ou seja: eles não poderão ser resultado de expansões, visto que elas só serão processadas na etapa seguinte.

Palavras reservadas

A maior diferença entre os comandos simples e os comandos compostos é que, em vez de começarem com o nome de um comando interno do shell ou com o nome de um executável, os comandos compostos geralmente começam com palavras reservadas (*keywords*).

A menos que estejam citadas, as palavras abaixo serão identificadas como reservadas pelo shell:

```
if      then    elif    else    fi      time
for     in      until   while   do      done
case    esac    coproc  select  function
{       }       [[      ]]      !
```

Apesar dos parêntesis não serem palavras reservadas, eles também formam dois comandos compostos:

- `(COMANDOS)` - Agrupamento de comandos.
- `((EXPRESSÕES))` - Expressões lógicas e aritméticas.

Etapa 3: expansões

Todas as demais expansões ocorrem na terceira etapa de processamento na seguinte ordem:

1. Expansão de chaves.
2. Expansão do til.
3. Expansão de parâmetros, substituições de comandos, expansões aritméticas, substituições de processos e expansão de sequências de escape.
4. Separação das palavras resultantes das expansões anteriores.
5. Expansão de nomes de arquivos.
6. Remoção das aspas.

Etapa 4: redirecionamentos

Na quarta etapa de processamento, se a linha contiver operadores de redirecionamento, todos os recursos para leitura e escrita de fluxos de dados serão providenciados, inclusive a criação de arquivos, se for o caso.

Etapa 5: monitoração de estados de saída.

Se a linha do comando contiver operadores de encadeamento condicional ou comandos compostos que condicionam a execução de blocos de comandos ao estado de saída de um comando testado, o shell passará a monitorar esses estados de saída. Independente disso, porém, sempre haverá o registro dos estados de saída no parâmetro especial `?`.

Exercícios de fixação

Analise as informações abaixo e resolva os exercícios...

O Bash tem três estruturas de repetição: os loops `for`, `while` e `until`. Se considerarmos que o `for` oferece duas formas de controle das repetições, nós temos quatro tipos de estruturas de repetição:

- Loop `for` controlado por uma lista de palavras.
- Loop `for` controlado por uma expressão aritmética ternária (estilo C).
- Loop `while`, interrompido quando um comando testado sai com erro.
- Loop `until`, interrompido quando um comando testado sai com sucesso.

Suas sintaxes gerais são...

Loop 'for'

```
for VAR in LISTA_DE_PALAVRAS; do
    BLOCO_DE_COMANDOS
done [redirecionamentos]
```

Loop 'for' estilo C

```
for ((INÍCIO;CONTROLE;ATUALIZAÇÃO)); do
    BLOCO_DE_COMANDOS
done [redirecionamentos]
```

Onde:

- `INÍCIO`: geralmente, uma atribuição (exemplo: `i=0`);
- `CONTROLE`: uma condição de parada (exemplo: `i<10`);
- `ATUALIZAÇÃO`: uma alteração do valor controlado (exemplo: `i++`).

Loop 'while'

```
while COMANDO_TESTADO; do
    BLOCO_DE_COMANDOS
done [redirecionamentos]
```

Loop 'until'

```
until COMANDO_TESTADO; do
    BLOCO_DE_COMANDOS
done [redirecionamentos]
```

Comandos de controle de loops

Além dos controles das próprias estruturas, o Bash nos dá dois comandos internos para

controlar loops:

- `break`: interrompe a execução do bloco de comandos e sai do loop.
- `continue`: interrompe a execução do bloco de comandos e segue para o próximo loop.

1 - Repetindo até acertar

Crie uma estrutura de repetição para que o comando `read`, abaixo, seja executado até que o usuário informe um dos dados solicitados.

```
read -sN1 -p 'Digite A, B ou C: ' resposta
```

2 - Dados em sequência

No script `calc2.sh`, nós pedimos três informações ao usuário (operação, valor 1 e valor 2). Sem se preocupar com validações, escreva uma estrutura de repetição que solicite os mesmos dados e na mesma ordem abaixo, com apenas uma linha do comando `read` e os armazene em um vetor de nome `args`:

Original:

```
read -p 'Digite um operador: ' op
read -p 'Digite o primeiro número: ' n1
read -p 'Digite o segundo número: ' n2
```

3 - Contador

Utilizando o loop `while`, escreva um contador que imprima os números de `1` a `10`.

4 - Outro contador

Faça o mesmo do exercício anterior com o loop `for` estilo C.

5 - Renomeando em massa

Considerando o diretório abaixo, escreva um loop onde todos os arquivos terminados com `.txt` sejam impressos com a terminação `.md`.

Diretório:

```
:~$ echo *
aula1.txt aula2.md aula3.txt exemplo.png exercicios.txt
```

6 - Cada macaco no seu galho

Considerando os vetores `bicho` e `ambiente`, escreva um loop que imprima a saída abaixo:

```
deserto - camelo
mar - tubarão branco
água doce - perereca
pastos - cavalo
```

Vetores:

```
ambiente=(deserto mar 'água doce' pastos)
bicho=(camelo 'tubarão branco' perereca cavalo)
```

7 - Menu

A variável `menu` contém todas as opções de um menu que você deverá implementar utilizando o comando `read` e um loop `until`. Detalhe: antes da execução de cada uma das opções, o terminal precisará ser limpo com o comando `clear` e o utilizador terá que pressionar qualquer tecla para voltar ao menu.

Variável:

```
menu="\
1. Exibir a data e a hora correntes (date)
2. Exibir o tempo de login no sistema (uptime)
3. Exibir a listagem do diretório do usuário (ls -l ~)
Q. Sair do menu (tanto faz digitar 'q' ou 'Q')
"
```

Importante! Se qualquer tecla não definida no menu for pressionada, o terminal deverá ser limpo apresentando o menu novamente.

Semana 4 (final)

9 – Comandos compostos

Comandos compostos são estruturas do shell que permitem o agrupamento de comandos e avaliação de expressões. Existem quatro categorias de comandos compostos:

Agrupamentos de comandos

- Agrupamento com chaves
- Agrupamento com 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))`

Palavras reservadas

A maior diferença entre os comandos simples e os comandos compostos é que, em vez de começarem com o nome de um comando interno do shell ou com o nome de um executável, os comandos compostos geralmente começam com **palavras reservadas** (*keywords*).

A menos que estejam citadas, essas palavras serão identificadas como reservadas pelo shell:

```
if      then    elif    else    fi      time
for     in      until   while   do      done
case    esac    coproc  select  function
{       }       [[      ]]      !
```

> **Importante!** Palavras reservadas são palavras e, portanto, exigem que um operador de controle as separem de outras palavras.

Agrupamentos de comandos

- Agrupamentos são tratados como um único comando.
- Podem ser feitos com chaves ou parêntesis.
- O estado de saída de agrupamentos será o estado de saída do último comando executado.
- Permite capturar a saída de cada comando em um único fluxo de dados.

Por exemplo, o operador de redirecionamento `>` trunca o conteúdo do arquivo de destino cada vez que faz uma escrita:

```
:~$ echo banana > frutas.txt; echo laranja > frutas.txt
:~$ cat frutas.txt
laranja
```

Aqui, o arquivo `fruta.txt` foi zerado antes de cada redirecionamento da saída de um comando `echo`, o que fez com que seu conteúdo fosse apenas `laranja` ao final.

Observe o que acontece em um agrupamento com chaves:

```
:~$ { echo banana; echo laranja; } > frutas.txt
:~$ cat frutas.txt
banana
laranja
```

Com os comandos agrupados, existe apenas um fluxo de dados e o redirecionamento só é desfeito ao término da execução do último comando.

O mesmo poderia ser feito com parêntesis:

```
:~$ (echo fusca; echo corcel) > carros.txt
:~$ cat carros.txt
fusca
corcel
```

Contudo, existem diferenças importantes entre as duas formas de agrupamento:

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 terminando o último comando: geralmente `;`, `&`, ou uma quebra de linha.

Agrupamento com parêntesis:

- Os comandos são executados em um ***subshell***;
- Os parêntesis são metacaracteres (que separam palavras), por isso não precisamos de espaços

nem da terminação do último comando;
- Se prefixado com `$`, um agrupamento com parêntesis será uma expansão, a **substituição de comandos**, e suas saídas poderão ser armazenadas em variáveis.

Estruturas de decisão

Assunto de anotações anteriores, as estruturas de decisão implementadas como comandos compostos são o `if`, que avalia o estado de saída de um comando testado, e o `case`, que compara uma string com padrões de texto descritos em várias cláusulas.

Estruturas de repetição

O Bash tem três estruturas de repetição: os loops `for`, `while` e `until`. Se considerarmos que o `for` oferece duas formas de controle das repetições, poderemos dizer que temos quatro tipos de estruturas de repetição:

- Loop `for` controlado por uma lista de palavras.
- Loop `for` controlado por uma expressão aritmética ternária (estilo C).
- Loop `while`, interrompido quando um comando testado sai com erro.
- Loop `until`, interrompido quando um comando testado sai com sucesso.

Loop 'for'

Sintaxe:

```
for VAR in LISTA_DE_PALAVRAS; do
    BLOCO_DE_COMANDOS
done [redirecionamentos]
```

Exemplo:

```
:~$ for nome in Maria José 'José Maria'; do echo $nome; done
Maria
José
José Maria
```

Expandindo a lista de palavras em um vetor:

```
:~$ var=(verde amarela azul)
:~$ for cor in "${var[@]}"; do echo Casa $cor; done
Casa verde
Casa amarela
Casa azul
```

Quando a lista de palavras é a expansão de todos os argumentos passados para um script ou uma função (`"$@"`), a parte `in LISTA_DE_PALAVRAS` pode ser omitida:

```
for arg; do
    echo $arg
done
```

Loop 'for' estilo C

```
for ((INÍCIO;CONTROLE;ATUALIZAÇÃO)); do
    BLOCO_DE_COMANDOS
done [redirecionamentos]
```

Onde:

- `INÍCIO`: geralmente, uma atribuição (exemplo: `i=0`);
- `CONTROLE`: uma condição de parada (exemplo: `i<10`);
- `ATUALIZAÇÃO`: uma alteração do valor controlado (exemplo: `i++`).

Exemplo:

```
for ((n=1; n<=10; n++)); do printf '%d ' $n; done; echo
1 2 3 4 5 6 7 8 9 10
```

Opcionalmente, podemos utilizar chaves para delimitar o bloco de comandos:

```
:~$ for ((n=1; n<=10; n++)) { printf '%d ' $n; }; echo
1 2 3 4 5 6 7 8 9 10
```

Loop 'while'

```
while COMANDO_TESTADO; do
    BLOCO_DE_COMANDOS
done [redirecionamentos]
```

Exemplo:

```
:~$ while read; do echo $REPLY; done < frutas.txt
banana
laranja
abacate
```

Aqui, todos os comandos têm acesso à leitura do arquivo `frutas.txt`, mas apenas o comando `read` faz essa leitura. Cada vez que o `read` lê uma linha do arquivo, ele termina com estado de sucesso e o `while` permite a execução do bloco de comandos. ao mesmo tempo, o ponteiro

interno do arquivo muda para a linha seguinte, e esta será a linha lida quando o `read` for novamente invocado. Os ciclos continuam até que o `read` encontre o fim do arquivo e termine com erro.

Loop infinito:

```
while true; do
    BLOCO_DE_COMANDOS
done

while :; do
    BLOCO_DE_COMANDOS
done
```

Os comandos internos `true` e `:` não fazem nada e terminam sempre com sucesso, o que faz com que o `while` sempre permita uma nova repetição do bloco de comandos. Nesta situação, ou terminamos o loop manualmente, com o atalho `Ctrl+C`, ou escrevemos alguma estrutura de controle no bloco de comandos para interromper o loop (comando `break`) condicionalmente:

```
:~$ v=1; while :; do echo $((v++)); [[ v -gt 5 ]] && break; done
1
2
3
4
5
```

Loop 'until'

```
until COMANDO_TESTADO; do
    BLOCO_DE_COMANDOS
done [redirecionamentos]
```

Tem o mesmo comportamento do loop `while`, exceto pelo fato de só permitir a execução do bloco de comandos se o comando testado terminar com erro.

Comandos de controle de loops

Além dos controles das próprias estruturas de repetição, o Bash nos dá dois comandos internos para controlar loops:

- `break`: interrompe a execução do bloco de comandos e sai do loop.
- `continue`: interrompe a execução do bloco de comandos e segue para o próximo loop.

O menu 'select'

O comando composto `select` é uma estrutura especializada na criação de menus rápidos a partir de listas de palavras. Sua sintaxe é idêntica à do loop `for` infinito:

```
select VAR [in PALAVRAS]; do
    BLOCO DE COMANDOS
done
```

O `select` se comporta como um loop infinito que espera a entrada de uma opção numérica do usuário para continuar. A opção digitada é armazenada em duas variáveis:

- A string correspondente à PALAVRA escolhida vai para a variável `VAR`;
- A entrada digitada pelo usuário 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 `menu-pizzas.sh`, abaixo:

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

titulo='CARDÁPIO DE PIZZAS
------------------'

menu=(Calabresa Muçarela Vegana Sair)

clear
echo "$titulo"
select opt in ${menu[@]}; do
    [[ $REPLY == ${#menu[@]} ]] && break
    [[ $REPLY -gt ${#menu[@]} ]] && {
        echo 'Opção inválida!'
    } || {
        echo "Você escolheu $REPLY: ${opt,,}"
    }
    read -n1 -p 'Tecle algo para continuar...'
    clear
    echo "$titulo"
done
```

Quando executado, este será o resultado:

```
CARDÁPIO DE PIZZAS
------------------
1) Calabresa
2) Muçarela
3) Vegana
4) Sair
#? _
```

O próprio `select` cuida 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`. Assim, o número associado a ele pelo `select` corresponde ao número total de elementos do vetor, o que nos permite testar facilmente se o utilizador escolheu sair do menu pela expansão da quantidade de elementos:

```
[[ $REPLY == ${#menu[@]} ]] && break
```

O prompt PS3

Os caracteres `#?`, exibidos no prompt do menu `select`, são definidos em uma das 5 variáveis do shell que determinam como os prompts serão exibidos:

- A variável `PS0` contém a string que é exibida antes de um comando ser executado no shell interativo.
- A variável `PS1` contém a string do prompt da linha de comandos.
- A variável `PS2` contém o caractere `>`, que indica a continuação da entrada de comandos.
- A variável `PS3` contém o prompt do menu `select`, que é `#?`, por padrão.
- A variável `PS4` contém o caractere `+`, que aparece quando estamos "debugando" a execução do shell (comando `set -x`).

Então, se quisermos um prompt mais amigável no nosso script, basta definir a string do prompt `PS3` antes da estrutura do menu:

```
...
menu=(Calabresa Muçarela Vegana Sair)
PS3='Escolha a sua opção: '
...
```

O resultado será:

```
CARDÁPIO DE PIZZAS
------------------
1) Calabresa
2) Muçarela
3) Vegana
4) Sair
Escolha a sua opção: _
```

Avaliação de expressões

Também já vistos neste guia, os comandos compostos `[[...]]` e `((...))` são estruturas que avaliam expressões:

- Avaliação de expressões afirmativas: `[[ expressão ]]`
- Avaliação de expressões aritméticas: `((expressão))`

> **Importante!** Apenas o comando `[[`, também chamado de "teste do Bash", é um comando composto. Os *builtins* `test` e `[` são comandos comuns.

10 – Funções

No shell, funções são implementadas na forma de comandos compostos identificados por um nome.

Sintaxe geral:

```
NOME() COMANDO-COMPOSTO [redirecionamentos]
```

Alternativamente, pode ser utilizada a palavra reservada `function`. Neste caso, os parêntesis são dispensáveis:

```
function NOME COMANDO-COMPOSTO
```

- Funções devem ser definidas antes do ponto onde são chamadas.
- Funções podem chamar outras funções definidas antes ou depois de suas próprias definições.
- Funções podem ter definições de outras funções nos seus corpos.

Criando funções com agrupamentos

É possível transformar qualquer comando composto em uma função, mas o agrupamento com chaves é a opção mais comum:

```
diga_oi() {
    echo oi
}
```

Muitos dos conceitos difundidos sobre funções no shell são apenas consequências do uso do agrupamento com chaves, como:

- Variáveis de escopo global;
- Executada na mesma sessão do shell.

Contudo, uma função criada com um agrupamento com parêntesis é perfeitamente válida, mas nós teríamos que lidar com comportamentos diferentes:

- Variáveis de escopo restrito à função;
- Executada em um subshell.

Nomeando funções

As regras para o nomear funções são muito mais flexíveis do que as que limitam os nomes de variáveis: em geral, qualquer conjunto de caracteres é válido:

```
:~$ 2 () { echo 'vale tudo?'; }
:~$ 2
vale tudo?
:~$ @ () { echo 'vale tudo?'; }
:~$ @
vale tudo?
:~$ ? () { echo 'vale tudo?'; }
:~$ ?
vale tudo?
:~$ . () { echo 'vale tudo?'; }
:~$ .
vale tudo?
```

Porém, é sempre uma boa ideia adotar alguns critérios, como:

- Utilizar apenas caracteres não acentuados minúsculos.
- Separar nomes compostos com o sublinhado.
- Escolher nomes sigificativos.
- Utilizar o sublinhado como primeiro caractere do nome quando o nome coincidir com algum comando existente.

Passagem de argumentos para funções

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
```

Os parâmetros posicionais `1` e `2` continuaram expandindo seus valores originais na sessão do shell, mas assumiram outros valores durante a execução da função.

Este é o ponto mais importante, porque, como todas variáveis têm escopo global numa mesma sessão do shell, o esperado seria que os parâmetros posicionais expandissem o mesmo valor da sessão mãe, mas este é um comportamento exclusivo de funções. Se fosse um simples 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 fazer referência aos parâmetros posicionais passados para a função, mas o parâmetro especial `0` continua contendo o nome do shell ou do script.

Escopo de variáveis em funções

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.

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:

```
:~$ 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
```

Destruindo funções

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
```

Retorno de funções no shell

No shell, não existe o conceito de funções retornando dados processados. Aqui, o conceito de **retorno** limita-se à ideia de devolver o controle da execução para o fluxo principal do programa entregando um estado de saída.

O comando interno 'return'

- O comando interno `return` equivale ao `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 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.

Variáveis, apelidos e funções

Na customização da linha de comandos, é muito comum nós criarmos apelidos para abreviar a digitação ou padronizar as opções de um comando. Isso pode ser feito tanto com o comando `alias` quanto com variáveis:

```
# 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
```

Tanto apelidos quanto a variáveis expandem para a string do comando, mas ambos os métodos carecem de uma coisa que pode ser bem mais útil numa linha de comandos: a capacidade de receber e processar argumentos **de fato**: isso só é possível com scripts ou funções.

Às vezes, não vale a pena passar por todo o processo de criação de scripts e gerenciamento de arquivos para fazer essas pequenas customizações. Nesses casos, pode ser mais interessante criar funções.

Se quisermos, por exemplo, um comando para calcular o quadrado de um inteiro qualquer, basta criar uma função `quadrado` e salvá-la no final de um arquivo de início, como o arquivo `~/.bashrc`. Assim, a função estará sempre disponível para quando precisarmos...

No arquivo `~/.bashrc`:

```
quadrado() { echo $(($1**2)); }
```

Executando (depois do shell recarregado):

```
:~$ quadrado 6
36
```