Capítulo 4

Introdução ao *script-shell* para LINUX

4.1 Aspectos básicos

4.1.1 Script e Script Shell

O *shell* é um interpretador de comandos que possui uma linguagem utilizada por diversas pessoas para facilitar a realização de inúmeras tarefas administrativas no Linux (como efetuar *backup* regularmente, procurar textos, criar formatações), e até mesmo para criar programas um pouco mais elaborados. A linguagem *shell* é interpretada, não havendo necessidade de compilar para gerar um arquivo executável.

Um ***script shell***, ou simplesmente ***script***, é um arquivo contendo uma seqüência de um ou mais comandos. Este arquivo é diretamente executável quando chamado pelo nome.

O *Shell* foi escrito em diferentes versões. Dos vários programas *Shell* existentes, o *Bourne Shell*, o *Korn Shell* e o *C Shell* se destacam por serem os mais utilizados e conhecidos.

O *Bourne Shell* é conhecido como *Shell* padrão, sendo o mais utilizado e estando na maioria dos sistemas *Unix like*.

O *Korn Shell* é uma versão melhorada do *Bourne Shell*.

O *C Shell* possui uma estrutura bastante parecida com a linguagem C e é também uma versão modificada do *Bourne Shell*.

Além desses, há um *shell* padrão do Linux, chamado *Bourne-Again Shell*. Este pode ser considerado o mais completo, sendo compatível com todos os *shells* citados anteriormente.

Mas qualquer programador pode fazer o seu *Shell*. Estes *shells* tornaram-se conhecidos pois já vinham com o sistema, exceto o *Korn*, que tinha que ser adquirido separadamente. O *Bourne Shell* vinha com o *System V* e o *C Shell* com o *BSD*.

Algumas características:

Shell		Prompt	Representação
Bourn Shell	⇒	\$	**sh**
Bourn-Again Shell	⇒	\$	**bash**
Korn Shell	⇒	\$	**ksh**
C shell	⇒	%	**csh**

4.2 Execução do programa

Um programa pode ser escrito em um editor de sua preferência como vi, kWrite, KEdit entre outros. O arquivo é salvo como texto comum. No ínicio do arquivo deve vir escrito:

`#!/bin/bash`

Os caracteres especiais #! (chamados *hash-bang*) informam ao kernel que o próximo argumento é o programa utilizado para executar este arquivo. No caso, `/bin/bash` é o shell que utilizamos. O kernel lê o *hash-bang* no início da linha, então ele continua lendo os caracteres seguintes e inicia o *bash*. Quando o *shell* lê o *hash-bang*, ele o interpreta como uma linha de comentário e a ignora, iniciando a execução do programa.

É preciso mudar a permissão do arquivo para executável para ele funcionar. Isso é feito pelo comando `chmod`. Para que o programa seja executável de qualquer parte do sistema é necessário salvá-lo em algum diretório que esteja no PATH (variável do sistema que contém a lista de diretórios onde o *shell* procura pelo comando digitado). Entretanto, é permitido salvá-lo em um diretório qualquer (como o diretório home do usuário), porém na hora de executá-lo pelo prompt, é necessário que seja indicado todo o caminho desde a raiz, ou, estando no mesmo diretório do arquivo, digitar no prompt: ./nomearquivo.

OBS: Também é possível modificar o PATH para incluir um diretório de sua escolha.
Por exemplo: vamos supôr que o usuário criou uma pasta em seu diretório para salvar seus scripts com o nome de `scripts`. Para tornar esse diretório como parte do PATH faça o seguinte:

Digite no prompt:

$ `echo $PATH`
Esse é seu PATH atual. Em seguida digite:
$ `PATH=$HOME/scripts:$PATH`
e
$ `echo $PATH`

Esse é seu novo PATH, com seu diretório de exemplos incluído. Agora seus scripts podem ser executados de qualquer diretório. Porém, dessa forma, a mudança da variável PATH só vale enquanto o *shell* corrente estiver aberto. Se for aberto outro *shell*, a mudança não terá efeito.

Existem arquivos que são inicializados assim que o *shell* é aberto:

`/etc/profile` : Tem as configurações básicas do sistema e vale para todos os usuários. Somente o *root* tem permissão para modificar esse arquivo.

`.bash_profile` ou `.bash_login` ou `.profile` ou `.bashrc` : Estes arquivos ficam no diretório home do usuário. As modificações feitas nesse arquivo só valem para o próprio usuário. Podemos, então, abrir o arquivo `.bashrc` e colocar nele o novo PATH. Além disso, podemos incluir também *aliases*.

Ex: Se tivéssemos um diretório chamado `scripts` e quiséssemos colocá-lo no PATH, bastaria acrescentar a linha abaixo ao arquivo `.bashrc`. Também foram colocados alguns *aliases*.

```
PATH=$PATH:~/scripts
alias c='clear'
alias montar='mount /dev/fd0'
```

Vamos fazer um exemplo passo-a-passo agora para criar e executar um programa.
Escreva em seu editor de texto o programa abaixo e salve como `um.sh`.

```
#!/bin/bash

echo "Programa UM!"
```

Agora no prompt, mude as permissões do arquivo.

```
$ chmod a+x um.sh
```

Agora é só chamar o programa no prompt.

```
$ sh um.sh
```

Neste caso o arquivo estava no mesmo diretório de trabalho. E se o arquivo estivesse em um diretório diferente? Lembre-se de mudar o PATH!

4.2.1 Erros na execução

Para o usuário iniciante, é bem provável que ele se depare com alguns erros bem comuns e fáceis de resolver. Os principais são:

- "*command not found* - Esse quer dizer que o shell não encontrou seu programa.

  A razão para isso pode ser que o nome do comando foi digitado de forma diferente do nome do arquivo. Certifique-se de que o nome está igual. Outra razão possível é que o arquivo está em um local diferente do PATH padrão. Nesse caso, deve-se proceder conforme explicado na seção anterior, que explica como salvar o arquivo.

- "*Permission denied*- A permissão para execução do arquivo foi negada. O usuário deve mudar a permissão do arquivo para executável.

- Outro erro comum é o de sintaxe. Nesse caso o *shell* encontra o *script* e indica a linha onde ocorreu o erro no programa.

4.2.2 *Quoting*

Denomina-se *quoting* a função de remover a predefinição ou significado especial dos metacaracteres. Dessa forma, é possível escrever os caracteres literalmente.
Há 3 tipos de mecanismos para *quoting*:

1. **Barra invertida** (\) - Chamada de caracter de escape, ela remove o significado especial do caracter seguinte à ela.

   Exemplo:

```
$ echo 0 # é um caracter especial
  0
$ echo 0 \# é um caracter especial
  0 # é um caracter especial
```

Observe a diferença que o uso da barra provoca.

```
$ echo agora a barra está sendo\
> usada para que seja possível\
> continuar escrevendo em outra linha\
> Veja que apareceu um prompt secundário \
> Mas na hora da impressão as linhas saem seguidas.
 agora a barra está sendousada para que seja possívelcontinuar
 escrevendo em outra linhaVeja que apareceu um prompt
 secundário Mas na hora da impressão as linhas saem seguidas.
```

Neste exemplo a barra suprimiu o significado de fim de linha.

2. **Aspas simples** (') - Todos os caracteres que vem entre estas aspas tem seu significado removido.

   Exemplo:

```
$ echo 'Com aspas simples não é necessário colocar\
> barra para cada caracter # $ * !'
  Com aspas simples não é necessário colocar barra para
  cada caracter # $ * !
```

3. **Aspas duplas** (") - É semelhante à anterior, porém não remove o significado de $, \, ", ' e {variável}.

4.3 Comentários

Para deixar seu programa mais claro e fácil de entender, o usuário não só pode como deve acrescentar comentários em seu código-fonte utilizando o caracter # no início da linha. Esta linha não será executada pelo *shell* quando ele o encontrar. Isso é bastante útil para permitir a leitura posterior do arquivo, correção de erros, mudanças no programa entre outras tarefas.

Por exemplo, no início de um *script* poderia ser escrita a sua função.

```
# Script para administrar as contas dos novos usuários

 <comandos>
```

Um bom código-fonte deve ter um cabeçalho identificando o autor e as funções do programa. É importante lembrar que, independentemente da linguagem usada, comentários são essenciais para um bom entendimento do programa escrito, tanto pelo autor como por outros usuários.

Abaixo segue uma lista de recomendações que um programador deve seguir:

- Escreva os comentários no momento em que estiver escrevendo o programa.
- Os comentários devem acrescentar algo e não apenas descrever o funcionamento do comando.
- Utilize espaços em branco para aumentar a legibilidade.
- Coloque um comando por linha, caso a situação permita.
- Escolha nomes representativos para as variáveis.

4.4 Impressão na tela

O comando `echo` permite mostrar na tela seus argumentos.

Ex:

```
$echo Escrevendo seu argumento
 Escrevendo seu argumento
```

Existem caracteres especiais que são interpretados pelo comando `echo`. Em algumas versões do Linux deve ser usado o parâmetro opcional `-e`. Esta opção habilita a interpretação dos caracteres de escape listados abaixo.

- `\\` - Barra invertida (*backslash*).
- `\nnn` - Escreve o caracter cujo octal é nnn.
- `\xHH` - Escreve o caracter cujo hexadecimal é HH.
- `\a` - Caracter de alerta sonoro (*beep*).
- `\b` - *Backspace*.
- `\c` - *Suprime newline*, forçando a continuação na mesma linha.
- `\f` - Alimentação de formulário.
- `\n` - Inicia um nova linha.
- `\r` - *Carriege return*. Retorna ao início da linha.
- `\t` - Equivale a um espaço de tabulação horizontal.
- `\v` - Equivale a um espaço de tabulação vertical.

Ex:

```
$ echo -e "Este exemplo mostra o uso de alguns dos caracteres mostrados:
> Começando pela contra-barra \\
> Caracter hexadecimal \100
> Usando backspace\b
> iniciando \n nova linha
> apagando o que foi \r escrito anteriormente na linha
> e tabulando \t horizontalmente e \v verticalmente."
Este exemplo mostra o uso de alguns dos caracteres mostrados:
Começando pela contra-barra \
Caracter hexadecimal @
Usando backspace
iniciando
 nova linha
 escrito anteriormente na linha
e tabulando      horizontalmente e
                                   verticalmente.
```

OBS: Para saber a representação octal e hexadecinal correspondente ao caracter desejado, consulte a tabela ASCII no manual: man ascii.

Quando o usuário não desejar que a saída do comando `echo` pule de linha, ele deve usar a opção `-n`.

4.5 Passagem de parâmetros e argumentos

Parâmetros são variáveis passadas como argumentos para o programa ou variáveis "especiais"que guardam certas informações.

Quando um programa recebe argumentos em sua linha de comando, estes são chamados de **parâmetros posicionais**. Eles são numerados de acordo com a ordem em que foram passados.

Exemplo:

```
$ cat local.sh
#Programa recebe e mostra dados

echo Cidade: $1
echo Estado: $2
echo País:   $3

$ ./local.sh Niterói "Rio de Janeiro" Brasil
Cidade: Niterói
Estado: Rio de Janeiro
País: Brasil
```

Como pode ser visto no exemplo, o programa chamado `local` recebe 3 argumentos. O primeiro, `Cidade`, é guardado na variável $1, o segundo, `Estado`, em $2 e o terceiro, `País`, em $3. Porém, o shell limita em 9 o número de argumentos. Para trabalhar com essa limitação, existe o comando `shift n` que faz com que os primeiros `n` argumentos passados não façam parte na contagem de argumentos.

Exemplo: Serão passados mais de 9 argumentos para o programa, mas através do uso do comando `shift` será possível listar todos.

```
$ cat shift.sh
#Programa recebe e mostra dados usando shift

echo Argumento1: $1
echo Argumento2: $2
echo Argumento3: $3
echo Argumento4: $4
echo Argumento5: $5
echo Argumento6: $6
echo Argumento7: $7
echo Argumento8: $8
echo Argumento9: $9
shift 9
echo Argumento10: $1
echo Argumento11: $2

$ ./shift.sh a b c d e f g h i j l m n
Argumento1: a
Argumento2: b
Argumento3: c
Argumento4: d
Argumento5: e
Argumento6: f
Argumento7: g
Argumento8: h
Argumento9: i
Argumento10: j
Argumento11: l
```

Há também os **parâmetros especiais** que podem ser usados pelo programa:

$ * - Este parâmetro informa uma string com todos os argumentos passados para o programa, onde cada argumento aparece separado pelo IFS (veja a seção 5.8, que fala de variáveis de sistema).

$ @ - Este parâmetro é semelhante ao anterior, porém os argumentos aparecem separados por espaços em branco.

$# - Este informa o número de argumentos passados na chamada do programa.

$? - Este guarda o valor de retorno do último comando executado. Quando a execução do programa acontece normalmente, é retornado 0, e se tiver ocorrido algum erro é retornado um valor diferente de zero.

$$ - Informa o número do processo de um determinado programa (PID).

$! - Informa o número do processo do último programa sendo executado em *background*.

$0 - Informa o nome do *shell script* executado.

Exemplo: Será visto o uso de alguns desses parâmetros especiais.

```
$ cat parametros.sh
# Mostra os dados relativos aos parametros passados

echo Foram passados $# argumentos.
echo Os argumentos foram: $@.

$ ./parametros.sh um dois três quatro
Foram passados 4 argumentos.
Os argumentos foram: um dois três quatro.
```

4.5.1 Leitura de parâmetros

Quando for necessário passar algum dado para o programa, usa-se o comando `read`, que lê a entrada escrita no terminal.

Exemplo:

```
$ cat read.sh
# uso do comando read

echo Digite uma palavra:
read algo
echo Você digitou: \"$algo\"

$ ./read.sh
Digite uma palavra:
Por que não uma frase?
Você digitou: "Porque não uma frase?"
```

Este comando permite ainda a passagem de uma lista de variáveis, desde que estas venham separadas entre espaços em branco.

Exemplo: Agora o programa `read` sofreu uma modificação para receber uma lista de variáveis.

```
$ cat read.sh
# uso do comando read

echo Digite umas palavras:
read prim segun ter
echo Você digitou:
```

```
echo      \"$prim\"
echo      \"$segun\"
echo      \"$ter\"

$ ./read.sh
Digite umas palavras:
Mas agora eu quero digitar uma frase!
Você digitou:
"Mas"
"agora"
"eu quero digitar uma frase!"
```

Se forem passadas mais variáveis do que o comando `read` vai ler, a variável excedente é interpretada como se fizesse parte da última variável. Como pode ser visto pelo exemplo anterior, onde foram passadas mais variáveis do que as 3 que o programa leria. Então, a terceira variável ficou com uma frase.

Algumas opções podem ser usadas com o comando `read`. São elas:

- `-p` – Nos exemplos acima foi preciso usar o comando `echo` toda vez antes do comando `read`. Porém, isso não é necessário, basta usar esta opção. O próximo exemplo mostra como.

  Exemplo:

  ```
  $ cat read.sh
  # uso do comando read

  read -p "Digite umas palavras:" prim segun ter
  echo Você digitou:
  echo      \"$prim\"
  echo      \"$segun\"
  echo      \"$ter\"

  $ ./read.sh
  Digite umas palavras:Dessa vez o echo não foi usado antes
  Você digitou:
  "Dessa"
  "vez"
  "o echo não foi usado antes"
  ```

- `-s` – Esse parâmetro serve para não ecoar o que foi digitado. Seu uso principal é na leitura de senhas.

- `-n` – Este parâmetro permite limitar o número de caracteres que serão lidos pelo `read`. Sua sintaxe é: `read -n N string`. Lerá apenas os N caracteres da *string* digitada.

4.6 Funções

Funções são estruturas que reúnem comandos na forma de blocos lógicos, permitindo a separação do programa em partes. Quando o nome de uma função é chamado dentro do *script* como um comando, todos os comandos associados a esta função são executados.

A sintaxe para o uso de funções é da forma:

```
function nome () {
    ...
    <comandos>
    ...
}
```

Onde `nome` é o nome que será dado à função criada. E `comandos` definem o corpo da função.

A principal vantagem de usar funções é a possibilidade de organizar o código do programa em módulos.

O exemplo a seguir demonstra o uso de funções em *script*. Um menu de opções é mostrado, sendo que cada opção leva à execução de uma função diferente.

```
#!/bin/bash
#
#
#  Este script executa as funções básicas de uma calculadora:
#   Soma, Subtração, Multiplicação e Divisão.
#
#
#
#

clear
menu()
  {
     echo "         Calculadora Básica          "
     echo "     Operação apenas com inteiros    "
     echo "|-----------------------------------|"
     echo "|   Escolha uma das opções abaixo:  |"
     echo "|-----------------------------------|"
     echo "|  1) Soma                          |"
     echo "|  2) Subtração                     |"
     echo "|  3) Multiplicação                 |"
     echo "|  4) Divisão                       |"
     echo "|  5) Sair                          |"
     echo "|-----------------------------------|"
     echo "|-----------------------------------|"

  read opcao
  case $opcao in
```

```
     1) soma ;;
     2) subtra ;;
     3) mult ;;
     4) div ;;
     5) exit ;;
     *) "Opção Inexistente" ;
         clear ;
     menu ;;
  esac
     }



soma()
  {
     clear
     echo "Informe o primeiro número"
     read num1
     echo "Informe o segundo número"
     read num2
     echo "Resposta = ‘expr $num1 "+" $num2‘"
     menu

  }

subtra()
  {
     clear
     echo "Informe o primeiro número"
     read num1
     echo "Informe o segundo número"
     read num2
      echo "Resposta = ‘expr $num1 "-" $num2‘"
     menu
  }

mult()
  {
     clear
     echo "Informe o primeiro número"
     read num1
     echo "Informe o segundo número"
     read num2
      echo "Resposta = ‘expr $num1 "*" $num2‘"
     menu
```

```
  }

div()
  {
     clear
     echo "Informe o primeiro número"
     read num1
     echo "Informe o segundo número"
     read num2
      echo "Resposta = ‘expr $num1 "/" $num2‘"
     menu
  }

menu
```

Repare no *script* acima que a função `menu` foi colocada no final do programa. Experimente chamar a função `menu` no início e veja o que acontece.

4.6.1 Execução de *script* por outro *script*

É possível executar um script de outro arquivo com se ele fosse uma função qualquer dentro de outro programa. Para isso, basta escrever: . `nomescript` no lugar onde ele deve ser executado.

Tendo como base o exemplo anterior, podemos mostrar a execução de *scripts* dentro de outros.

O primeiro programa é o *script* principal referente ao `menu` de opções. Observe como as operações são chamadas.

```
#!/bin/bash
#
#
#  Este script executa as funções básicas de uma calculadora:
#   Soma, Subtração, Multiplicação e Divisão.
#
#  PARTE -> MENU
#
#

clear


menu(){

     echo "         Calculadora Básica          "
     echo "     Operação apenas com inteiros    "
     echo "|-----------------------------------|"
     echo "|   Escolha uma das opções abaixo:  |"
     echo "|-----------------------------------|"
```

```
      echo "|  1) Soma                          |"
      echo "|  2) Subtração                     |"
      echo "|  3) Multiplicação                 |"
      echo "|  4) Divisão                       |"
      echo "|  5) Sair                          |"
      echo "|-----------------------------------|"
      echo "|-----------------------------------|"

  read opcao
  case $opcao in
     1) . soma.sh ;;
     2) . subtra ;;
     3) . mult ;;
     4) . div ;;
     5) exit ;;
     *) "Opção Inexistente" ;
         clear ;
     menu ;;
  esac

}
menu
```

O *script* seguinte refere-se a operação de soma.

```
#!/bin/bash
#
#
#  Este script executa as funções básicas de uma calculadora:
#   Soma, Subtração, Multiplicação e Divisão.
#
#  PARTE -> SOMA

soma()
  {
     clear
     echo "Informe o primeiro número"
     read num1
     echo "Informe o segundo número"
     read num2
     echo "Resposta = `expr $num1 "+" $num2`"
     menu

  }
  soma
```

4.7 Depuração

Para verificar os problemas e possíveis causas de erros que acontecem nos *scripts*, basta rodar o programa da seguinte forma:

```
$ sh -x programa
```

A opção -x faz com que o programa seja executado passo-a-passo, facilitando a identificação de erros.

Capítulo 5

Manipulação de variáveis

Variável é uma posição nomeada de memória, usada para guardar dados que podem ser manipulados pelo programa.

Em *shell* não é necessário declarar a variável como em outras linguagens de programação.

5.1 Palavras Reservadas

Palavras reservadas são aquelas que possuem um significado especial para o *shell*.

O *Shell* possui comandos próprios (intrínsecos) como:

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

Além disso, o Unix possui outros comandos, vistos nos capítulos anteriores.

Em programação, geralmente trabalhamos com manipulação de variáveis. Dessa forma, é importante lembrar que o uso dessas palavras deve ser evitado, tanto no nome dado às variáveis quanto no nome dado ao *script*.

5.2 Criação de uma variável

Uma variável é criada da seguinte forma:

```
$  nomevar=valor
```

Uma variável é reconhecida pelo *shell* quando ela vem precedida pelo símbolo $. Quando este símbolo é encontrado, o *shell* substitui a variável pelo seu conteúdo. `nomevar` é o nome da variável e *valor* é o conteúdo que será atribuído à variável. É importante assegurar que não haja espaço antes e depois do sinal "="para evitar possíveis erros de interpretação. No exemplo acima, foi criada uma variável local. Para criar um variável global, é usado o comando `export`.

$ `export nomevar`

ou

$ `export nomevar`=valor

Exemplo: Será atribuído um valor à variável chamada `var`, e em seguida este valor será mostrado pelo comando `echo`.

```
$ var=Pensamento
$ echo "O conteúdo é o $var"
O conteúdo é o Pensamento
```

Vale lembrar algumas regras para a nomenclatura de variáveis que se aplicam às linguagens de programação:

- O nome da variável só pode começar com letras ou *underline*.
- São permitidos caracteres alfanuméricos.
- Não devem haver espaços em branco nem acentos.

  Exemplo: Veja o uso de aspas simples, duplas e crases com variáveis:

```
$ variavel="Meu login é: $user"
$ echo $variavel
 Meu login é: ze
$ variavel='Meu hostname é: $HOSTNAME'
$ echo $variavel
 Meu hostname é: $HOSTNAME
$ variavel="O diretorio de trabalho  é: `pwd`"
$ echo $variavel
 O diretorio de trabalho é: /home/ze
```

Quando vamos executar um script ou comando, um outro *shell* é chamado, executa os comandos e então retorna ao *shell* original onde foi feita a chamada. Por isso é importante lembrar de exportar suas variáveis para que elas sejam reconhecidas pelo "*shell* filho".

```
$ cat dir.sh
#!/bin/bash

echo "Veja que o diretório vai mudar"
echo "Inicialmente o diretório era: $PWD"
# neste ponto o diretorio mudou
cd /usr/bin
echo "Agora o diretório atual é $PWD"
echo "Mas terminado o programa parece que nada aconteceu.
O diretório continua sendo o inicial."

$ sh dir.sh
Veja que o diretório vai mudar
Inicialmente o diretório era: /home/kurumin/scripts
Agora o diretório atual é /usr/bin
Mas terminado o programa parece que nada aconteceu.
O diretório continua sendo o inicial.
$
```

5.3 Deleção de uma variável

Uma variável é apagada quando for usado o comando `unset`.

`$unset nomevar`

Exemplo: Vamos ver o que acontece quando a variável `var`, criada anteriormente, for deletada.

```
$ unset var
    $ echo "O conteúdo é o $var"
    O conteúdo é o
```

5.4 Visualização de variáveis

Utilizando o comando `set` é possível visualizar as variáveis locais e com o comando `env` as variáveis globais podem ser vistas.

5.5 Proteção de uma variável

Para evitar alterações e deleção de uma determinada variável usa-se o comando `readonly`. Dessa forma, a variável ganha atributo de somente leitura.

`$ readonly nomevar`

Todas as variáveis readonly, uma vez declaradas, não podem ser "unsetadas"ou ter seus valores modificados. O único meio de apagar as variáveis readonly declaradas pelo usuário é saindo do shell (logout).

5.6 Substituição de variáveis

Além de substituição de variáveis (variável pelo seu conteúdo, visto em exemplos anteriores), outros tipos de substituições são possíveis no *shell*. As principais são:

- **Substituição de comando** - Nesse caso, o nome do comando é substituído pelo resultado de sua operação quando ele for precedido pelo símbolo $ e entre () ou vier entre sinais de crase (‘).

```
$(comando)

ou

‘comando‘
```

Geralmente este tipo de manipulação é utilizada na passagem do resultado de um comando para uma variável ou para outro comando.

Exemplo:

```
$ dir=‘pwd‘
    $ echo "O diretório atual tem o seguinte caminho: $dir"
    O diretório atual tem o seguinte caminho: /home/kurumin/scripts
```

5.7 Variáveis em vetores

O *shell* permite o uso de variáveis em forma de *array*. Ou seja, vários valores podem ser guardados em uma variável seguindo a ordem de uma indexação. Assim como não é necessário declarar o tipo de variável no início do programa, também não é preciso declarar que uma variável será usada como vetor.

Um array é criado automaticamente se for atribuído um valor em uma variável da seguinte forma: `nomevar[indice]=valor`, onde índice é um número maior ou igual a zero.

Exemplo:

```
$ camada[0]=Física
$ camada[1]=Enlace
$ camada[2]=Redes
$ echo "As 3 camadas mais baixas da internet são: ${camada[*]}"
As 3 camadas mais baixas da internet são: Física Enlace Redes
```

Outra forma de se atribuir valores em *array* é:
`nomevar=(valor1, valor2, ..., valorn)`.

```
$ camada=(Física, Enlace, Redes)
$ echo "As 3 camadas mais baixas da internet são: ${camada[*]}"
As 3 camadas mais baixas da internet são: Física, Enlace, Redes
```

Para fazer referência a um elemento do array é só fazer: `${nomevar[indice]}`.

Para ver toda a lista de valores do *array* basta fazer `${nomevar[*]}` ou `${nomevar[@]}`. A diferença entre o uso do * ou @ é semelhante a diferença vista no uso de parametros especiais.

5.8 Variáveis do sistema

Existem algumas variáveis que são próprias do sistema e outras que são inicializadas diretamente pelo *shell*.

Algumas dessas variáveis, denominadas **variáveis do shell** são explicadas abaixo:

HOME – Contém o diretório *home* do usuário.

LOGNAME – Contém o nome do usuário atual.

IFS – Contém o separador de campos ou argumento (*Internal Field Separator*). Geralmente, o IFS é um espaço, tab, ou nova linha. Mas é possível mudar para outro tipo de separador.

Exemplo:

```
$ num=(1 2 3 4 5)
$ echo "${num[*]}"
1 2 3 4 5
$ echo "${num[@]}"
1 2 3 4 5
$ OLDIFS=$IFS
$ IFS='-'
$ echo "${num[*]}"
1-2-3-4-5
$ echo "${num[@]}"
1 2 3 4 5
$ echo $IFS

$ echo "$IFS"
-
$ IFS=$OLDIFS
$ echo "$IFS"
```

Vamos entender o que foi feito: foi criado um vetor para ilustrar o IFS quando forem usados os caracteres e * para listar o *array*. O IFS inicial é um espaço em branco, então tanto pelo uso do como pelo uso do *, os elementos foram listados separados por um espaço em branco. Em seguida uma variável chamada OLDIFS recebeu o conteúdo de IFS e IFS recebeu um hífen (-). A separação na listagem dos elementos saiu diferente, ou seja, o novo separador foi usado quando foi usado o asterisco. Finalmente, o IFS recebeu seu valor inicial, espaço em branco. Essa mudança permanece somente na seção em que foi modificada e até que ela seja fechada.

PATH – Armazena uma lista de diretórios onde o shell procurará pelo comando digitado.

PWD – Contém o diretório corrente.

PS1 – Esta é denominada *Primary Prompting String*. Ou seja, ela é a string que está no *prompt* que vemos no *shell*. Geralmente a string utilizada é: `\u@\h:\w $`. O significado desses caracteres e de outros principais está explicado abaixo:

- `\s` O nome do shell.
- `\u` Nome do usuário que está usando o shell.
- `\h` O *hostname*
- `\w` Contém o nome do diretório corrente desde a raiz.
- `\d` Mostra a data no formato: `dia_da_semana mês dia`.
- `\nnn` Mostra o caracter correspondente em números na base octal.
- `\t` Mostra a hora atual no formato de 24 horas, HH:MM:SS.
- `\T` Mostra a hora atual no formato de 12 horas, HH:MM:SS.

- `\W` Contém somente o nome do diretório corrente.

PS2 – Esta é denominada *Secondary Prompting String*. Ela armazena a string do *prompt* secundário. O padrão usado é o caracter `>`. Mas pode ser mudado para os caracteres mostrados acima.

MAIL – É o nome do arquivo onde ficam guardados os e-mails.

COLUMNS – Contém o número de colunas da tela do terminal.

LINES – Contém o número de linhas da tela do terminal.

Existem muitas outras variáveis que são descritas na página do manual do *Bash*, na seção *Shell Variables*.

Capítulo 6

Testes e Comparações em *Script-Shell*

6.1 Código de retorno

Antes de falar sobre testes e comparações é importante que o usuário entenda como as decisões são tomadas dentro de um programa.

Todo comando do UNIX retorna um código e este é chamado *código de retorno*. Quando o comando é executado sem erros, o código retornado vale 0. Porém, se houver alguma falha, é retornado um número diferente de 0.

O caracter especial ? funciona como uma variável que guarda o código de retorno do comando anterior. O exemplo abaixo mostra o resultado da operação de alguns comandos.

Exemplo: Quando acontece algum erro na execução do comando, o código de retorno é diferente de zero.

```
$ echo "$PS1"
\u@\h:\w\$
$ echo $?
0
$ rm documento.txt
rm: cannot lstat ‘documento.txt’: No such file or directory
$ echo $?
1
```

6.2 Avaliação das expressões

As expressões são avaliadas no *shell* através do comando `test` ou pelo uso da expressão entre colchetes [ ], uma maneira mais prática do uso do comando `test`.

```
$ var=Z
$ test $var = w
$ echo $?
1
$ [ $var = Z ]
$ echo $?
0
```

O resultado da expressão é retornado 0 para verdadeiro ou não 0 para falso.

6.3 Operadores *booleanos*

Os operadores *booleanos* são relacionados à lógica `e`, `ou`, `negação` entre outras relações. Os operadores que podem ser usados em expressões no *shell* são:

- `-a` - e (*and*).
- `-o` - ou (or).
- `!` - negação (not).

A combinação desses 3 operadores pode gerar outras funções lógicas.
Várias condições também podem ser agrupadas com o uso de *condio*".
Ex:

```
$ cat boole.sh
#!/bin/bash

read -p "Informe um número e uma letra: " num letra

if [ \( "$num" -gt 0 -a "$num" -lt 10 \) -o \( $letra = v \) ]
 then
   echo "Acertou a faixa do numero ou a letra."
 else
   echo "Errou as duas informações."
fi
$ sh boole.sh
Informe um número e uma letra: 15 v
Acertou o faixa do numero ou a letra.
$
```

6.4 Testes Numéricos

As relações utilizadas para testes numéricos são as descritas abaixo:

- `-eq` – Igual à (*equal to*).
- `-gt` – Maior que (*greater than*).
- `-ge` – Maior ou igual (*greater or equal*).
- `-lt` – Menor que (*less than*).
- `-le` – Menor ou igual (*less or equal*).
- `-ne` – Não-igual a (*not equal to*).

A sintaxe para teste é:
`[ var/número relação var/número ]`
onde `var/número` indica o conteúdo da variável ou um número.
Exemplo:

```
$ num=10
$ [ $num -eq 9 ]; echo $?
1
$ [ $num -le 9 ]; echo $?
1
$ [ $num -ge 9 ]; echo $?
0
```

Outra maneira de realizar o teste é colocando `var/número` entre aspas. Dessa forma evitamos a ocorrência de erros.
`[ "var/número" relação "var/número" ]`
Exemplo: Observe o que acontece quando a expressão é comparada com o valor nulo sem aspas

```
$ [ $num -ge  ]; echo $?
bash: [: 10: unary operator expected
2
$ [ "$num" -ge " " ]; echo $?
0
```

6.4.1 O Comando `let`

Este comando permite outra maneira de fazer testes numéricos. Em vez de usar as relações citadas anteriormente, são usados os símbolos:

- == – Igual à
- > – Maior que
- >= – Maior ou igual
- < – Menor que
- <= – Menor ou igual
- != – Não-igual à

A sintaxe é:
`let expressão`
Exemplo:

```
$ let "0 != 1"
$ echo $?
0
```

Uma variação desse comando é o uso de parênteses duplo:
`((expressão))`

```
$ ((0 <= 5)) ; echo $?
0
```

Essa é uma sintaxe semelhante a da linguagem C. Outro uso comum é no incremento de variáveis:

```
let var++  # equivalente a "var=$[ $var + 1 ]"

let var--  # equivalente a "var=$[ $var - 1 ]"
```

Exemplo:

```
$ num=102
$ let num++
$ echo $num
103
$ num=$[$num + 1]
$ echo $num
104
$ num=$((num+1))
$ echo $num
105
$ let num=num+1
$ echo $num
106
$ let num+=4
$ echo $num
110
```

6.5 Testes de *Strings*

O tamanho de uma *string* pode ser obtido pelo uso do comando `expr length` *`string`*.
Ex:

```
$ expr length palavra
7
$
```

Ex:

```
#!/bin/bash

echo "Digite a senha: "
read -s  senha
```

```
comp=$(expr length $senha)

if [ "$comp" -lt 6 -o "$comp" -gt 9 ]
 then
    echo "Senha Inválida."
    echo "Por segurança não são aceitas senhas com menos de
    6 caracteres ou mais que 9."
    echo "Informe uma nova senha."
 else
    echo "Senha aceita."
fi
```

Os operadores para testes de *string* podem ser:

Binários

- `=` - Retorna verdadeiro se as duas *strings* forem iguais.

```
$ str=palavra
$ [ "$str" = "cadeia" ]
$ echo $?
1
```

- `!=` - Retorna verdadeiro se as duas *strings* forem diferentes.

Unários

- `-z` - Retorna verdadeiro se o comprimento da *string* é igual à 0.
- `-n` - Retorna verdadeiro se o comprimento da *string* é diferente de 0.

```
$ [ -n $str ]
$ echo $?
0
```

A sintaxe para a comparação de *string* segue o mesmo modelo para a comparação numérica:
`["var/string" relação "var/string"]`
A preferência para o uso de aspas foi dada porque geralmente as *strings* contêm espaços em branco. Assim, são evitados erros de interpretação pelo *shell*.

6.6 Testes de arquivos

Sempre que vamos trabalhar com arquivos é necessário realizar testes como os mesmos para evitar erros. Para testar arquivos existem as opções:

- -d – O arquivo é um diretório.
- -e – O arquivo existe.
- -f – É um arquivo normal.
- -s – O tamanho do arquivo é maior que zero.
- -r – O arquivo tem permissão de leitura.
- -w – O arquivo tem permissão de escrita.
- -x – O arquivo tem permissão de execução.

A sintaxe usada deve ser:
`[ opção arquivo ]`
Ex:

```
$ ls -l  arqu1.txt data
-rwxrwxrwx    1 pet  linux       161   2005-04-09 21:42   arqu1.txt
-rw-r--r--    1 pet  linux        95   2005-04-07 22:00   data
drwxr-xr-x    3 pet  linux      1312   2005-01-07 22:57   scripts
$ [ -d scripts ]
$ echo $?
0
$ [ -d arqu1.txt ]
$ echo $?
1
$ [ -e arqu1.txt ]
$ echo $?
0
$ [ -f arqu1.txt ]
$ echo $?
0
$ [ -r data ]
$ echo  $?
0
$ [ -x data ]
$ echo $?
1
```

Capítulo 7

Controle de fluxo

Para as linguagens de programação, uma das mais importantes estruturas é o controle de fluxo. Com o *shell* não é diferente. Com ele, a execução do programa pode ser alterada de forma que diferentes comandos podem ser executados ou ter sua ordem alterada de acordo com as decisões tomadas. São realizados saltos, desvios ou repetições. Nas próximas seções explicaremos cada tipo de estrutura.

7.1 Decisão simples

A estrutura de decisão simples é aquela que realiza desvios no fluxo de controle, tomando com base o teste de uma condição dada, uma opção entre duas que podem ser escolhidas.

Uma decisão simples é uma construção com os comandos `if/then`. Isso representa **se** condição **então** realiza determinado comando.

Sintaxe:

```
if[expressão]; then

    comando
fi
```

Ex: Este programa bem simples mostra o uso do **if**. Há um arquivo chamado `livro.txt` cujo conteúdo segue abaixo:

```
#LIVRO          #EXEMPLARES
##########################
eletromagnetismo 5
redes            4
cálculo          8
física           6
eletrônica       7
```

O programa abaixo mostra o número de exemplares de um determinado assunto. Mas, primeiro, é verificado se o livro está na lista.

```
#!/bin/bash

echo -n  "Qual livro você deseja? "
```

```
read Livro

if grep $Livro livro.txt>>/dev/null
  then echo "O livro $Livro possui `grep $Livro livro.txt|cut -f2` exemplares."

  else echo "Este livro não está na lista"

fi
```

Obs: O diretório `/dev/null` é um lugar para onde redirecionamos a saída de um comando quando não é desejável que ela apareça no *prompt*.

7.2 Decisão múltipla

Este tipo de estrutura engloba, além dos comandos vistos anteriormente, os comandos `elif` e `else`. Neste caso, se a condição dada não for satisfeita, há outro caminho a ser seguido, dado pelo `elif`, que seria a combinação de **else** com **if** (senão se...). A diferença entre o uso de `elif` e `else if` é que, se fosse usado o último, seria necessário usar o `fi`.

Sintaxe:

```
if [expressão]; then

    comando

elif [expressão]; then

    comando

elif [expressão]; then

    comando
...

else

    comando

fi
```

7.2.1 O comando `case`

Outra forma de fazer desvios múltiplos é pelo uso do comando `case` . Ele é semelhante ao `if` pois representa tomada de decisão, mas permite múltiplas opções. Esta estrutura é bastante usada quando é necessário testar um valor em relação a outros valores pré- estabelecidos, onde cada um desses valores tem um bloco de comando associado.

Ex: Este exemplo dá o Estado de acordo com o DDD digitado.

```
echo "Insira o código DDD: "

read cod

case $cod in
  21) echo "Rio de Janeiro";;
  11) echo "São Paulo";;
  3[0-8]) echo "Minas Gerais";;
  *) Echo "Insira outro código";;

esac
```

Decisão com && e || Esses caracteres permitem a tomada de decisões de uma forma mais reduzida. A sintaxe usada é:

```
comando1 && comando2

comando1 || comando2
```

O && faz com que o `comando2` só seja executado se `comando1` retornar 0, ou seja, se `comando1` retornar verdadeiro o `comando2` é executado para testar se a condição toda é verdadeira.

O || executa o `comando2` somente se o `comando1` retornar uma valor diferente de 0, ou seja, somente se `comando1` retornar falso é que `comando2` será executado para testar se a condição toda é verdadeira.

Ex: Se o arquivo `livro.txt` realmente existir será impresso na tela: O arquivo existe.

```
[ -e livro.txt ] && echo "Arquivo Existe"
```

Ex: Se o diretório `NovoDir` não existir é criado um diretório com o mesmo nome.

```
cd NovoDir 2> /dev/null || mkdir NovoDir
```

7.3 Controle de *loop*

Existem 3 tipos de estruturas de *loop*, que serão vistas na próxima seção. Esse tipo de estrutura é usada quando é preciso executar um bloco de comandos várias vezes.

7.3.1 *While*

Nesta estrutura é feito o teste da condição, em seguida ocorre a execução dos comandos. A repetição ocorre enquanto a condição for verdadeira.

```
while <condição> do

    <comandos>

done
```

`condição` pode ser um teste, uma avaliação ou um comando.
Ex:

```
#!/bin/bash

echo "Tabela de Multiplicação do 7: "

i=7;
n=0;
while [ $n -le 10 ]
do
  echo $i x $n = $(($i * $n))
  let n++
done
```

7.3.2 *Until*

Neste caso, a repetição ocorre enquanto a condição for falsa. Ou seja, primeiramente a condição é testada, se o código de retorno for diferente de zero os comandos são executados. Caso contrário, a execução do programa continua após o *loop*.

```
until <condição> do

    <comandos>

done
```

`condição` pode ser um teste, uma avaliação ou um comando.
Ex: Este exemplo é semelhante ao exemplo anterior do comando `while`. O que mudou foi a condição de teste.

```
#!/bin/bash

echo "Tabela de Multiplicação do 7: "

 i=7;
n=10;
until [ $n -eq 0 ]
do
   echo $i x $n = $(($i * $n))
   let n--
done
```

7.3.3 *For*

A sintaxe da estrutura `for` é a seguinte:

```
for variavel in lista do

    <comandos>

done
```

Seu funcionamento segue o seguinte princípio: `variavel` assume os valores que estão dentro da lista durante os *loops* de execução dos `comandos`. As listas podem ser valores passados como parâmetros, dados de algum arquivo ou o resultado da execução de algum comando. Com o exemplo abaixo fica mais fácil de entender isso.

Ex: Este programa cria diretórios com o nome diretorioNUMERO, onde NUMERO vai de 1 à 5.

```
#!/bin/bash


for i in ‘seq 1 5‘
do
    mkdir diretorio$i
done
```

O comando `seq NumInicial NumFinal` faz uma contagem seqüencial do número inicial dado até o número final.

Para estes 3 tipos de construção de *loops*, existem dois comandos que permitem alterar a rotina de sua execução . São eles:

- `break [n]` - Este comando aborta a execução do *loop* e salta para a próxima instrução após o *loop*.
- `continue [n]` - Este comando faz com que o fluxo de execução do programa volte para o início do *loop* antes de completá-lo.

  Ex: O Exemplo abaixo ilustra o uso de `break` e do `continue`

```
#!/bin/bash

echo "Tente acertar o número "
echo "Dica: Ele está entre 10 e 50. "

i=1
while true
do
  echo "Digite o Número: "
  read num
  if [ $num != 30 ]
    then
      echo "Você errou. Tente outra vez"
      let i++
      continue
```

```
    fi

    if [ $num == 30 -a $i == 1 ]
      then
        echo Você acertou de primeira. Parabéns!
        break
    fi
    if [ $num == 30 ]
      then
        echo Você acertou após $i tentativas.
        break
    fi

done
```