


Shell Script

- Andre Luiz
 - Giuliano Lacerda
-
- 


Índice

1. Introdução e História
 2. Tipos de Dados
 3. Variáveis e Constantes
 4. Expressões e Comandos
 5. Modularização
 6. Polimorfismo
 7. Exceções
 8. Concorrência
 9. Avaliação da linguagem
 10. Conclusão
 11. Referencias
-

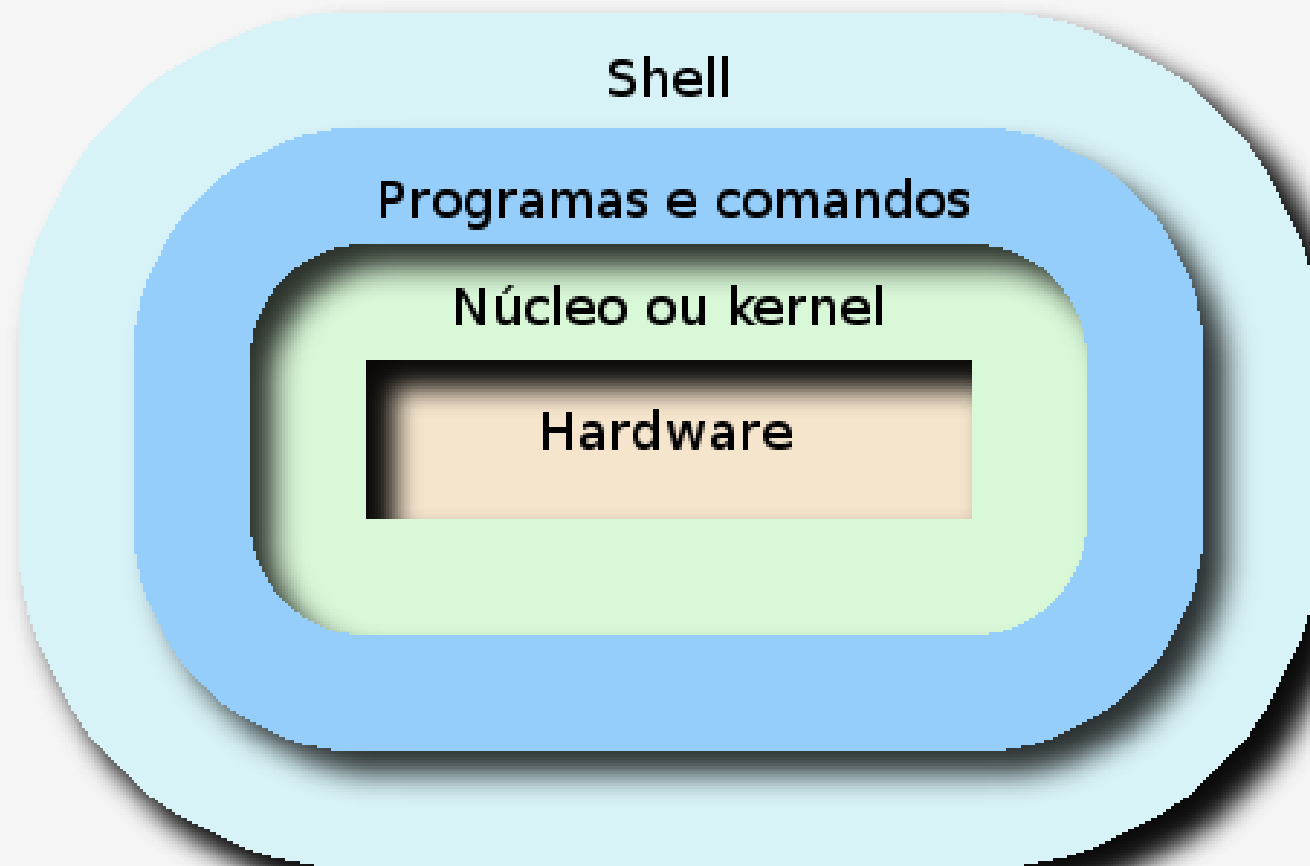


Introdução

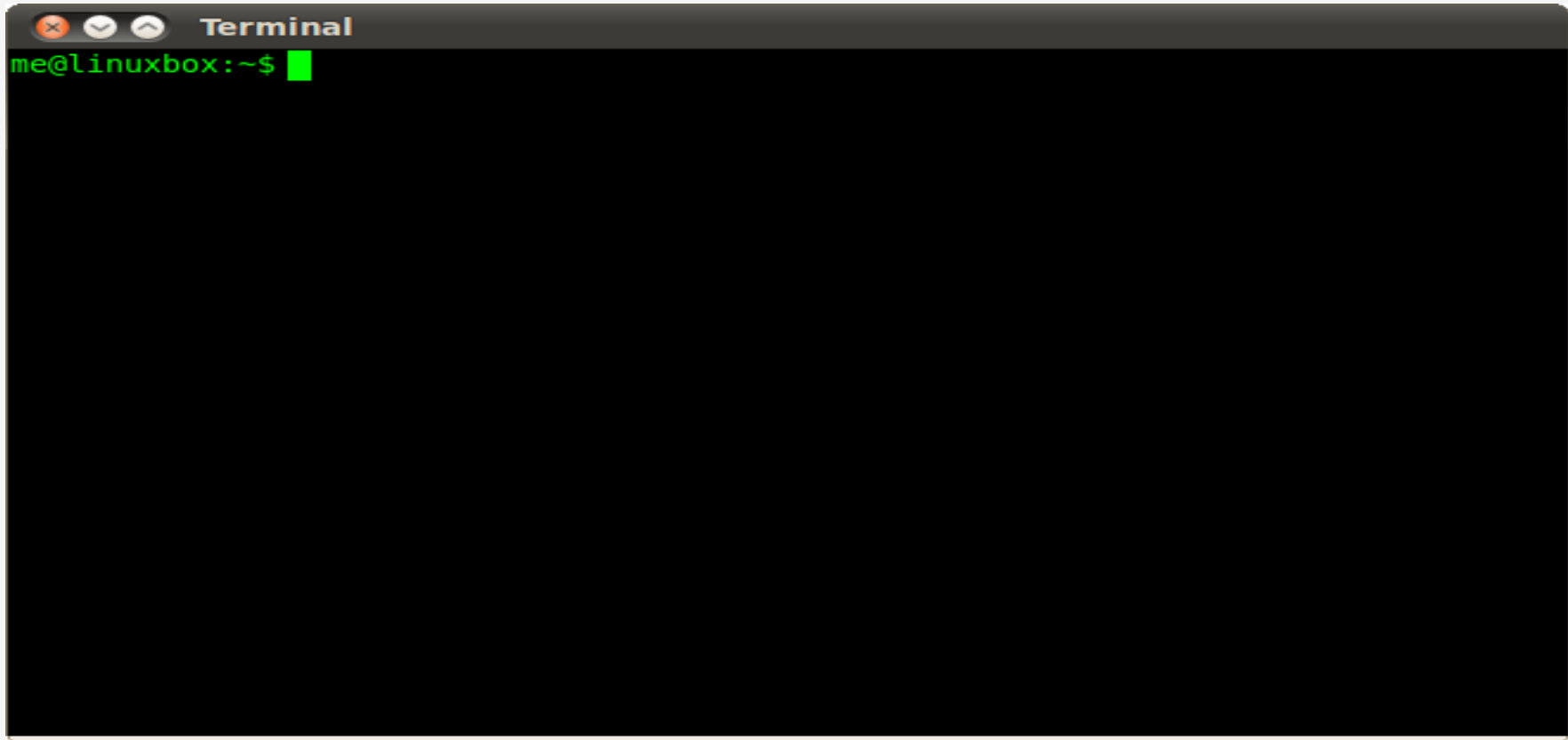
O que é Shell ?



Visão Geral em camadas



Shell



Linux – Terminal

Windows – Prompt de Comandos

Histórico


O primeiro shell do Unix foi o Thomson shell. Criado por Ken Thompson, foi distribuído entre as versões de 1 a 6 do Linux, durante 1971 a 1975. Ele era muito primitivo, somente com estruturas de controle básicas e sem variáveis.

Mas o que vem a ser Shell Script ?



Mas o que vem a ser Shell Script ?

Uma linguagem que utiliza o shell para realizar ações automatizadas através de seus scripts, codigos e comandos.



Linguagem Script

São linguagens de programação executadas do interior de programas e/ou de outras linguagens de programação, não se restringindo a esses ambientes.

As linguagens de script servem para estender a funcionalidade de um programa e/ou controlá-lo, acessando sua API e, são frequentemente usadas como ferramentas de configuração e instalação em sistemas operacionais

Tipos de Shell (Principais)

Bourne Shell: é o shell padrão para Unix, ou seja, a matriz dos outros shells, portanto é um dos mais utilizados. É representado por "sh". Foi desenvolvido por Stephen Bourne, por isso Bourne Shell.

Korn Shell: este shell é o Bourne Shell evoluído, portando todos os comandos que funcionavam no Bourne Shell funcionarão neste com a vantagem de ter mais opções. É representado por "ksh".

C Shell: é o shell mais utilizado em BSD, e possui uma sintaxe muito parecida com a linguagem C. Este tipo de shell já se distancia mais do Bourne Shell, portanto quem programa para ele terá problemas quanto a portabilidade em outros tipos. É representado por "csh".

Bourne Again Shell: é o shell desenvolvido para o projeto GNU usado pelo [GNU/Linux](#), é muito usado pois o sistema que o porta evolui e é adotado rapidamente. Possui uma boa portabilidade, pois possui características do Korn Shell e C Shell. É representado por "**bash**". O nosso estudo estará focado neste.

Qual o shell corrente no seu linux ?

```
root@kali:/home/shell# echo $0  
bash  
root@kali:/home/shell#
```

Born Again Shell - Bash

Bash é o shell, ou interpretador de comandos da linguagem do sistema operacional GNU.

O Unix Shell é ao mesmo tempo um **interpretador** de comandos e uma **linguagem de programação**.

Como interpretador de comandos, ele dá acesso ao rico conjunto de utilidades do GNU.

Como linguagem de programação ele permite que tais utilidades sejam combinadas. Arquivos contendo comandos podem ser criados e se tornar comandos. Esses novos comandos tem o mesmo status de comandos de sistema como os do diretório /bin.

*Shell é uma linguagem
totalmente interpretada!*



Palavras reservadas

!: Pipelines

[]: Conditional Constructs

{ }: Command Grouping

case: Conditional Constructs

do: Looping Constructs

done: Looping Constructs

elif: Conditional Constructs

else: Conditional Constructs

esac: Conditional Constructs

for: Looping Constructs

function: Shell Functions

if: Conditional Constructs

in: Conditional Constructs

select: Conditional Constructs

then: Conditional Constructs

time: Pipelines

until: Looping Constructs

while: Looping Constructs

Criando um Shell Script

1 – Crie um arquivo que possa ser editado **shell1.sh**

```
mago@magobi:~$ touch shell1.sh
```

Criação do arquivo shell1.sh

Onde: touch - Comando utilizado para criar um arquivo vazio.

Criando um Shell Script

2 - Dê direito de execução para o arquivo criado

Para que seja possível executar o shell script é preciso atribuir ao mesmo o direito de execução, para isso é necessário usar o comando `chmod` com a opção `+x`.

```
mago@magobi:~$ chmod +x shell1.sh
```

Atribuindo direito de execução ao arquivo **shell1.sh**

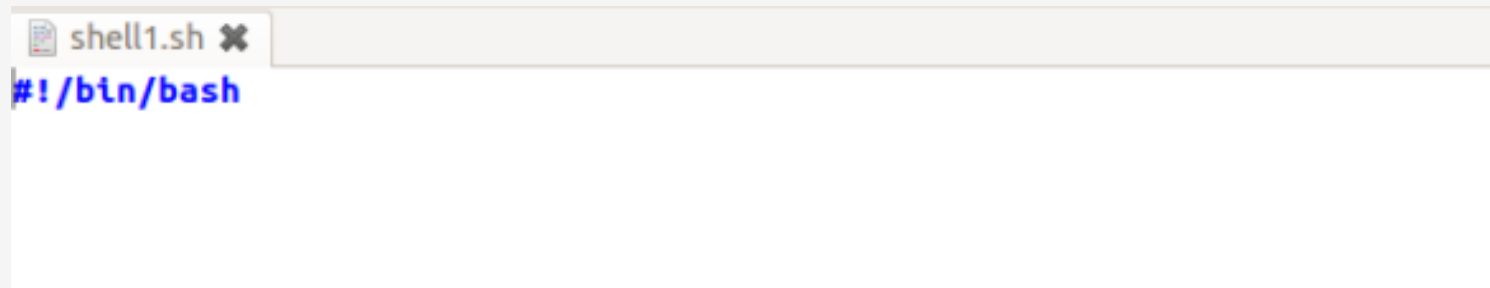
Antes de atribuir o direito de execução, o arquivo era visto pelo terminal da cor padrão (neste caso preto), com o direito de execução agora passou a ser verde.

```
mago@magobi:~$ ls *.sh  
shell1.sh  
mago@magobi:~$
```

Criando um Shell Script

3. Abra o arquivo recém criado para editá-lo

Abaixo vemos o código quando editado pelo gedit.



```
shell1.sh ✕  
#!/bin/bash
```

A primeira linha escrita do shell script é utilizada para informar qual modelo de shell será utilizado para criar o shell script (ksh, sh, bash). Neste exemplo o shell responsável será o bash que está localizado na pasta /bin.

Criando um Shell Script

Após a linha que contém o código referente ao tipo de shell é possível inserir os comandos desejados.

```
#!/bin/bash  
echo "Hello, World!"
```

Salve e feche o arquivo.

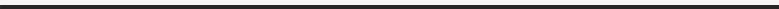
Execute:

```
./shell1.sh
```

Saida:

```
Hello, World!
```

Tipos de dados



Tipagem fraca

Ocorre quando a linguagem permite que uma variável tenha seu valor automaticamente alterado para outro tipo para possibilitar uma operação.

Tipagem dinâmica

Ocorre quando a linguagem não obriga a prévia declaração de tipo de uma variável. O tipo é assumido na atribuição de valor à variável, que pode ser por presunção ou forçado com casting. Além disso, é possível modificar o tipo da variável atribuindo-lhe outro valor.

Exemplos

Tipagem fraca:

```
var=1
```

```
var="Agora sou uma string"
```

```
var=12.5
```

```
var[0]=a
```

Tipagem dinâmica:

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

```
cor_casa=VERDE
```

```
echo "A cor da casa é $cor_casa"
```

Arrays

```
array[0]="AULA"
```

```
array[1]="DE"
```

```
array[2]="LP"
```

Ou :

```
array=("AULA" "DE" "LP")
```

Arrays

```
echo "${array[0]}"  
echo "${array[1]}"
```

Saída:
AULA DE

```
echo "${array[@]}" → Imprime todo o conteúdo
```

Saída:
AULA DE LP

```
echo "${#array[@]}" → Imprime a quantidade  
de elementos  
Saída:  
3
```

Arrays

Imprime todos os conteúdos:

```
echo ${var[@]} echo ${var[*]}
```

imprime todos os índices:

```
echo ${!var[@]} echo ${!var[*]}
```

Arrays

```
$ declare -A valores  
valores=( [valor1]=1 [valor2]=2 [valor3]=3 )
```

ou

```
valores[valor1]=1  
valores[valor2]=2  
valores[valor3]=3
```

1) Obtendo as chaves:

```
$ echo ${!valores[@]}
```

Saída:

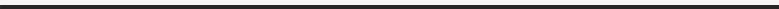
```
valor1 valor2 valor3
```

2) Obtendo os valores das chaves:

```
$ echo ${animais[@]}
```

Saída: 1 2 3

Variaveis e constantes



Variáveis e constantes

- Variáveis devem ser declaradas junto com seu valor.
 - Locais e Globais
 - Todas as variáveis são globais por definição a não ser que os comandos “local” ou “declare” sejam usados.
-

Variaveis Globais

Elas são visíveis para todas as sessões shell, e para qualquer processo filho que o shell criar.

Isso torna variáveis globais úteis para processos filhos que requerem informações de processos pai.

Exemplo

```
#!/bin/bash

func ()
{
    var=23
}

func

echo "$var"
```

Saída: 23

Observe que var é uma variável global.

Outro ponto importante é que para acessar o conteúdo da variável é necessário o uso do símbolo dólar (\$).

Variáveis Locais

Estão disponíveis apenas para o shell que as criou



Exemplo

```
#!/bin/bash

func ()
{
    local var=23
}

func

echo "$var"

O Shell retornará um erro.
```

- Comando local só pode ser usado dentro de uma função .
- Var tem um escopo visível restrita a esta função.

Constantes

Constantes são criadas utilizando o comando **readonly**.

```
readonly const=1  
const=var;
```

Neste exemplo temos um erro pois tentamos alterar o valor da constante **const**.

Declare

Atraves do comando **declare** podemos especificar o tipo da variável

```
declare -i var=5  
var="string"
```

Neste caso **var = 0** pois ela apenas aceita inteiros e nesse caso atribuimos uma string.

- i variável é um inteiro
- a variável é um vetor (array)
- f lista todas as funções declaradas
- p mostra os atributos e valores de cada variável
- r faz comque as variáveis sejam readonly (constantes)

Declare

Este comando também serve para restringir o escopo de uma variável:

Exemplo:

```
funcao ()  
{  
  a=1  
}
```

```
funcao  
echo $a
```

bar # Imprime bar.

Porem...

```
funcao ()  
{  
  declare a=1  
}
```

```
funcao  
echo $a
```

bar # Imprime nada.

Strings

ex 1:

```
var=testando)
```

```
echo $var
```

ex 2:

```
var=testando) esse comando
```

```
echo $var
```

Strings

ex 1:

```
var = testando
```

```
echo $var
```

Saida: testando

ex 2:

```
var = testando esse comando
```

```
echo $var
```

Strings

ex 1:

```
var = testando
```

```
echo $var
```

ex 2:

```
var = testando esse comando
```

```
echo $var
```

Erro

Saida: -bash: a: command not found

Forma correta:

```
var='testando esse comando'
```

Isso porque, quando temos uma string que contém espaço devemos utilizar aspas.

Interpolação

Recurso utilizado com a finalidade de acrescentar variáveis ou algum tipo de dado diferente de string dentro de uma strings sem precisar de conversão nem concatenação.


Interpolação

```
var1=shell  
var2=script
```

```
var3="$var1 $var2"  
echo $var3  
Saída: shell script
```

```
var3='$var1 $var2'  
echo $var3  
Saída: $var1 $var2
```

O valor pode ser expressado entre aspas (“”), apóstrofos (”) ou crases (``).



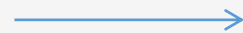
Podemos utilizar crase (`) no script para chamar comandos do sistema.

```
echo "O usuario eh: `users`"
```

```
echo "Estou no diretorio: `pwd`"
```

Podemos utilizar crase (`) no script para chamar comandos do sistema.

```
echo "O usuario eh: `users`"
```



```
O usuario eh: a2011100772
```

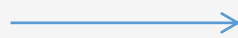
```
echo "Estou no diretorio: `pwd`"
```



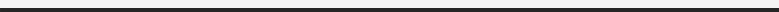
Podemos utilizar crase (`) no script para chamar comandos do sistema.

```
echo "O usuario eh: `users`"
```

```
echo "Estou no diretorio: `pwd`"
```




```
Estou no diretorio: /home/trab
```



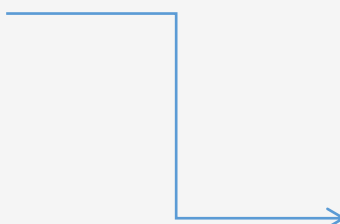
Não pode haver espaço entre as variáveis, o sinal de igual e o valor.

`$ a = 1`



Não pode haver espaço entre as variáveis, o sinal de igual e o valor.

```
$ a = 1
```



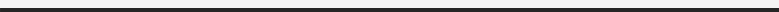
```
-bash: a: command not found
```



Não pode haver espaço entre as variáveis, o sinal de igual e o valor.

Forma correta

\$ a=1



Gerenciamento de Memória

As variáveis são criadas como variáveis de ambiente, deixando o sistema operacional responsável pela gerência da memória.

Variáveis de ambiente

São variáveis que guardam informações sobre preferências pessoais usadas por programas para que eles peguem dados sobre seu ambiente sem que você tenha que passar sempre os mesmos dados.

- As variáveis de ambiente normalmente são escritas em letras maiúsculas.

Variáveis de ambiente

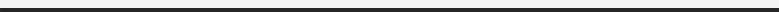
Variáveis Default

\$TERM

Define o terminal padrão.

\$HOME

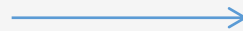
Indica o diretório pessoal do usuário em questão.



Variáveis de ambiente

Variáveis Default

\$TERM
Define o terminal padrão.



```
$ echo $TERM  
xterm
```

\$HOME
Indica o diretório pessoal do usuário em questão.

Variáveis de ambiente

Variáveis Default

\$TERM

Define o terminal padrão.

\$HOME

Indica o diretório pessoal do usuário em questão.

→
\$ echo \$HOME
/home/andre



Variáveis de ambiente

Variáveis Default

\$TERM

Define o terminal padrão.

\$HOME

Indica o diretório pessoal do usuário em questão.

→ \$ echo \$HOME
/home/andre

Essa variável é muito usada em scripts que necessitam saber qual o diretório pessoal do usuário. A própria variável retorna o valor automaticamente. E esse script pode ser usado por qualquer usuário que tenha permissão de executá-lo.

Variáveis de ambiente

Variáveis Default

\$USER

Guarda o nome do usuário no momento.

\$SHELL

Guarda o valor do shell padrão:

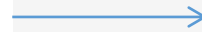
\$TMOUT

Variáveis de ambiente

Variáveis Default

\$USER

Guarda o nome do usuário no momento.



```
$ echo $USER  
giuliano
```

\$SHELL

Guarda o valor do shell padrão:

\$TMOUT

Variáveis de ambiente

Variáveis Default

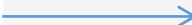
\$USER

Guarda o nome do usuário no momento.

\$SHELL

Guarda o valor do shell padrão:

\$ echo \$SHELL
/bin/bash



\$TMOUT

Variáveis de ambiente

Variáveis Default

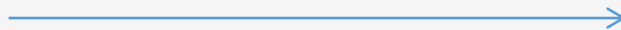
\$USER

Guarda o nome do usuário no momento.

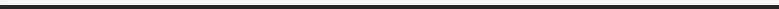
\$SHELL

Guarda o valor do shell padrão:

\$TMOUT



```
TMOUT=30  
$ export TMOUT
```



Variáveis de ambiente

Variáveis Default

\$USER

Guarda o nome do usuário no momento.

\$SHELL

Guarda o valor do shell padrão:

\$TMOUT

Essa variável define o tempo máximo que o shell ficará inativo.

Com esse comando, se você sai e deixa o terminal de texto aberto, após 30 segundos de inatividade o shell se fecha.

Expressões e Comandos



Operadores

Operadores Aritméticos	
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Módulo
**	Exponenciação

Operadores

Operadores de Atribuição	
=	Atribui valor a uma variável
+=	Incrementa a variável por uma constante
-=	Decrementa a variável por uma constante
*=	Multiplica a variável por uma constante
/=	Divide a variável por uma constante
%=	Resto da divisão por uma constante
++	Incrementa em 1 o valor da variável
--	Decrementa em 1 o valor da variável

Operadores

Operadores de BIT	
<<	Deslocamento à esquerda
>>	Deslocamento à direita
&	E de bit (AND)
	OU de bit (OR)
^	OU exclusivo de bit (XOR)
~	Negação de bit
!	NÃO de bit (NOT)
Operadores de BIT (atribuição)	
<<=	Deslocamento à esquerda
>>=	Deslocamento à direita
&=	E de bit
=	OU de bit
^=	OU exclusivo de bit

Concatenar Strings

```
var1="Sou uma string"  
var2="em shell script"  
var3=" $var1 $var2"  
echo $var3
```



Concatenar Strings

```
var1="Sou uma string"  
var2="em shell script"  
var3=" $var1 $var2"  
echo $var3
```



```
Sou uma string em shell script
```

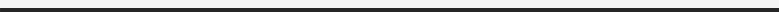
```
vetor[0]=Sou  
vetor[1]=uma  
vetor[2]=string
```

```
#{vetor[@]}
```

vetor[0]=Sou
vetor[1]=uma
vetor[2]=string

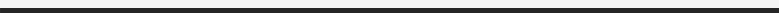
`${vetor[@]}`

Sou uma string



Alterando uma String para um vetor

```
vetor=(${var3})
```



Alterando um Vetor para uma String

```
var=${vetor[#]}
```



Comparadores

Comparação Numérica	
-lt	É menor que (LessThan)
-gt	É maior que (GreaterThan)
-le	É menor igual (LessEqual)
-ge	É maior igual (GreaterEqual)
-eq	É igual (Equal)
-ne	É diferente (NotEqual)
Comparação de Strings	
=	É igual
!=	É diferente
-n	É não nula
-z	É nula

Operações condicionais

Exemplo:

```
if [ $num -lt 5]
then
  {...}
elif (( $num <= 10 ))
Then
  {...}
else
  {...}
fi
```

Operações condicionais

Exemplo:

```
if [ $string == "string" ] && (( $num == 1 ))  
then  
    {...}  
elif [ -n string ]  
then  
    {...}  
else  
    {...}  
fi
```


Comandos de repetição

Exemplos:

```
for ((i=1;i<=10;i++))  
do  
  {...}  
done
```

```
for i in "for" "com" "string"  
do  
  {...}  
done
```

```
for i in `seq 1 10`  
do  
  {...}  
done
```

Comandos de repetição

Exemplos:

```
while [ $c -le 5 ]  
do  
    echo "Welcome $c times"  
    (( c++ ))  
done
```

Leitura de Arquivos

```
for line in $(cat arquivo.txt)
do
  [comando]
done
```

Escrita em Arquivos

```
echo "algo a ser escrito" > arquivo
```

ou

```
echo "algo a ser escrito">>> arquivo
```

Blocos e Agrupamentos

{...} - Agrupa comandos em um bloco

(...) - Executa comandos numa subshell

\$(...) - Executa comandos numa subshell, retornando o resultado

((...)) - Testa uma operação aritmética, retornando 0 ou 1

\$((...)) - Retorna o resultado de uma operação aritmética

[...] - Testa uma expressão, retornando 0 ou 1

[[...]] - Testa uma expressão, retornando 0 ou 1 (podendo usar && e ||)

Comandos básicos

Comando	Descrição	Sintaxe
echo	Exibe o texto na tela	echo "texto a ser mostrado"
sleep	Dá um tempo antes de seguir	sleep segundos exemplo: Sleep 1
read	Recebe o valor de uma variável	read variável exemplo: read dados
>	Escreve num arquivo texto <i>(apagando o que estava lá)</i>	echo "texto" > /home/luiz/arquivo
>>	Escreve num arquivo texto <i>(na ultima linha)</i>	echo "texto" >> /home/luiz/arquivo
&	Roda o comando em 2º plano e continua o script	Comando
exit	Sai do script	exit
touch	Cria arquivos texto	touch arquivo
#	Comenta tudo depois deste simbolo	# Comentário

Comandos básicos

Diretórios		
Comando	Sintaxe	Descrição
rm -rf	rm -rf +diretório	Deleta arquivos/pastas e tudo que estiver dentro (cuidado)
pwd	pwd	Mostra em qual diretório estamos
chmod	chmod 777 arquivo_ou_pasta	Muda as permissões, 777 = permissão total
chown	chown user:grupo arq_ou_diret.	Muda o proprietário de arquivos e pastas
cd	cd diretório	Entra em diretórios

Comandos básicos

Usuários		
Comando	Sintaxe	Descrição
useradd	useradd luiz -g alunos (no grupo)	Adiciona um usuário
userdel	userdel usuário	Deleta usuário e seus arquivos
groupdel	groupdel grupo	Deleta um grupo
groups	groups nome_usuario	Mostra os grupos do usuário
addgroup	addgroup usuario grupo ou addgroup nomedogruppo	Cria um grupo ou adiciona um usuário ao grupo
sudo	sudo comando	Executa comandos como root
whoami	whoami	Identifica com qual usuário você esta logado

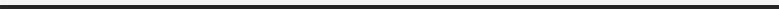
Comandos básicos

Rede		
Comando	Sintaxe	Descrição
ifconfig	ifconfig	Mostra as interfaces de rede
hostname	hostname	Mostra ou muda o nome de seu computador na rede
ping	Ping ip_desejado	Dispara pacotes para outro pc, para testar conexões etc

Comandos básicos

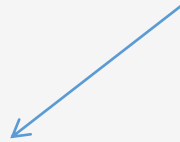
Comando	Sintaxe	Descrição
killall	Killall nome_do_programa	Mata um processo
whatis	Whatis +nome do programa	Descreve o que faz o comando
diff	diff arquivo1 arquivo2	Compara 2 arquivos
ps	ps -elf	Mostra os programas que estão rodando
cat	cat arquivo_texto	Mostra o conteúdo de um arquivo de texto
grep	Comando grep palavra	Filtra a saída do comando, mostra a linha da palavra pedida
ln	ln -s arquivo_original atalho	Cria atalho
cp	cp arquivo destino	Copia um arquivo ou diretório (-R para diretórios)
apt-get	apt-get nome_programa	Instala aplicativos
find	Find +nome	Procura por arquivos e diretórios

Modularização



A modularização é feita através de Funções

Criando a função



```
imprime ()  
{  
  echo "Sou o programa $0"  
  echo "Recebi $# parametros"  
  echo "Param 1: $1"  
  echo "Param 2: $2"  
  echo "Lista de parâmetros: $*"  
}
```



A modularização é feita através de Funções

Chamando a função



```
imprime ()  
{  
  echo "Sou o programa $0"  
  echo "Recebi $# parametros"  
  echo "Param 1: $1"  
  echo "Param 2: $2"  
  echo "Lista de parâmetros: $*"  
}
```

```
imprime um dois tres quatro
```



A modularização é feita através de Funções

```
imprime ()  
{  
  echo "Sou o programa $0"  
  echo "Recebi $# parametros"  
  echo "Param 1: $1"  
  echo "Param 2: $2"  
  echo "Lista de parâmetros: $*"  
}
```

```
imprime um dois tres quatro
```

resultado



```
Sou o programa teste.sh  
Recebi 4 parametros  
Param 1: um  
Param 2: dois  
Lista de parâmetros: um dois tres quatro
```

A modularização é feita através de Funções

Observamos que nas funções, nós não declaramos os tipos e nem mesmo quantos argumentos a mesma irá receber.

Polimorfismo

Não possui!



Exceções

Não possui tratamento de Exceções!



Concorrença

Não possuí tratamento!



Avaliação da linguagem

- Facilidade de aprendizado
 - Baixa legibilidade
 - Baixa redigibilidade
 - Baixa confiabilidade
 - Baixa eficiência
 - Prática para rotinas e sub-rotinas de sistemas.
-

Conclusão

O shell script é uma linguagem altamente recomendada para criar rotinas e sub-rotinas de sistemas, por lidar diretamente com comandos internos e ter acesso direto a executáveis. Porém, para projetos maiores, não é recomendada, por ser de difícil escrita e leitura.

Referencias

<http://www.devmedia.com.br/introducao-ao-shell-script-no-linux/25778>

[http://bash.cyberciti.biz/guide/Hello, World! Tutorial](http://bash.cyberciti.biz/guide/Hello,_World!_Tutorial)

http://pt.kioskea.net/faq/2269-como-ler-um-arquivo-linha-por-linha#simili_main

<https://www.youtube.com/watch?v=CMp6H4A9AJU>

http://www.inf.ufes.br/~mberger/Disciplinas/2015_1/EDII/trab3.pdf

<http://www.inf.ufes.br/~vitorsouza/wp-content/uploads/teaching-lp-20142-seminario-shell.pdf>
