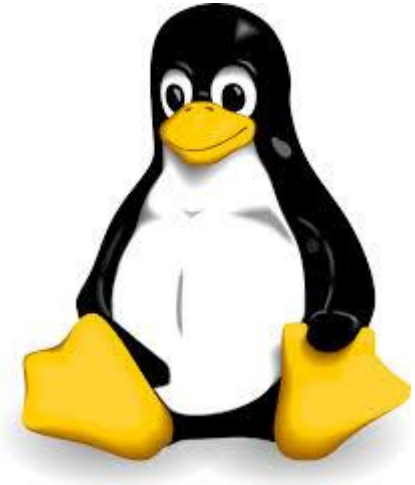
SHELL SCRIPT



André Paris/Marcos Cardoso/Patrick Januário

Índice

1- INTRODUÇÃO E HISTÓRIA

O primeiro unix shell

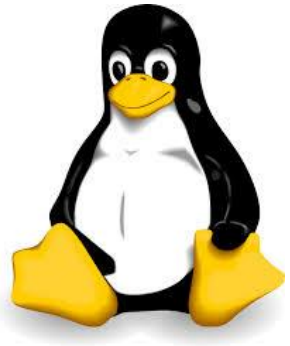
A primeira geração do shell é descendente do Multic shell. O primeiro deles foi Thomson shell. Criado por Ken Thompson, foi distribuído entre as versões de 1 a 6 do Linux, durante 1971 a 1975. Foi o primeiro Unix shell e era muito primitivo, somente com estruturas de controle básicas e sem variáveis.

Shell Script

A linguagem Shell Script é a linha de comando Linux (Unix), os quais armazenados em um arquivo texto são executados sequencialmente. Esta linguagem interpreta o script do usuário no terminal. Além de executar comandos do sistema, o Shell também tem seus próprios comandos, e também possui variáveis e funções. Existem diversos tipos de shell: bash, csh, ksh, ash, etc. O mais utilizado atualmente é o bash (GNU Bourne-Again SHell). A primeira linha de todo Shell script deve começar com algo do tipo: #!/bin/bash, a qual indica com qual Shell deverá ser executado o script.

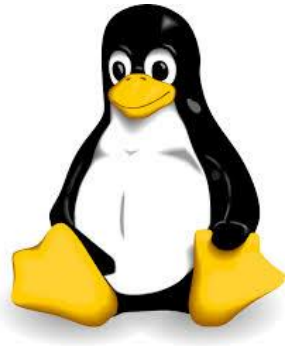
BASH

O bash é um interpretador de comandos da linguagem do sistema operacional. Também é conhecido como shell. Através dele é possível executar uma sequência de comandos direto no prompt do sistema ou escrito em arquivos de texto, conhecidos como shell script. Foi liberado em 1989 e é mantido pelo projeto GNU.



O que é um script?

LINGUAGEM SCRIPT



Esse tipo de linguagem é voltado para tarefas de menor escala, como criação de rotinas de execução corriqueiras (macros) e integração entre aplicações existentes (a saída de uma aplicação servir de entrada para outra). São linguagens de programação executadas do interior de programas e/ou de outras linguagens, não se restringindo a esses ambientes.

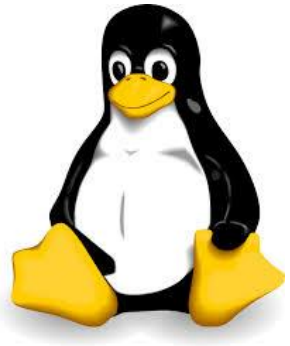
MOTIVAÇÃO

- **Linguagens script são mais flexíveis e permitem a criação de programas através da combinação de componentes já existentes;**
- **São, às vezes, chamadas de glue languages, pois permitem a junção de programas menores para se construir um grande sistema; e**
- **Também são vistas como linguagens de extensão, pois permitem que algumas ferramentas sejam estendidas com estas linguagens.**

TODA LINGUAGEM SCRIPT É INTERPRETADA

Em contraste com o ambiente de desenvolvimento os scripts são, geralmente, escritos em linguagens compiladas e distribuídos na forma de código de máquina.
No caso do shell script, ele é uma linguagem interpretada e Imperativo.

PALAVRAS RESERVADAS



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

2- TIPOS DE DADOS

classificação

- **Tipagem fraca, ou seja, ocorre quando a linguagem permite que uma variável tenha seu valor automaticamente alterado para outro tipo para possibilitar uma operação; e**
- **Tipagem dinâmica, ou seja, 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 DE TIPAGEM

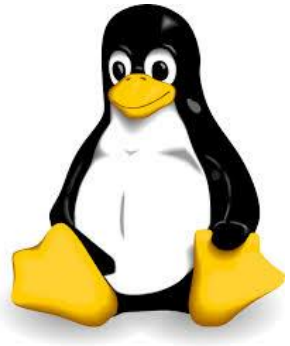
Exemplo de tipagem fraca:

```
var=1
var="Now I am a String"
var=12.5
var[0]=a
```

Exemplo de tipagem dinâmica:

```
#!/bin/bash
cor_casa=VERDE
echo "A cor da casa é $cor_casa"
exit
```

Mais exemplos



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

- **imprime todos os conteúdos**

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

- **imprime todos os índices**

ARRAYS

array[0]=”AULA”
array[1]=”DE”
array[2]=”LP”

Ou : array=(“AULA" “DE" “LP")

echo “${array[0]}”
echo “${array[1]}”

Saída: AULA
DE

echo “${array[@]}”

Saída: AULA DE LP

echo "${#array [@]}”

Saída: 3

ARRAYS ASSOCIATIVOS

```
$ 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

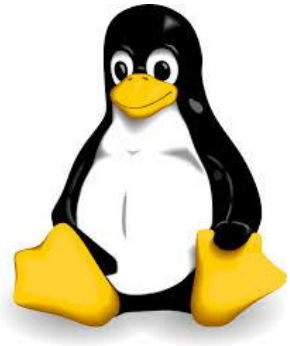
3 – VARIÁVEIS E CONSTANTES

Variáveis de ambiente

EM SHELL SCRIPT HÁ DOIS TIPOS DE VARIÁVIES DE AMBIENTE:

- **Variáveis locais**
- **Variáveis Globais**

Variáveis globais



Elas são visíveis para todas as sessões shell, e para qualquer processo filho que o shell criar. Variáveis locais estão disponíveis apenas para o shell que as criou. Isso torna variáveis globais úteis para processos filhos que requerem informações de processos pai.
Além disso, todas as variáveis são globais por padrão. Diferencia-se utilizando os comandos "*local*" ou "*declare*".

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

Variáveis locais, como o próprio nome diz, podem ser vistas apenas no processo que as criou.
Em shell, essas variáveis são declaradas utilizando o comando *local* seguido do nome da variável. Elas são atribuídas dentro de uma função e não podem ser utilizadas em outro local.

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.

Exemplo

O exemplo abaixo é um trecho do código do trabalho prático.

```
ordenar_fornecedores(){
local i=0
while (( $i < ${#vetor_fornecedores[*]} ))
        do
                vetor_fornecedores_ord+=
                ( ${vetor_fornecedores[i+1]} )
                let i=$i+7
        done
...}
```

No linux, não há um comando que permita visualizar somente variáveis locais, mas com o comando *set* podemos ver as globais e locais.

Constantes

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

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

Observe o exemplo acima. Nele há um erro, pois como const é tipo readonly, não podemos fazer var receber seu valor desse modo.

Declare

Existe uma maneira de se especificar o tipo de uma variável, utilizando-se o comando “declare”. Exemplo:

declare -i var=5
var=“string”

(var fica com o valor 0)

Opções do comando declare:

-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)

MAIS SOBRE SINTAXE DE VARIÁVEIS

Strings:

ex 1:
var = testando
echo $var

ex 2:
var = testando esse comando
echo $var

Nos exemplos acima, o ex 1 irá funcionar imprimindo ‘testando”. Já o ex 2 aparecerá a seguinte mensagem:

-bash: a: command not found

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

var=‘testando esse comando’

MAIS SOBRE SINTAXE DE VARIÁVEIS

Exemplo de interpolação:

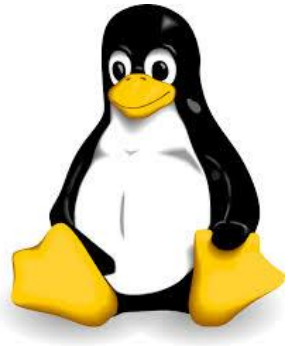
```
var1=shell
var2=script

var3=”$var1 $var2”
echo $var3
Saída: shell script

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

- Interpolação é um 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.

MAIS SOBRE SINTAXE DE VARIÁVEIS



Além disso, podemos utilizar crase (`) no script para chamar comandos do sistema. Exemplo:

echo “O usuário é: `users`”
Saída: O usuario é: a2011100796

Atenção

É EXTREMAMENTE IMPORTANTE QUE NÃO TENHA ESPAÇOS ENTRE AS VARIÁVEIS DE AMBIENTE, O SINAL DE IGUAL E O VALOR.
EXEMPLO:

$ test2 = test
-bash: test2: command not found

Variáveis e Gerenciamento de Memória

As variáveis são criadas como variáveis ambientes, deixando o sistema operacional responsável pela gerência de 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.

Se não existisse uma variável de ambiente chamada PATH você teria que digitar todo o caminho do comando para listar por exemplo:

$ /bin/ls

Variáveis de ambiente default

$TERM
Define o terminal padrão.

$ echo $TERM
xterm

$HOME
Indica o diretório pessoal do usuário em questão.

$ echo $HOME
/home/patrick

Essa variável é muito usada em scripts que necessitam saber qual o diretório pessoal do usuário, ou seja, ao invés de indicar diretamente 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.

$USER
Guarda o nome do usuário no momento.

$ echo $USER
patrick

Variáveis de ambiente

$SHELL
Guarda o valor do shell padrão:

$ echo $SHELL
/bin/bash

$OSTYPE
Guarda o nome do sistema operacional.

$ echo $OSTYPE
linux-gnu

$TMOUT
Essa variável define o tempo máximo que o shell ficará inativo. Essa variável é de grande utilidade quando se pensa em segurança, pois se você sai e deixa o terminal de texto aberto, se tiver um valor com 30 setado nela, após 30 segundos de inatividade o shell se fecha.

TMOUT=30
$ export TMOUT

Variáveis de ambiente

Um comando interessante também é o printenv (o comando env também faz a mesma coisa) Esse comando exibe uma lista de todas as variáveis globais do linux.

$PS1 - Guarda o valor do prompt primário. Você pode personalizá-la.Veja meu prompt:

patrick@ubuntu:~$

Códigos para configurar o prompt:
\w Diretório corrente
\d Exibe data
\t Exibe hora
\s Exibe o shell corrente
\u Exibe o nome do usuário
\h Exibe o nome do host (máquina)

Para mudar seu prompt temporariamente faça:

PS1='\h@\u:\w$ \t '

Isso vai gerar um prompt assim: ubuntu@patrick:/tmp 12:29:36$

4 - Expressões e Comandos

Expressões relacionais e booleanas

ARG1 | ARG2 - Return ARG1 if neither argument is null or zero; otherwise, return ARG2.
ARG1 & ARG2 - Return ARG1 if neither argument is null or zero; otherwise, return 0.
ARG1 < ARG2 - Return 1 if ARG1 is less than ARG2; otherwise, return 0.
ARG1 <= - ARG2 Return 1 if ARG1 is less than or equal to ARG2; otherwise, return 0.
ARG1 = ARG2 - Return 1 if ARG1 is equal to ARG2; otherwise, return 0.
ARG1 != ARG2 - Return 1 if ARG1 is not equal to ARG2; otherwise, return 0.
ARG1 >= ARG2 - Return 1 if ARG1 is greater than or equal to ARG2; otherwise, return 0.
ARG1 > ARG2 - Return 1 if ARG1 is greater than ARG2; otherwise, return 0.

Expressões ARTIMÉTICAS

ARG1 + ARG2 - Return the arithmetic sum of ARG1 and ARG2.
ARG1 - ARG2 - Return the arithmetic difference of ARG1 and ARG2.
ARG1 * ARG2 - Return the arithmetic product of ARG1 and ARG2.
ARG1 / ARG2 - Return the arithmetic quotient of ARG1 divided by ARG2.
ARG1 % ARG2 - Return the arithmetic remainder of ARG1 divided by ARG2.
** - Exponentiation

Retirados do livro "Linux Command Line and Shell Scripting Bible, 2nd Edition

Exemplo com uso de strings

- **Concatenando strings:**

```
a="I am a string"
b="in Shell Script"
c="$a $b"
echo $c
```

Saída: I am a string in Shell Script
Transformar vetor em string: vet=(${c})
Echo ${vetor[@]} – Saída: I am a string in Shell Script
Revertendo: c={vet[#]}

Comparadores para strings

= - Igual
!= - Diferente
-n - Não nula
-z - Nula
Exemplo:

```
var1=shell
var2=var1
if var2=var1
        then
        echo $var2
fi
Saída shell
```

Expressões de agregação

EXEMPLOS

var=1
((var+=2))
let var+=1
let var=var/2
let var*=2
let var--
var+=4
echo $var
Saída: 34
*let é a mesma coisa que usar (( )).
Ex.:
$ var='5'; let var+=bar; echo "$var“ – Saída: 5
$ var='5'; bar=2; let var+=bar; echo "$var" - Saída: 7

Exemplos de operadores aritméticos

```
$ cat test7
#!/bin/bash
var1=100
var2=50
var3=45
var4=$[$var1 * ($var2 - $var3)]
echo “O resultado final é $var4”
```

Executando:

```
$ ./test7
O resultado final é 500
```

USO DE PARÊNTESES

No exemplo anterior é importante notar o uso de Brackets (Parênteses, colchetes, ...). Como pôde ser notado, em shell script há o uso do dólar para indicar o fim de uma operação matemática, na forma *($[operation]).*

Outro exemplo disso é:

$ var1=$[1 + 5]

$ echo $var1

Output: 6

$ var2 = $[$var1 * 2]

$ echo $var2

Output: 12

Expressões binárias

<< - Deslocamento à esquerda
>> - Deslocamento à direita
& - E de bit (AND)
| - OU de bit (OR)
^ - OU exclusivo (XOR)
~ - Negação de bit
! – Não de bit (NOT)
Atribuição:
<<= - Deslocamento à esquerda
>>= - Deslocamento à direita
&= - E de bit (AND)
|= - OU de bit (OR)
^= - OU exclusivo (XOR)

Expressões condicionais

If ***command***
then
commands
Fi

Exemplo:

```
$ cat test1
#!/bin/bash
# testing the if statement
if date
then
echo “it worked”
fi
$ ./test1
Sat Nov 22 14:09:24 EDT 2014
it worked
```

USO DO CASE

Definição

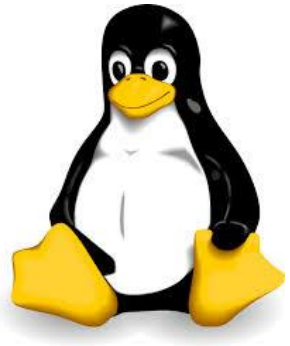
```
case variable in
pattern1 | pattern2)
commands1;;
pattern3)
commands2;;
*) default commands;;
esac
```

```
case $USER in
rich | barbara)
        echo “Welcome, $USER”
        echo “Please enjoy your visit”;;
testing)
        echo “Special testing account”;;
jessica)
        echo “Do not forget to log off when you're
done”;; *)
        echo “Sorry, you are not allowed here”;;
esac
```

- **Saída:**

```
Welcome, rich
Please enjoy your visit
```

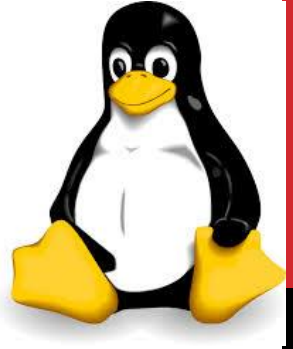
COMPARAÇÕES NUMÉRICAS



n1 -eq n2 - Check if n1 is equal to n2.
n1 -ge n2 - Check if n1 is greater than or equal to n2.
n1 -gt n2 - Check if n1 is greater than n2.
n1 -le n2 - Check if n1 is less than or equal to n2.
n1 -lt n2 - Check if n1 is less than n2.
n1 -ne n2 - Check if n1 is not equal to n2.

Retirados do livro "Linux Command Line and Shell Scripting Bible, 2nd Edition

Mais exemplos



```
$ cat test5
#!/bin/bash
# using numeric test comparisons
val1=10
val2=11
if [ $val1 -gt 5 ]
then
echo “The test value $val1 is greater than 5”
fi
www.it-ebooks.info
if [ $val1 -eq $val2 ]
then
echo “The values are equal”
else
echo “The values are different”
fi
```

Saída: The test value 10 is greater than 5
The values are different

```
$ cat test8
#!/bin/bash
# testing string equality
testuser=baduser
if [ $USER != $testuser ]
then
echo “This is not $testuser”
else
echo “Welcome $testuser”
fi
```

Saída: This is not baduser

Comandos de repetição

Comando for

for var in list do
commands
done
Exemplo:
for i in 1 2 3 4 5 do
echo $i
done
Saída:
1 2 3 4 5

Comando while

While test command
do
Other commands
done
Exemplo:
var1=10
while [$var1 -gt 0]
do
echo $var1
var1=$[$var1 - 1]
Done
Saída: 1 2 3 4 5 6 7 8 9 10

Leitura/escrita de arquivos

LEITURA

```
for line in $(cat file.txt);
do
echo "$line" ;
done
Saída:
Linha1
Linha2
Linha3
```

Escrita

```
echo “shell” > arquivo
Ou
echo “shell” >> arquivo
```

5 - Modularização e polimorfismo

Modularização e polimorfismo

Modularização é feita através de Funções

```
#!/bin/bash
#Criando uma função

minhaFuncao(){
        echo “Sou uma funcao”
}

#chamando a funcao

minhaFuncao
```

Modularização e polimorfismo

Utilizamos $ seguido da posicao do parametro para capturar o seu valor:

```
#!/bin/bash
#Criando uma função

minhaFuncao(){
        echo “Sou uma funcao com o parametro $1”
}

#chamando a funcao

minhaFuncao ”$parametro1”
```

Modularização e polimorfismo

Observamos que nas funções, nós não declaramos os tipos e nem mesmo quantos argumentos a mesma irá receber. Porém é muito perigoso, pois cabe ao programador o tratamento de todas as possiveis combinações de argumentos da função.

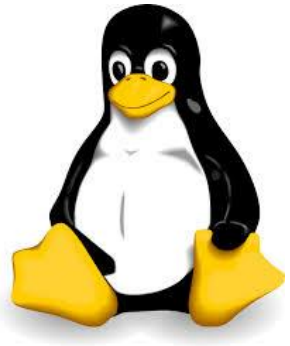
6 - Exceções

NAO EXISTE!

7 - Concorrencia

NAO EXISTE!

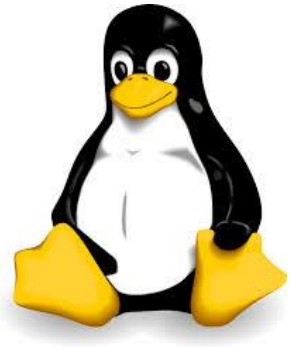
8 – Avaliação da linguagem



•Baixa Legibilidade e Redigibilidade.
•Dificil de aprender.
•Baixa Confiabilidade(tipagem fraca)
•Baixa Eficiência.
•Alta Reusabilidade.
•Prática para rotinas e sub-rotinas de sistemas.

Concluimos que 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 executaveis. Porém, para projetos maiores, não é recomendada, pois é de dificil escrita, e por ter várias formas de se fazer a mesma coisa, torna-se muito dificil a compreenção do código por outro programador.

Referência



1. http://www.softpanorama.org/People/Shell_giants/introduction.shtml acessado em 19/11/2014