# Shell script

اولین دستور در فایل های shell script‌ باید آدرس مفسری را مشخص کرده باشیم (بعد از دو کارکترها (شی بنگ) #! می آید) که قرار است دستورات بعدی در فایل اسکریپت را اجرا کنند.

#!/bin/bash

#!/usr/bin/python3.6

برای اینکه یک فایل امکان اجرا داشته باشد باید این دسترسی به فایل داده شده باشد.

#chmod +x first.sh

بعد از اعمال این دسترسی برای اجرای این فایل کافی است که مسیر فایل را در bash بنویسیم که این عمل به سه روش انجام می شود

1. به روش روبرو که در این مدل یک fork از شل فعلی ساخته می شود و این اسکریپت را اجرا می کند

# ./first.sh

1. به روش روبرو که در این مدل نیز یک fork از شل فعلی ساخته می شود و این اسکریپت را اجرا می کند

# bash first.sh

1. به روش روبرو که در این مدل خود شل فعلی است که اجرای اسکریپت را انجام می دهد

# . first.sh

برای گرفتن ورودی می توان از دستور read استفاده کنیم

read arg1

بعد از این دستور شل منتظر وارد کردن ورودی از کاربر می ماند. در ضمن می توانیم یک پرامپت به کاربر نشان دهیم و از او درخواست ورودی کنیم به وسیله -p

read -p "enter a number : " arg1

برای اینکه داده وارد شده از کاربر مانند پسورد دیده نشود باید از گزینه -s استفاده کنیم

read -s -p "enter a number : " arg1

اگر بخواهیم در یک مدت زمان منتظر بماند تا عدد را واردد کند در صورتی که وارد نکرد به دستور بعد برود می توانیم برای آن زمان مشخص کنیم با استفاده از گزینه -t (برای ۵ ثانیه منتظر می ماند)

read -s -t 5 -p "enter a number : " arg1

حالا برای نشان دادن این متغیر می توان از echo استفاده کرد.

echo $arg1

نکته: تفاوت $var و ${var} در آن است که در استفاده از {} ما متغیر را کاملا از باقی متن جدا می کنیم. برای مثال تفاوت $vartext و ${var}text مشخص است که در اولی متغیر vartext است و در دومی var

برای دیباگ کردن محتویات فایل اسکریت می توان از دستور set -V و بعد دستورات و بعد برای بستن آن از set +V استفاده کرد.

set -v -x

date

#ls

ceph -s

set +v +x

به این ترتیب در خروجی bash اعلام می کند که چه دستوری را خوانده (و به دلیل -x با قرار دادن ++ میگوید کدام دستورات را اجرا کرده است) و نتیجه اجرای آن را نیز نشان می دهد.

## کنترل جریان در اسکریپت

ابتدا برای آنکه بتوانیم ورودی به شل اسکریپت بدهیم باید بعد از درخواست اجرای اسکریپت آرگومان ها را به ترتیب اعلام کنیم. مانند روبرو

#./first.sh arg1 arg2

حالا برای اینکه بتوانیم داخل اسکریپت از این آرگومانها استفاده کنیم $1 آرگومان اول و $2 نیز آرگومان دوم و به همین ترتیب بعدی ها را می توانیم فراخوانی کنیم.

به وسیله $@ و $\* می توانیم تمام آرگومان ها را داشته باشیم و در $# تعداد آرگومانها را خواهیم داشت.

echo $\* یا echo $@ : تمام آرگومان های ورودی را چاپ می کند

echo $# تعداد آرگومان های ورودی را چاپ می کند

## ساختار if

ساختار if در شل اسکریپت به این فرم است که با if شروع می شود و با fi پایان می یابد و شرط باید داخل [ ] و حتما باید بین [ و متن شرط و ] اسپیس زده شود. برای انجام دستوری در لینوکس نیز از then استفاده می شود. همچنین میتوانیم مانند سایر زبانهای برنامه نویسی از else استفاده کنیم. در ضمن به جای else if از elif استفاده می شود.

مثالهایی از استفاده از if

num1=5

num2=10

num3=100

num4=20

### if ساده

if [ $num1 -lt $num2 ]

then

echo "Num1 < Num2"

fi

#####################################3

### If,then,else

if [ $num1 -ge $num2 ]

then

echo "Num1 >= Num2"

else

echo "Num1 < Num2"

fi

#####################################3

### If تو در تو

if [ $num1 -lt $num2 ]

then

if [ $num3 -lt $num4 ]

then

echo "num1 < num2 & num3 >= num4"

else

echo "Second Condition Not Correct"

fi

else

echo "Condition Not Correct"

fi

echo "You are Here\*\*\*\*\*\*\*\*\*\*88"

### if,elif

if [ 5 -gt 10 ]

then

echo "First IF"

elif [ 10 -eq 11 ]

then

echo "Second IF"

elif [ 20 -gt 30 ]

then

echo "Third IF"

else

echo "FINISHED..."

fi

## ساختار case,esac

این ساختار شبیه switch,case در زبانهای برنامه نویسی است که یک سری الزامات دارد

1. case 'statement' in با این دستور شروع می شود
2. بعد از هر شرطر باید یک ( قرار بگیرد
3. در خط پایانی هر قسمت ;; قرار بگیرد
4. به عنوان default باید از \*) استفاده شود
5. در این ساختارهای [0-9] به معنای تمام اعداد و [a-z] به معنای تمام حروف از کوچک و بزرگ است

مثالی از case,esac در زیر آمده است

input=$1

case $input in

linux)

echo "Linux Typed"

echo "Finished..."

mkdir /home/faradars/Documents/linux

;;

gnu)

echo "Gnu Typed"

;;

gnu/linux)

echo "Gnu/linux Typed"

;;

opensource)

echo "openSource Typed"

;;

[0-9][0-9] )

echo "You Enter DIGIT"

;;

[a-z] | [A-Z] )

echo "You Enter CHAR"

;;

\*)

echo "Undefined Input"

exit

;;

esac

## شروط عددی

انواع شرط می تواند مانند زیر باشد

-lt : یعنی lower-than کوچکتر از

-gt : یعنی greater-than بزرگتر از

-le : یعنی lower-equal کوچکتر مساوی

-ge : یعنی greater-equal بزرگتر مساوی

-eq : یعنی equal برابر

-ne : یعنی not-equal مساوی نیست با

-o : یعنی or منطقی دو عبارت

-a : یعنی and منطقی دو عبارت

! : نیز برای NOT کردن عبارت استفاده میشود باید دقت داشت که باید یک فاصله بین این علامت و عبارت باشد

$var : در صورتی که یک این مقدار داخل [] جلوی یک شرط باشد در صورت وجود متغیر مقدار if درست است و در صورت عدم وجود این شرط غلط می شود



## شرطهای رشته ای

فرق اول و مهم شرطهای رشته ای با مدل عددی این که شرایط شرط باید بین دو کروشه قرار بگیرد [[ ]] تا بررسی شود. برای بررسی طول متغیر از دو آپشن-n و -z استفاده می کنیم، -z بررسی می کند که آیا طول متغیر برابر صفر است در این صورت true‌ می شود و -n بررسی میکند که آیا طول متغیر متفاوت از صفر است (بزرگتر از صفر) است در این صورتtrue بر میگرداند.

در ضمن برای بررسی regex می باید از =~ استفاده شود

## 

str1="F"

str3="Faradars"

str4="F"

str6="GB"

str7="GA"

str8=

###############################3#

if [[ $str1 > "G" ]]

then

echo "1.Correct"

else

echo "1.Not Correct"

fi

###############################3#

if [[ $str6 > $str7 ]]

then

echo "6.Correct"

else

echo "6.Not Correct"

fi

#####################################

if [[ -z $str3 ]]

then

echo "3.Correct"

else

echo "3.Not Correct"

fi

#####################################

str5="Faradars"

#if [[ $str5 =~ [1-5][A-D] ]]

#if [[ $str5 =~ ^Farz ]]

if [[ $str5 =~ ars$ ]]

then

echo "4.Correct"

else

echo "4.Not Correct"

fi

در ضمن می شود در قسمت های بررسی چک از کلاسها استفاده کرد دقیقا مانند آنچه در دستور tr استفاده می شود

1. [[:upper:]] : که فقط حرف بزرگ را تایید می کند
2. [[:lower:]] : که فقط حرف کوچک را تایید می کند
3. [[:digit:]] : که فقط عدد را تایید می کند
4. [[:space:]] : که فقط فاصله را تایید می کند
5. [[:punct:]] : که فقط علائم نوشتاری مانند کاما و غیره را تایید می کند
6. [[:cntrl:]] : که فقط کاراکتر های کنترلی را تایید می کند
7. [[:alpha:]] : که فقط حروف را تایید می کند
8. [[:alnum:]] : که فقط حروف و اعداد را تایید می کند
9. [[:xdigit:]] : که فقط کاراکتر های هگزادسیمال را تایید می کند

## بررسی شروط مربوط به فایل

شروط به شرح زیر هستند:

-a و -e: بررسی می کند که آیا فایلی که به آن داده ایم وجود دارد یا نه

-b : بررسی می کند که آیا فایلی که داده شده از نوع بلاک است

-d : بررسی می کند که آیا دایرکتوری وجود دارد یا نه

-f : بررسی می کند که آیا فایلی از نوع معمولی است یا نه

-h : بررسی می کند که آیا فایلی از نوع سیمبولیک لینک است یا نه

-r : بررسی می کند که آیا فایل دسترسی خواندن دارد یا نه

-w : بررسی می کند که آیا فایل دسترسی نوشتن دارد یا نه

-x : بررسی می کند که آیا فایل دسترسی اجرا دارد یا نه

-s : بررسی می کند که آیا حجم فایل بیشتر از صفر است یا نه

# تعریف متغیر جدید

برای تعریف یک متغیر جدید در شل کافی است نام آن را بنویسیم و مقداری به آن تخصیص بدهیم

#var1="ali" : تعریف متغیر

#echo $var1 : مقدار ذخیره شده در متغیر را نمایش می دهد

### دستور let

**همچنین برای تخصیص مقدار جدید به متغیر از let استفاده می شود.**

**let "index+=1"**

از این دستور برای assignment های ریاضی استفاده می شود.   
let  [expresion]  
**۱-**  
 **Basic arithmetic operators :** The addition(+), subtraction(-), multiplication(\*), division(/) and modulus(%) operators can be used in the expression with the **let** command.

let "myvar =2" "myvar1=1" "myvar2=myvar1+myvar"; echo $myvar2

let "myvar =2" "myvar1=1" "myvar2=myvar1-myvar"; echo $myvar2

let "myvar =2" "myvar1=1" "myvar2=myvar1\*myvar"; echo $myvar2

let "myvar =2" "myvar1=1" "myvar2=myvar1/myvar"; echo $myvar2

let "myvar =2" "myvar1=1" "myvar2=myvar1%myvar"; echo $myvar2

۲-   
**Post-increment(var++) / Post-decrement(var**- -**) operator :** This operator is used to interpret the integer value then increase/decrease the integer variable by 1.

let "myvar=2" "myvar2=myvar++" ; echo $myvar $myvar2

**Pre-increment(++var) / Pre-decrement(**- -**var) operator :** This operator increases/decreases the integer value by 1 and then interpret the integer variable.

let "myvar=10" "myvar2=--myvar"; echo $myvar $myvar2

۳-   
**Unary plus(+exp) / Unary minus(-exp)**  
let  "var1=10" "var2= -var1"; echo $var2  
۴-   
**Exponent(\*\*) operator :** This operator is used to raise one quantity to the power of another.

let "myvar= 5 \*\* 2" ; echo $myvar

5-   
**biwise opertions  
Bit-wise negation(~)**

let "myvar=0" "myvar= ~myvar"; echo $myvar

**Bitwise shift left / Bitwise shift right**

let "myvar = 5 << 2"; echo $myvar

**Bitwise AND operator**

let "myvar=5" "myvar2=4" "myvar3 = myvar & myvar2" ; echo $myvar3

**Bitwise OR operator**

let "myvar=7" "myvar2=4" "myvar3= myvar | myvar2" ; echo $myvar3

**Bitwise XOR operator**

let "myvar=7" "myvar2=4" "myvar3= myvar ^ myvar2" ; echo $myvar3

## دستور expr

از این دستور برای انجام عملیات ریاضی در خط استفاده می شود مثلا

#expr 1 + 2 # add

#expr 1 - 2 # subtract

#expr 1 \\* 2 # multiply

#expr 1 / 2 # division

#expr 1 % 2 # module

عملیات غیر ریاضی با دستور expr

#expr length "hello how are you? # length of a string

#expr index "hello how are you" y # find index of first "y" in text "hello how are you"

#expr substr "hello how are you" 4 10 # substring form char 4 to char 10

عملیات مقایسه ای

#expr $var1 = $var2 # check if var1 is = var2

#expr $var1 \> $var2 # check if var1 is > var2

#expr $var1 \< $var2 # check if var1 is < var2

#expr $var1 != $var2 # check if var1 is != var2

## ایجاد متغیر ها

برای ایجاد متغیر با مقدار ثابت می توان به روش زیر در داخل فایل شل اسکریپت عمل کرد

Vim file.sh

readonly var1=15

var1=16 : این خط در اجرا خطا ایجاد می کند

برای تعریف متغیر محلی داخل فایل شل اسکریپت به این ترتیب عمل می شود

Vim file.sh

Func1(){

local var1="ALI"

echo $var1

}

Func1

echo var1 : این دستور خروجی ندارد زیرا دیگر مقدار در متغیر خارج از تابع وجود ندارد

برای عمومی کردن متغیر از کلمه کلیدی export استفاده می شود.

#export var1="ali"

البته وقتی به این ترتیب یک متغیر را عمومی می کنیم این متغیر فقط در همین شل فعلی و فرزندان ساخته شده از این شل شناخته می شود

# ./first.sh : که یک شل فرزند می سازد و اسکریپت را اجرا می کند متغیر را میبیند

# . first.sh : خود شل کنونی که می خواهد اجرا شود نیز این متغیر را می بیند

اما اگر این شل را بندیم و دوباره شل را باز کنیم دیگر این متغیر را نمی شناسد (بنابراین سطح این متغیر برابر با شل فعلی است) برای آنکه این متغیر حتی در صورت بسته شدن شل کماکان در شل های بعدی نیز وجود داشته باشد باید در فایل .bashrc که در دایرکتوری خانه هر کاربر قرار دارد این متغیر را تعریف کنیم.

(home)# vim .bashrc

export var="ali"

به این ترتیب در تمام شل های بعدی نیز این متغیر در دسترس است

حالا برای اینکه این متغیر دیگر مقدار نداشته باشد از دستور unset استفاده می کنیم

#unset var1

البته این متغیر فقط در شل کنونی unset شده است ولی اگر متغیر در فایل .bashrc تعریف شده باشد بعد از باز کردن دوباره شل این متغیر در دسترس است

## کوتیشن ها

ما به طور کلی سه دسته عمومی داریم

1. " "
2. ' '
3. ` `

که مقادیری که داخل این کوتیشن ها می نویسیم مقداری مقاومت جهت تفسیر توسط شل از خود نشان می دهند.

بیشترین مقاومت را ' ' از خود نشان می دهد یعنی حتی اگر دستورات مشخص شل را هم داخل این کوتیشن بنویسیم بدون تفسیر باقی می ماند

# var1 = 'salam $SHLVL'

# echo $var1

Salam $SHLVL : این خروجی است

اما اگر همین دستور را داخل " " می نوشتیم شل وارد کوتیشن می شد و به جای $SHLVL مقدار آن که مثلا برابر با ۱ است را نمایش می داد.

` ` برای آن استفاده می شود که دستوری داخل این کوتیشن قرار می گیرد مقدار حاصل از اجرای آن مثلا در متغیر قرار می‌گیرد نه خود مقدار

# var1= `pwd` : مقدار خروجی دستور داخل متغیر قرار می گیرد 🡺 var1=$(pwd)

این مورد دقیقا معادل آن است که دستور را در یک پرانتز قرار دهیم و قبل از آن $ بگذاریم

## اختصاصی کردن ترمینال (رنگی کردن)

اولا می توان پرامپت پشت دستورات را در شل اختصاصی کرد. حالت پیش فرض پرامپت  [\u@\h \W]\$ است.

* **\u**: Display the current username .
* **\h**: Display the hostname
* **\w**: Print the base of current working directory.
* **\$**: Display # (indicates root user) if the effective UID is 0, otherwise display a $.

برای این که این تغییرات همواره در bash دیده شود باید این متغیر PS1 که نگهدارنده نوع پرامپت است را در .bashrc تعریف کنیم

export PS1= "[\u@\h\w]\$"

البته برای آنکه نام host نوشته نشود می توان این متغیر را به اندازه زیر تغییر داد.

export PS1= "[\u@\w]\$"

برای رنگی کردن کاراکترها باید از فرمت زیر استفاده شود ( مثلا برای آنکه مقدار پرامپت رنگی نمایش داده شود باید به شکل زیر عمل کرد)

\e[x;ym $PS1 \e[m'

در این مدل قسمتی که می خواهیم رنگی شود باید با عبارت \e[code\_of\_colorm شروع شود و این رنگ تا جایی ادامه پیدا می کند که به عبارت \e[m برسد کدهای رنگی مطابق جدول زیر هستند.

|  |  |
| --- | --- |
| Color | Code |
| Black | 0;30 |
| Blue | 0;34 |
| Green | 0;32 |
| Cyan | 0;36 |
| Red | 0;31 |
| Purple | 0;35 |
| Brown | 0;33 |
| Blue | 0;34 |
| Green | 0;32 |
| Cyan | 0;36 |
| Red | 0;31 |
| Purple | 0;35 |
| Brown | 0;33 |

قسمت x;y دو قسمت دارد که قسمت y رنگ را و قسمت x به ترتیب زیر مدل نمایش را مشخص می کند

### 1. color-mode

It modifies the style of color NOT text. For example make the color bright or darker.

* 0; reset
* 1; lighter than normal
* 2; darker than normal

This mode is not supported widely. It is fully support on Gnome-Terminal.

### 2. text-mode

This mode is for modifying the style of text NOT color.

* 3; italic
* 4; underline
* 5; blinking (slow)
* 6; blinking (fast)
* 7; reverse
* 8; hide
* 9; cross-out

3- foreground

* 30; black
* 31; red
* 32; green
* 33; yellow
* 34; blue
* 35; purple
* 36; cyan
* 37; white

4- background

* 40; black
* 41; red
* 42; green
* 43; yellow
* 44; blue
* 45; purple
* 46; cyan
* 47; white

## alias راهی برای ساده سازی دستورات

برای اینکه به جای اینکه یک دستور طولانی را مدام تایپ نکنیم می توانیم از alias استفاده کنیم و این ترتیب آن دستور بلند را با یک دستور کوتاه جایگزین کنیم.

#alias ll = 'ls -l'

به این ترتیب دیگر می توان در این bash به جای ls -l فقط ll را تایپ کرد و البته برای آنکه در تمام شل ها نیز اعمال شود باید آن را در فایل .bashrc قرار داد.

## عملیات ریاضی

برای آنکه بتوانیم حاصل عملیات ریاضی را در یک متغیر ذخیره کنیم باید به روش زیر عمل کنیم

# var1 = $((1+2))

در این حالت حاصل جمع ۱+۲ در متغیر var1 قرار می گیرد.

یک پرانتز برا مواقعی استفاده می شود که میخواهیم خروجی یک دستور را در متغیر منتقل کنیم. مثلا

# var1 = $(ls)

به این ترتیب خروجی ls به متغیر var1 منتقل می شود. بنابراین در عبارات ریاضی مثل اینکه (1+2) خود یک دستور است. عملیات ریاضی در shell script به شرح زیر است.

در ضمن می توان این دستور را داخل "" نیز استفاده کرد مثلا

# echo "current directory is $(pwd)"

|  |  |
| --- | --- |
| **Description** | **Syntax** |
| Pre and post-increment. | **++x**, **x++** |
| Pre and post-decrement. | **--x**, **x--** |
| Addition, subtraction, multiplication, division. | **+**, **-**, **\***, **/** |
| Modulo (remainder) and exponentiation. | **%**, **\*\*** (or **^**) |
| Logical AND, OR, and negation. | **&&**, **||**, **!** |
| Bitwise AND, OR, XOR, and negation. | **&**, **|**, **^**, **~** |
| Right shift, left shift | **>> , <<** |
| Less than or equal to, less than, greater than, and greater than or equal to comparison operators. | **<=**, **<**, **>**, **=>** |
| Equality and inequality comparison operators. | **==**, **!=** |
| Assignment operator. Combines with other arithmetic operators. | **=** |

برای انجام عملیات پیچیده تر ریاضی از دستور bc استفاده کنیم.

bash calculator  
**bc command supports the following features:** 

* Arithmetic operators
* Increment or Decrement operators
* Assignment operators
* Comparison or Relational operators
* Logical or Boolean operators
* Math function

**Arithmetic Operators** 

Input : $ echo "12+5" | bc

Output : 17

Input : $ echo "10^2" | bc

Output : 100

**Assignment Operators**

* **var = value** : Assign the value to the variable
* **var += value** : similar to var = var + value

Input: $ echo "var=10;var" | bc

Output: 10

**Increment Operators**

Input: $ echo "var=10;++var" | bc

Output: 11

**Decrement Operators**

Input: $ echo "var=10;--var" | bc

Output: 9

**Comparison or Relational Operators**

* **expr1<expr2** : Result is 1 if expr1 is strictly less than expr2.
* **expr1<=expr2** : Result is 1 if expr1 is less than or equal to expr2.

Input: $ echo "10>5" | bc

Output: 1

**Logical or Boolean Operators**

* **expr1 && expr2** : Result is 1 if both expressions are non-zero.
* **expr1 || expr2** : Result is 1 if either expression is non-zero.
* **! expr** : Result is 1 if expr is 0.

Input: $ echo "10 && 5" | bc

Output: 1

**Mathematical Functions**   
The built-in math functions supported are :

* **s (x)**: The sine of x, x is in radians.
* **c (x)** : The cosine of x, x is in radians.
* **a (x)** : The arctangent of x, arctangent returns radians.
* **l (x)** : The natural logarithm of x.
* **e (x)** : The exponential function of raising e to the value x.
* **j (n,x)** : The bessel function of integer order n of x.
* **sqrt(x)** : Square root of the number x. If the expression is negative, a run time error is generated.

Input: $ echo "scale($pi)" | bc -l

Output: 20

Input: $ echo "s($pi/3)" | bc -l

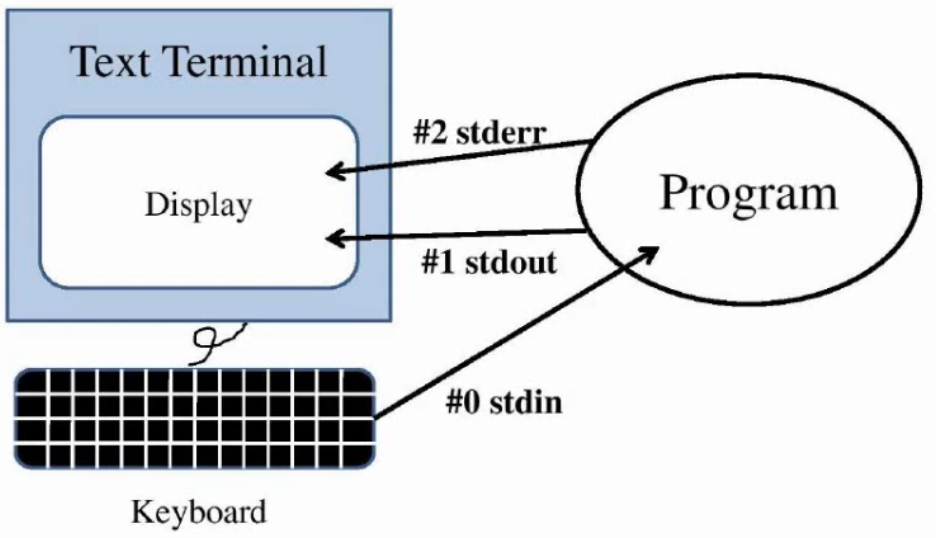
Output: .86602540378443864675

آپشن -l در واقع کتابخانه ریاضی را برای دستور bc فعال می کند

# جریان های داده ای

در لینوکس به صورت کلی سه جریان داده ای داریم

1. stdin
2. stdout
3. stderr



در حالت استاندارد که یک دستور نوشته می شود جریان ورودی کیبورد است جریان خروجی مانیتور است و جریان داده در صورتی که دستور با خطا مواجه شود جهت خروجی خطا استفاده می شود.

## علائم تغییر جریان داده

1. < : این علامت باعث می شود که خروجی به جای اینکه در خروجی استاندارد نوشته شود در فایلی یا مسیری که اعلام می کنیم می نویسد

# cat passwd > passwords.txt

در ضمن با استفاده از اعداد میتوانیم تمام جریان های داده را به مکان دقیق مد نظرمان بریزیم. مثلا با دستور زیر می توان خروجی خطا را به فایل err.txt و خروجی درست را به فایل out.txt ارسال کنیم.

# ls /boot /asd 1>out.txt 2>err.txt

نکته : خیلی وقتها می دانیم یک دستور خروجی های خطای زیادی دارد که ما نمی خواهیم آنها را مشاهده کنیم. بنابراین می توانیم خروجی آنها را به null ارسال کنیم با دستور زیر.

# دستور 2>/dev/null

1. << : این علامت نیز مانند تغییر جریان بالا کار می کند ولی با این تفاوت که محتویات جدید را به انتهای محتویات قبلی اضافه می کند. این مورد بسیار به درد مورادی که به صورت استریم مدام اضافه می شوند می خورد.
2. | : این علامت که به آن پایپ کردن هم میگویند باعث می شود که مثلا خروجی دستور قبلی به عنوان ورودی دستور بعدی مورد استفاده قرار بگیرد. یعنی در نهایت خروجی دستور آخر (سمت راست) در خروجی استاندارد نمایش داده می شود.

# cat /etc/passwd | grep ali

1. < : این علامت باعث می شود که دستور به جای اینکه ورودی خود را از ورودی استاندارد دریافت کند از فایل بعدی دریافت کند.

# grep ali < /etc/passwd

جالب است که بدانیم استفاده از این جریان داده ای بر پایپ ارجحیت دارد زیرا در این مدل خود فایل به عنوان خروجی به دستور داده می شود در صورتی که مثلا اگر با cat مقدار فایل را می خواندیم و خروجی را به grep می دادیم یکبار عملیات خواندن توسط cat انجام شده بود و یکبار عملیات grep در صورتی که در این مدل فقط یکبار فایل توسط خود grep باز می شود.

1. << : در واقع این جریان داده ای را here document می نامند به این معنی که می توان به جای اینکه یک فایل را به عنوان ورودی به یک دستور بدهیم می توانیم از این جریان داده ای استفاده کنیم و چند خط را یک جا به عنوان ورودی یک دستور ارسال کنیم. مثال

# cat > ali.txt <<EOF

>salam

>khubi

>EOF

در این مدل اولین کلمه ای که بنویسیم به عنوان شروع فایل فرض می شود و باید با همین کلمه نیز فایل را ببندیم. در ضمن در خط دوم تا آخر دیگر پرامپت عوض می شود و PS2‌ نمایش داده می شود.

1. >>> : این جریان داده ای را here string می نامند و مانند قبلی here doc است با این تفاوت که در این مدل یک متغیر به عنوان ورودی داده می شود. مثال

# read -a words <<< "salam chetori khubi"

با این دستور مقدار salam chetori khubi را به عنوان یک متغیر ورودی به دستور read دادیم و با آپشن -a باعث شدیم که این ورودی در خانه های آرایه آن قرار بگیرند. حالا می توان محتویات آن را به روش زیر مشاهده کرد

# echo ${words[1]} ## chetori

## استفاده از xargs

این دستور خود دستوری را می گیرد و ورودی های داده شده به این دستور را روی دستور دوم اعمال می کند. یعنی آرگومان های دریافتی را به آرگومان های دستور بعدی خود تبدیل می کند. به این ترتیب می توانیم بعضی از دستورات تکراری را خیلی سریعتر اجرا کنیم. مثال

# echo "one two three" | xargs mkdir

در این حالت ورودی mkdir را xargs از ورودی خود میگیرد و یکی یکی برای mkdir ارسال می شود. در واقع این دستور خیلی وقتها با دستور find‌ استفاده می شوند. به این ترتیب که به وسیله find فایل مد نظر را پیدا می کنیم و خروجی آن را به xargs می دهیم تا آن اینها را به rm برای پاک کردن بسپارد.

# find -type f -name \*.mps | xargs rm -rf

**در واقع دستوراتی که یک ورودی می گیرند ولی ما میخواهیم چند بار آنها اجرا کنیم و چند ورودی به آنها بدهیم** را می توانیم با xargs این عمل را انجام دهیم.

## حلقه ها

حلقه for خیل شبیه for در زبان c است.

For ((c=1; c>=10; c++))

do

echo "loop number is $c"

done

### حلقه بینهایت

For (( ; ; ))

### حلقه for به وسیله in

for i in {1..10} # معادل for i in 1 2 3 4 5 6 7 8 9 10

do

statements

done

### حلقه for به پرش بیش از ۱

for i in {1..10..2}

do

statements

done

که در این حالت پرش ها ۲ تایی است.

### حلقه for بر روی یک دایرکتوری

for file in /etc/\*

do

statement

done

می توان این دستور را بر روی خروجی دستور ls نیز انجام داد

for file in $(ls /tmp)

do

statements

done

### حلقه for برای پیمایش آرگومانهای ورودی

for i in "$@"

do

statements

done

## کامنت چند خطی در اسکریپت

برای آنکه یک نوشته کامنت شود از # استفاده می کنیم ولی برای اینکه چند خط کامنت شود از جریان داده >> که در واقع here document است استفاده می کنیم ولی خروجی آن را به هیچ برنامه ای نمی دهیم. مانند زیر

<< commnet

Statements

Comment

به این ترتیب تمام خطوط بین دو کلمه comment دیگر اجرا نمی شوند.

## حلقه while

نحوه کار این حلقه شبیه به زبانهای برنامه نویسی است که در آن تا وقتی شرط جلوی while درست است این حلقه به اجرای خود ادامه می دهد.

X = 1

while [ x -lt 10 ]

do

statements

let "x+=1"

done

### حلقه بینهایت

while :

do

statements

done

### حلقه بینهایت و case

while :

do

echo "Talk to me..."

read INPUT

case $INPUT in

hello)

echo "Hello $USER"

;;

salam)

echo "salam $USER"

;;

bye)

echo "Bye $USER"

break

;;

\*)

echo "Sorry $USER..."

exit 10

;;

esac

done

### کاراکترهای کنترلی داخل حلقه

با استفاده از break می توان از حلقه داخلی خارج شد و با استفاده از exit 10 می توان از کل اسکریپت با عدد ۱۰خارج شد. بنابراین اگر کاربر بعد از آنکه شل با exit 10 تمام شده است دستور $? را وارد کند عدد 10 را در خروجی می بیند. در ضمن می توان از continue نیز استفاده کرد که به این ترتیب از انجام ادامه دستورات حلقه داخلی منصرف می شود و شماره حلقه داخلی را یک واحد افزایش میدهد.

## حلقه until

این حلقه شبیه while است با این تفاوت که در while تا وقتی متن حلقه اجرا می شود که شرط جلوی while درست باشد در صورتی که متن حلقه در until تا زمانی اجرا می شود که شرط جلوی آن نا درست باشد.

i=0

until [ $i -gt 10 ]

do

echo "Wellcome $i Times."

let "i+=1"

done

که در این حلقه شرط در ابتدا غلط است بنابراین متن داخل حلقه انجام می شود تا مقدار شمارنده به عدد ۱۱ برسد که دیگر شرط حلقه درست می شود پس متن حلقه دیگر اجرا نمی شود.

## حلقه Select

در این حلقه از PS3 استفاده می شود و مقادیر مشخص شده جلوی select به صورت آپشن برای کاربر نمایش داده می شود. در حالت برای دریافت ورودی از کاربر مقداری که در PS3 مشخص کرده ایم نمایش داده می شود و در صورتی که مقداری برای PS3 مشخص نکرده باشیم مقدار PS3 آن یعنی #? نمایش داده میشود.

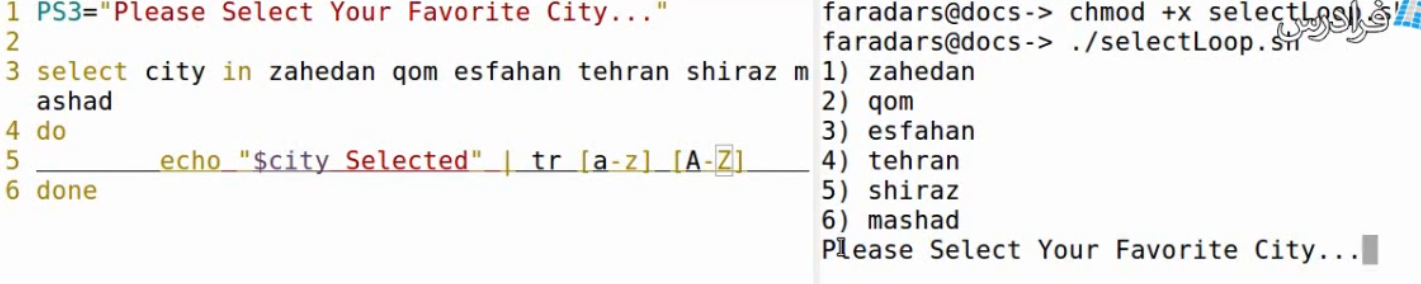
PS3="Please Select Your Favorite City..."

select city in zahedan qom esfahan tehran shiraz mashad

do

echo "$city Selected" | tr [a-z] [A-Z]

done



در این دستور با زدن عددی که به عنوان آپشن مشخص شده است دستورات داخل select با این ورودی اجرا می شود مثلا با زدن عدد ۱ zahedan انتخاب شده و حروف آن بزرگ میشود.

### حلقه select و case

PS3="Shahreto Entekhab kon: "

select CITY in zahedan qom shiraz tehran esfahan

do

case $REPLY in

1)

echo $CITY|tr [a-z] [A-Z]

;;

2)

echo $CITY|tr [a-z] [A-Z]

;;

3)

echo $CITY|tr [a-z] [A-Z]

;;

4)

echo $CITY|tr [a-z] [A-Z]

;;

5)

echo $CITY|tr [a-z] [A-Z]

;;

\*)

echo "Incorrect..."

break

;;

esac

done

همان طور که مشخص شده است می توان عددی که کاربر وارد می کند در select در متغیر $REPLY ذخیره می شود که میتوان از آن در موارد Case استفاده کرد.

## آرایه های یک بعدی

برای پر کردن یک متغیر که خود یک آرایه است باید محتویات را در میان پرانتز قرار داد.

* # names=(ali taghi naghi) #در این مدل یک آرایه ساخته می شود که سه مقدار در خانه های آن وجود دارد
* #numners=([0]=1 [1]=40 [2]=25 [3]=79) # در حالت گفته ایم که کدام خانه چه محتویاتی دارد
* P[1]="arm"; p[2]="intel" # در این مدل هریک از خانه های آرایه جداگانه به صورت یک متغیر مقداردهی شده است

برای مشاهده تمام خانه های یک آرایه می توان به دو روش زیر عمل کرد:

1. echo "names are"${names[\*]}
2. echo "names are"${names[@]}

برای آنکه خانه های آرایه به صورت دلخواه نمایش داده شود می توان به روش زیر عمل کرد:

1. echo ${numbers[@]:1} # این دستور باعث میشود که کل خانه های آرایه از خانه شماره ۱ به بعد نمایش داده شود
2. echo ${numbers[@]:1:4} # باعث می شود که خانه های آرایه از خانه شماره ۱ تا خانه شماره ۴ (با احتساب آنها نمایش دهد)
3. echo ${numbers[5]} # در اینجا فقط خانه شماره ۵ نمایش داده می شود

برای آنکه تعداد خانه ها نمایش داده شود:

# echo -e "Lenght of pTypes : "${**#**pTypes[\*]}

# تابع در شل

برای نوشتن تابع می توان از کلمه کلیدی function استفاده کرد. البته این کلمه را نیز می توان حذف کرد. ولی یک تابع با دو پرانتز () جلوی آن مشخص می شود. البته شروع و پایان تابع نیز با علامت {} است

function hello()

{

echo "Hello $1"

به وسیله $1 می توان به آرگومان اول دسترسی داشت (و به همین ترتیب به سایر ارگومان ها)

}

برای توابع می توان تعداد دلخواه آرگومان ارسال کرد (در تعریف تابع لازم نیست این آرگومان ها بیان شوند)

function showAge()

{

echo "$1 Age is $2"

echo "Number of Argument : $#"

echo "All Argument : $@"

}

function last()

{

echo "GoodBye"

}

read -p "Plz Enter Your name : " name

read -p "Plz Enter Your age : " age

hello $name # این تابع یک آرگومان دریافت کرده است

showAge $name $age # این تابع دو آرگومان دریافت کرده است

last # این تابع آرگومانی دریافت نکرده است

می توان برای توابع مقدار خروجی در نظر گرفت که این مقدار باید به وسیله return مشخص شود. اگر مشخص نکنیم که خروجی تابع در چه متغیری ریخته شود، برای دریافت خروجی تابع می توان مقدار متغیر ? که مقدار خروجی آخرین تابع اجرا شده را نگهداری می کند را echo کرد.

function plus()

{

local exprString="expr $1 + $2"

echo $exprString

return `expr $1 + $2`

}

plus 2 10

echo $?

امکان استفاده توابع تو در تو در شل وجود دارد.

function firstFunc()

{

echo $#

secondFunc $@ # در اینجا تمام آرگومان های تابع اول به تابع دوم ارسال شده است (**البته در یک آرایه** )

}

function secondFunc()

{

#echo $\*

}

firstFunc fedora ubuntu opensuse mint

firstFunc AMD INTELL ARM

می توان متن توابع را در یک فایل مجزا نوشت و با استفاده از کلمه کلیدی source متن آن توابع را در فایل دوم فراخوانی کرد. حالا می توانیم مثلا با استفاده از ساختار select از کاربر بخواهیم که یک عمل را انتخاب کند.

source ./mathFunc

read -p "Enter Number 1 :" input1

read -p "Enter Number 2 :" input2

PS3="PLZ Select ... "

select CHOICE in "addition" "subtraction" "multiplication" "division" "module"

do

case $REPLY in

1)

echo $CHOICE

addition $num1 $num2

;;

2)

echo $CHOICE

subtraction $num1 $num2

;;

3)

echo $CHOICE

multiplication $num1 $num2

;;

4)

echo $CHOICE

division $num1 $num2

;;

5)

echo $CHOICE

module $num1 $num2

;;

\*)

echo "Incorect Choice"

break;;

esac

done

البته باید این توابع ریاضی قبلا در فایلی به نام mathFunc در همین دایرکتوری قرار داده شده باشد. برای آنکه بتوان یک تابع را به پس زمینه فرستاد می توان از علامت & به عنوان آخرین آرگومان استفاده کنیم. در ضمن از آنجایی که این پراسس هایی که به پس زمینه رفته اند قابل دسترسی نیستند باید بلافاصله پس از اجرا شماره پراسس آنها را نگهداری کنیم و در زمان مناسب آن پراسس را kill کنیم.

function processAdd()

{

echo -n "the result is : "

echo `expr $1 + $2`

echo -n "..."

while true

do

echo -n "."

sleep 1

done

}

function dofor()

{

for i in {1..3}

do

# echo $i

sleep 1

done

echo " "

}

processAdd 2 3 &

processid=$! # به این ترتیب شماره پراسس آخرین پراسس اجرا شده که همان پراسس پس زمینه است را داریم

dofor

kill -9 $processid # با این دستور پراسس پس زمینه را از بین می بریم

echo "end of process"