Programación de Shell Scripts

Agenda (módulos)

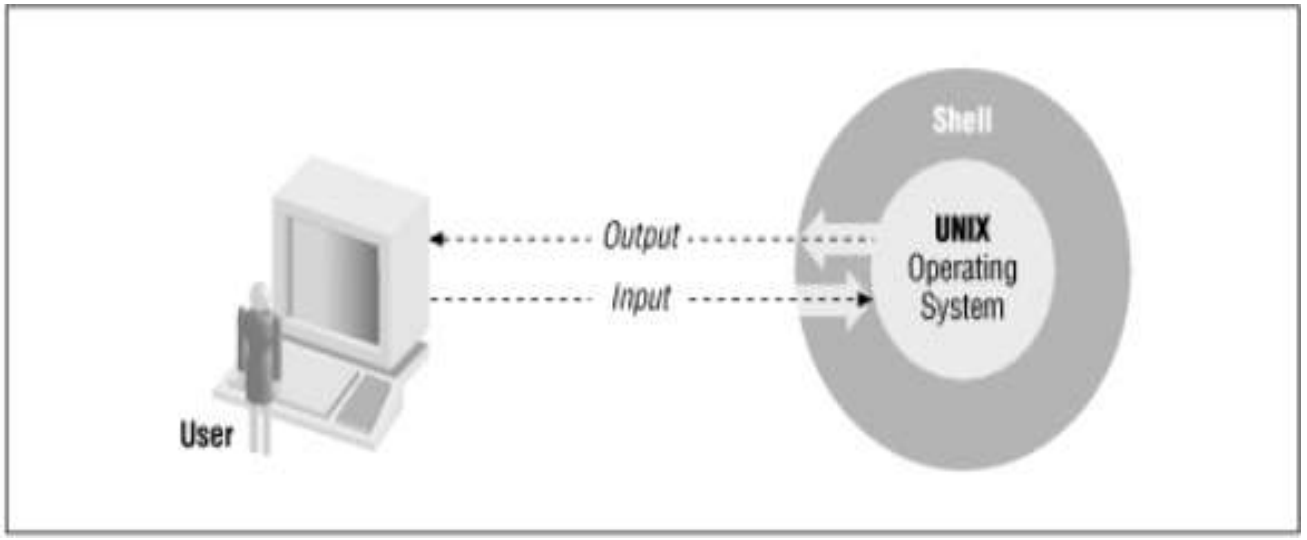
Contenido:

- **Módulo 0** - Generalidades
- **Módulo 1** - Introducción
- **Módulo 2** - Caracteres especiales y Quoting
- **Módulo 3** - Variables
- **Módulo 4** - Operadores
- **Módulo 5** - Construcciones condicionales
- **Módulo 6** - Construcciones iterativas o de repetición
- **Módulo 7** - Funciones
- **Módulo 8** - Entrada y Salida - Redirección
- **Módulo 9** - Herramientas Misceláneas
- **Módulo 10** - Filtros
- **Módulo 11** - Lenguaje **`awk`**
- **Módulo 12** - Lenguaje **`sed`**
- **Módulo 13** - Depuración de scritps. Ejercitación
- **Módulo 14** - Conclusiones

Módulo 0

Generalidades

Generalidades



- **Shell**

❶ Intérprete de comandos

❷ Lenguaje de programación

- **Entorno de trabajo**
  - Case sensitive
  - Si un programa no está en el **`PATH`**: ./nombre_programa
  - Prompts:
    - **`$`**: usuario normal
    - **`#`**: usuario administrador o superusuario (root)

- **Comandos** **`sintaxis:`** **`comando opciones argumentos`**

- Ubicaciones
  - **`/bin:`** comandos estándar para todos los usuarios (**`ls`**, **`cat`**, **`cp`**, **`mv`**)
  - **`/sbin:`** comandos estándar para root (**`shutdown`**, **`mkfs`**)
  - **`/usr/bin:`** comandos para todos los usuarios no presentes en todo sistema UNIX
  - **`/usr/sbin:`** comandos para root no presentes en todo sistema UNIX

- **Scripts**

- Lista de comandos UNIX reunidos en un archivo. Reutilización de código
- Un script es un nuevo comando

  Filosofía UNIX "crear comandos complejos a partir de comandos simples"

Historia de los shell de UNIX

- El shell es independiente del S.O. → generación de docenas de shells
- **Bourne shell** (Steven Bourne, UNIX Version 7, 1979), conocido como **sh**
- Principal alternativa a **sh** fue el **C shell** (Bill Joy, BSD, 1981), **csh**
- **Turbo C shell**, **tcsh**, superconjunto de **csh** con mejoras respecto a amigabilidad y velocidad
- **Korn shell** (David Korn, AT&T, 1986), **ksh**, comercial, {**ksh**} ≈ {**sh**} U {**csh**}
- Una alternativa sin costo es la versión de **Korn shell** conocida como **pdksh** (Public Domain Korn shell)**. pdksh** está disponible como código fuente

- **Bourne Again shell** (Brian Fox, Chet Ramey, 1988-1993), **bash**

- Creado para su uso en el proyecto **GNU** (Richard Stallman, FSF), no comercial
- Se convirtó rápidamente en el derivado de Bourne Shell más popular
- Shell estándar utilizado ampliamente en los sistemas UNIX e incluido en Linux
- {**bash**} ≈ {**csh**} U {**ksh**}
- Intuitivo y flexible

- **Z shell**, **zsh**, (Paul Falstad, Princeton, aprox. 1990) posee similitudes con **ksh**

{**zsh**} ≈ {**bash**} U {**ksh**} U {**tcsh**}

Módulo 1

Introducción

Características de bash

{Bourne Shell} < {Bourne shell Again} ≈ {C Shell} U {Korn Shell}

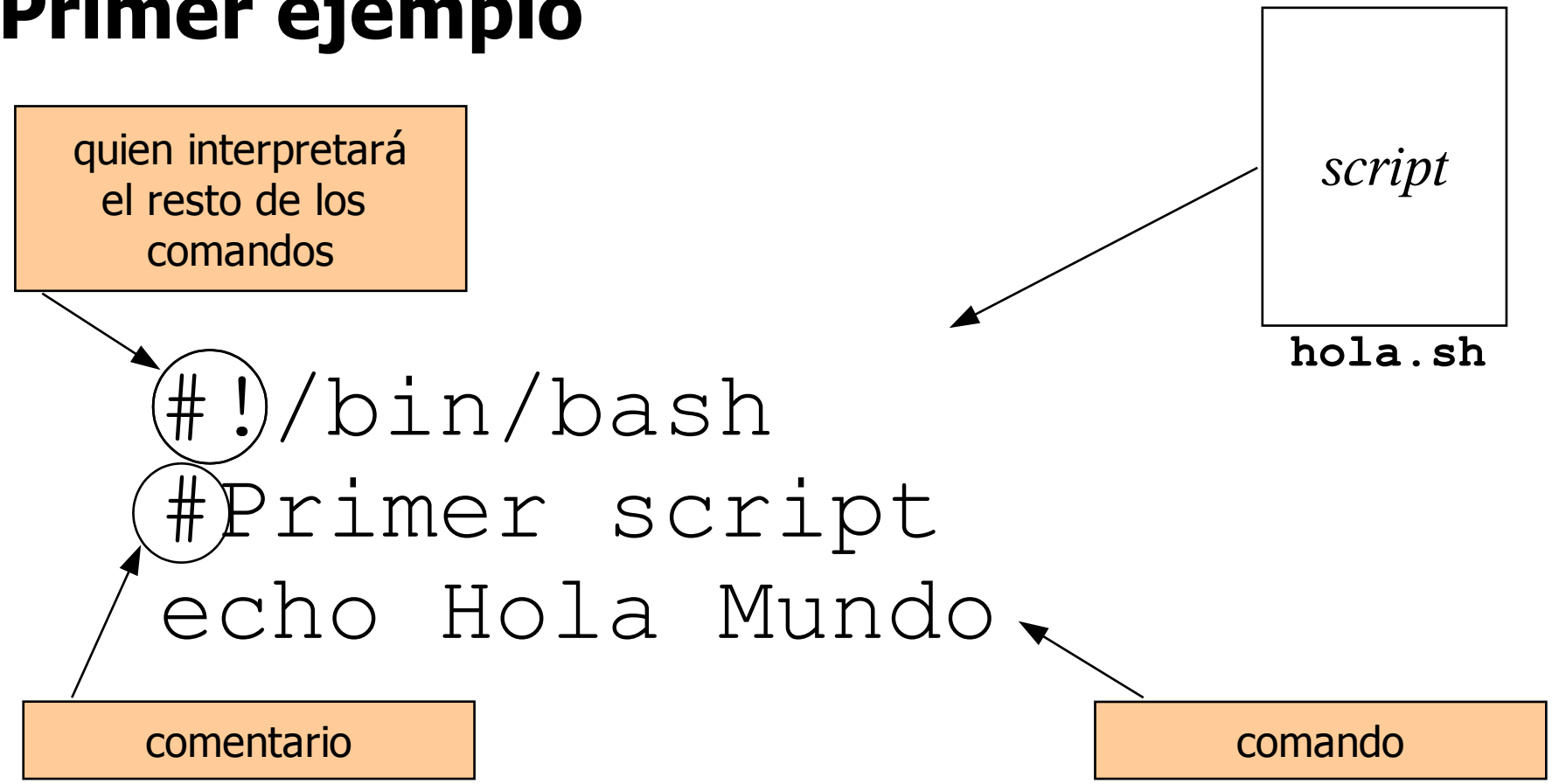
- **Características propias de C-shell incorporadas:**

- Manipulación de directorios.
- Control de trabajos.
- Expansión de llaves, para la generación de cadenas arbitrarias.
- Carácter tilde (~), manera de referenciar al directorio home.
- Alias, que permiten referenciar más convenientemente comandos y sus opciones.
- Histórico de comandos, que posibilita reutilizar comandos previamente tipeados.

- **Características propias:**

- Edición de línea de comandos, permite usar comandos al estilo vi o emacs.
- Configuración de teclas (key bindings) permiten establecer secuencias de teclas de edición personalizadas .
- Características de programación integrada: la funcionalidad de comandos UNIX (`test`, `expr`, `getopt`, `echo`) se integraron en el shell, permitiendo que tareas comunes de programación sean realizadas más clara y eficientemente.
- Estructuras de control, especialmente el select para la generación sencilla de menús.
- Opciones y variables nuevas permiten personilizar más el entorno.
- Arrays uni-dimensionales que permiten fácil acceso a lista de datos.

Primer ejemplo



Ejecución:

```
$./hola.sh
Hola Mundo
```

¿Cómo ejecuta un comando en shell?

❶ Lee la entrada desde un archivo, como un argumento o desde la terminal

❷ Divide la entrada en tokens de acuerdo a las reglas de quoting. Se hacen expansiones de metacaracteres y alias

❸ Se analizan los tokens y se dividen en comandos simples y compuestos

❹ Se hacen expansiones separando los tokens expandidos en listas de nombres de archivo y comandos con sus argumentos

❺ Se realizan redireccionamientos, eliminando operadores y operandos de redirección

❻ Se ejecuta el comando

❼ Opcionalmente se espera la finalización del mismo para recoger su exit status

Archivos de configuración de bash (1/5)

- **Existen, eventualmente, tres archivos en el directorio home**

  `.bash_profile`
  `.bashrc`
  `.bash_logout`

- **Se utilizan para definir variables y/o ejecutar comandos al ingresar al sistema, cuando se invoca un nuevo shell o al cerrar la sesión**

- **Pueden existir o no dependiendo de como se creo la cuenta de usuario.**
  Si no existen, el usuario utilizará sólo el archivo de sistema `/etc/profile`, o puede optar por editarlos él mismo

- **El más importante es `.bash_profile`, el cual es leido por el shell para ejecutar los comandos que contiene cada vez que se ingresa al sistema**

Archivos de configuración de bash (2/5)

- **Algunas líneas de `.bash_profile`**

```
PATH=/sbin:/usr/sbin:/bin:/usr/bin:/usr/local/bin
SHELL=/bin/bash
MANPATH=/usr/man:/usr/X11/man
EDITOR=/usr/bin/vi
PS1='\h:\w\$ '
PS2='> '
set -o ignoreeof
export EDITOR
```

- **Si se agregan nuevas líneas no serán consideradas hasta que el archivo `.bash_profile` sea leído/ejecutado nuevamente, reingresando al sistema, por ejemplo**

- **Comando `source`: Ejecuta los comandos en el archivo especificado**

```
$ source .bash_profile
```

- **Alternativa para `source` comando, utilizar el comando punto (.)**

```
$ ./bash_profile
```

Archivos de configuración de bash (3/5)

- **bash admite dos sinónimos para `.bash_profile`:**
  - **`.bash_login`, derivado del archivo `.login` de C shell**
  - **`.profile`, derivado del archivo `.profile` del Bourne shell y el Korn shell**

- **Se lee sólo uno de estos al ingresar al sistema**

- **<u>Orden:</u> `.bash_profile` → `.bash_login` → `.profile`**

- **`.bashrc` puede o no existir**

- **Al iniciar un nuevo shell (un subshell) se leerán los comandos de `.bashrc`**

- ***Separación de comandos de inicio y de personalización de shell***

- **Si `.bashrc` no existe entonces no se ejecutarán cuando se inicia un subshell**

Archivos de configuración de bash (4/5)

```
# If running interactively, then:
if [ "$PS1" ]; then
    # don't put duplicate lines in the history.
    # export HISTCONTROL=ignoredups

    # enable color support of ls and also add handy aliases
    eval `dircolors -b`
    alias ls='ls --color=auto

    # some more ls aliases
    #alias ll='ls -l'
    #alias la='ls -A'
    #alias l='ls -CF'

    # set a fancy prompt
    PS1='\u@\h:\w\$ '

    # If this is an xterm set the title to user@host:dir
    #case $TERM in
    #xterm*)
    # PROMPT_COMMAND='echo -ne "\033]0;${USER}@${HOSTNAME}: ${PWD}\007"
    #    ;;
    #*)
    #    ;;
    #esac
fi
```

Archivos de configuración de bash (5/5)

- **`.bash_logout` es leído y ejecutado cada vez que se sale del *login shell***

- **Brinda la capacidad de ejecutar comandos, como eliminar archivos temporarios**

- **Generalmente hay que definirlo mediante edición manual**

- **Si no existe, no se ejecutarán comandos extra a la salida**

Alias

- **A veces la sintaxis de los comandos es difícil de recordar, especialmente si se utilizan con varias opciones y argumentos**
- **Alias = sinónomo enriquecido**
- **Los alias pueden definirse en línea de comandos, en el `.bash_profile`, o en `.bashrc`, mediante:**

  `alias [name=command]` **// sin espacios entre el signo =**

- **Ejemplos:**

```
$ alias lf='ls -F'
$ alias revsort='ls [0-9]* | sort -v'
```

- **Notas:**

- Se permite definir un alias de un alias

- No están permitidos los alias recursivos `$ alias ls='ls -la'`

- Los alias sólo pueden usarse al principio de un comando (existen excepciones)

```
$ alias pkgs=/var/sadm/pkg
$ alias cd='cd '
```

- **Implicancias**

- Brevedad	(`ls=ls -lha`)	- Costumbre	(`dir=ls`)
- Protección	(`rm=rm -i`)	- Personalización	(`moer=more`)

Opciones

- **Los alias permiten definir nombres convenientes para los comandos pero no cambian realmente el comportamiento del shell**
- **Una opción de shell se establece como activa/inactiva (`on`/`off`) y cambia efectivamente el comportamiento del shell**
- **Sintaxis básica (contraintuitiva)**

```
set +o opcion     off
set -o opcion     on
```

- **Para visualizar el estado de las opciones `set -o`**
- **La mayoría de los nombres de opciones tienen asociado una letra para abreviarlas, `set -o noglob` → `set -f`**
- **<u>Ejemplos:</u> (en `on`)**

`ignoreeof`	Deshabilita **`Ctrl-d`** para salir de la sesión; debe usarse **`exit`**
`noclobber`	Deshabilita la redirección de salida (>) sobre archivos existentes
`noglob`	Deshabilita la expansión de metacaracteres como **`*`** y **`?`**
`notify`	Reporta el estado de terminación de los **`jobs`** de inmediato
`nounset`	Indica un error cuando se intenta utilizar variables no definidas
`vi`	Entra al modo edición de vi directamente

Módulo 2

Caracteres especiales y Quoting

Caracteres especiales

- **Conocidos también como metacaractes**
- **Los metacaracteres poseen significado especial para el shell**
- **Existen diversas categorías de acuerdo a la funcionalidad con que estén relacionados**
- Ejemplos:

```
$ cd ~/libros
$ rm *.bck
$ find / -name a* &          # Búsqueda prolongada
$ echo El dijo \"Hola\"
$ echo Fecha y hora actual: `date`
$ echo Hay `wc -l /etc/passwd | awk '{print $1}'` usuarios
```

Caractes especiales

Caracter	Significado
~	Directorio home
`	Sustitución de comando
#	Comentario
$	Valor de variable
&	Trabajo en background
*	Estrella de Kleene (expresiones regulares)
(	Inicio de subshell
)	Fin de subshell
\	Carácter de Escape
|	Pipe
[	Inicio de conjunto de caracteres (expresiones regulares)
]	Fin de conjunto de caracteres (expresiones regulares)
{	Inicio de bloque de comandos
}	Fin de bloque de comandos
;	Secuencializar comandos
'	Comilla simple (Strong quote)
"	Comillas dobles (Weak quote)
<	Redirección entrada
>	Redirección salida
/	Separador de directorios en pathname
?	Reemplazo de un carácter (expresiones regulares)
!	Negación de pipeline

Archivos, comodines y pathname expansion

- **Los archivos ocultos comienzan con punto (.), utilizar `ls -a` (*a*:all)**
- **Comodines**

`?`	un carácter cualquiera
`*`	cualquier cadena de caracteres
`[...]`	cualquier carácter entre los corchetes (conjunto)
`[!...]`	cualquier carácter no perteneciente al conjunto

- **Conjuntos**

`[abc]`	**a, b o c**
`[.,;]`	punto, coma y punto y coma
`[a-c]`	**a, b o c**
`[a-z]`	Todas las minúsculas
`[!0-9]`	Ningún dígito
`[0-9!]`	Todos los dígitos y el carácter `!`
`[a-zA-Z]`	Todas las letras minúsculas y mayúsculas

- **Expansión de llaves (brace expansion)**

```
prefijo{cadenas}sufijo
$ echo ca{pa,ra,sa}s
capas caras casas
```

Control keys

- **La teclas de control (`CTRL-letra`) son otro tipo de metacaracter**
- **Normalmente no imprimen nada**
- **`RETURN` = `CTRL-m`**
  `BACKSPACE` = `CTRL-h`
- **Pueden diferir de sistema en sistema**

Control Key	Descripción
CTRL-C	Detiene el comando actual (envia SIGINT)
CTRL-D	Fin de entrad (eof)
CTRL-\	Detiene el comando actual (si no funciona CTRL-C, envia SIGQUIT)
CTRL-S	Detiene salida por pantalla
CTRL-Q	Reinicia salida por pantalla
DEL or CTRL-?	Borra último carácter
CTRL-W	Borra la última palabra de la línea de comandos
CTRL-U	Borra la linea de comandos entera
CTRL-Z	Suspende el proceso actual
CTRL-A	Cursor al principio de la línea de comandos
CTRL-E	Cursor al final de la línea de comandos
CTRL-R	Búsqueda recursiva de comandos

Quoting

- **Deshabilitar el comportamiento por defecto o imprime textualmente un metacaracter**
- **Proteger metacaracteres dentro de una cadena a fin de evitar que se reinterpreten o expandan por acción del shell**
- **Ejemplos:**

```
$ echo 2 * 3 > 1 es cierto
        # No produce salida, que pasó?
$ echo El valor de este producto es $120
El valor de este producto es 20
        # Por que sale 20 y no 120?
$ ls [Aa]+
-rw-r--r--    1   admin  admin  1250 Apr 2 15:05 Apendice.txt
-rwxrw-rw-    1   admin  admin   804 May 5 18:09 append.c
-rwxrw-rw-    1   admin  admin   539 May 9 20:58 a.out
```

- **Existen tres mecanismos de quoting**
  - El carácter de escape \ (escape character)
  - Comillas dobles " (double quotes)
  - Comillas simples ' (single quotes)

El carácter de escape

- **Es el carácter \ (backslash)**
- **Evita que el siguiente carácter sea interpretado por el shell**

```
$ echo $1234
234
$ echo \$1234
$1234
```

- **Excepción: `\newline`, esta secuencia se interpreta como continuación de línea eliminando posteriormente esta secuencia de la entrada del comando**

- **Ejemplo:**

```
$ echo Texto escrito en \
> mas de una linea
Texto escrito en mas de una linea
```

Comillas dobles

- **Los caracteres encerrados entre comillas dobles preservan su valor literal**
- **También se conoce como Weak quoting o Partial quoting**
- **Los caracteres `*` y `@` tienen un significado especial cuando se encierran con comillas dobles**

- **Excepciones:**
  - `$` y `'` siguen manteniendo sus significados especiales
  - `\` sigue manteniendo su significado especial sólo si antecede los caracteres `$`, `'`, `"`, `\` o **`newline`**.

- **Ejemplos:**

```
$ echo "El reloj tiene un valor de $123"
El reloj tiene un precio de 23
$ echo "El reloj tiene un valor de \$123"
El reloj tiene un precio de $123
$ echo "Es un vino con buen 'bouquet'"
Es un vino con buen 'bouquet'
```

Comillas simples

- **Los caracteres encerrados entre comillas simples preservan su valor literal**
- **No se permite la desreferencia de variables entre comillas simples**
- **No puede aparecer una comilla simple entre dos comillas simples**
- **También se conoce como Strong quoting o Full quoting**

- **Excepción:** `\newline`

- **Ejemplos:**

```
$ VAR=10
$ echo '$VAR'
$VAR

$ echo 'd* = el caracter <d> seguido de cualquier cadena, \
inclusive la vacía'

$ echo 'La comilla simple (') tamién es llamado aopostrofo'
```

ANSI-C Quoting

- **Las cadenas de la forma `$'texto'` son consideradas de manera especial**
  La cadena se expande a `texto` con los caracteres de escape `\` reemplazados como lo especifica el estándar ANSI-C

\a	alerta	(alerta)
\b	retroceso	(backspace)
\n	nueva línea	(new line)
\t	tab horizontal	(horizontal tab)
\v	tab vertical	(vertical tab)
****	barra invertida	(backslash)

- **Ejemplos:**

```
$ echo Barra invertida = $'\\'
Barra invertida = \

$ echo Se oye .... $'\a'
Se oye .... (beep)

$ echo Hola $'\n' Mundo
Hola
Mundo
```

Ejemplos combinados

```
$ echo <-$1250.**>; (update?) [y|n]
$ echo \<-\$1250.\*\*\>\; \(update\?\) \[y\|n\]
$ echo '<-$1250.**>; (update?) [y|n]'

$ echo La variable '$UID' contiene el valor --\> "$UID"
La variable $UID contiene el valor --> 1002

$ echo It's <party> time!
Se solicita más entrada
```

Módulo 3

Variables

Introducción (1/2)

- **El uso de variables permite crear scripts flexibles y depurables**
- **Una variable tiene un `nombre` y un valor (`$nombre`)**
- **Para permitir concatenación `${variable}cadena`**
- **Bash es case sensitive**
- **Bash es un lenguaje NO fuertemente tipado**
- **Con el shell se pueden crear, asignar y borrar variables**
  - **Creación** `$ var1=10` ***# sin espacios!***
  - **Asignación** `$ var2=$var1`
  - **Borrado** `$ unset var1`
- **El nombre de una variable puede contener sólo letras (`a-z` o `A-Z`), números (`0-9`) o guión bajo (_) y comenzar con una letra o _**
- **Las variables con nombres "numéricos" están reservadas**
- **Ejemplos:**

```
CantPersonas          cantpersonas      CANTPERSONAS
_Nueva_Variable_      producto_120      esta-mal?
1                     10mil             __________
```

- **Preguntas:**

```
$ var3 = 33
$ $var2=50
$ 3=400
```

Introducción (2/2)

- **No se advierten sobreescrituras**

- **Es posible almacenar en una variable el resultado de la ejecución de un comando.**
  - **Con acentos graves**

```
$ lista_de_archivos=`ls`
```

  - **Con `$(...)`: anidable**

```
$ lista_de_archivos=$(ls)
$ lista_de_archivos=$(ls $(cat directorios.txt))
```

- **Referencia indirecta: Si el valor de una variable es el nombre de una segunda podemos recuperar el valor de la segunda a través la primera**

```
$ dosmil=numero
$ numero=2000
$ echo $dosmil                    #Referencia directa
numero
$ eval echo \$$dosmil             #Referencia indirecta
2000
```

Variables de shell y entorno

- **Variables Locales**
  - **Presentes en la instancia actual del shell**
  - **No disponibles para programas iniciados desde el shell (no exportadas)**

- **Variables de Entorno**
  - **Disponibles por todo proceso hijo del shell**
  - **Muy útiles para la escritura de scripts y programas**
  - **Pueden visualizarse mediante el comando `env`**
  - **Se pueden agregar variables al entorno mediante `export`**
  - **Nombres en mayúsculas por convención**

- **Variables de Shell**
  - **Establecidas y utilizadas por el shell para su funcionamiento**
  - **Algunas son variables de entorno otras son locales**
  - **Pueden visualizarse mediante el comando `set`**
  - **Convencionalmente tienen nombres en mayúsculas**

    PWD **UID** **SHLVL**
    PATH **HOME** **IFS** (Internal Field Separator)

Parámetros posicionales o argumentos (1/2)

- **Son aquellas variables cuyos nombres son números**
- **Estas referencian a los argumentos de los comandos**

```
$ find / -name programa.c
```

Nombre comando = **find** ↔ **$0**
1er argumento = / ↔ **$1**
2do argumento = **-name** ↔ **$2**
3er argumento = **programa.c** ↔ **$3**
Más argumentos **$4, $5, ..., ${10}, ${11}**

- **Observaciones:**
  - **`basename $0`: nombre del comando**
  - **`dirname $0`: path al nombre del comando**
  - **`shift`: rota los argumentos hacia la izquierda `$i`←`${i+1}`, `$0` no cambia**
    Puede tomar un número `n` como argumento, así los parámetros `n+1...$#` se renombran a `1..$#-(n+1)`

Parámetros posicionales o argumentos (2/2)

- **Variables argumento especiales**
  - `$#`: **cantidad de argumentos pasados al comando**
  - `"$*"`: **todos los argumentos** `"$*"="$1 $2 $3..."`**, una sola palabra formada por los parámetros separados por el 1er carácter en** `IFS`
  - `"$@"`: **todos los argumentos** `"$@"="$1" "$2" "$3"...` **separadas por ' '**
  - `$_`: **comando previo**
  - `$-`: **flags pasadas al script**
  - `$$`: `pid` **del proceso shell**
  - `$!`: `pid` **del último trabajo ejecutándose en background**
  - `$?`: `exit status`
- `$*` **y** `$@` **difieren sólo cuando estan encerradas entre comillas dobles**

- **Nota:** **Estas variables son read-only**
- **El comando** `readonly` **establece como read-only a las variables**
- **Ejemplo:**

```
$ FRUTA=kiwi
$ readonly FRUTA
$ echo $FRUTA
kiwi
$ FRUTA=durazno    # Produce error
```

Variables enteras

- **Las variables enteras en Bash son enteros con signo (32 bits)**
- **Posibilidad de overflow**
- **Bash por sí mismo no comprende la aritmética de punto flotante**
- **Bash considera a los números conteniendo punto decimal como cadenas**
- **Utilizar el lenguaje `bc` en los scripts si es necesario realizar cálculos de punto flotante o emplear funciones matemáticas de bibliotecas**
- **<u>Ejemplos:</u>**

```
$ echo "2.35 + 56.90" | bc          → 59.25
$ echo "sqrt(2)" | bc               → 1
$ echo "sqrt(2.0)" | bc             → 1.4
$ echo "sqrt(2.00000000)" | bc      → 1.41421356
```

- **<u>Nota:</u> Veremos posteriormente más capacidades de `bc`**

Arrays

- **Un array es una serie de casillas, cada una conteniendo un valor**
- **Casilla ≈ elemento, los elementos se acceden mediante índices**
- **Los índices comienzan en 0 hasta más de 5 mil billones**
- **En bash son únicamente uni-direccionales**
- **Asignaciones:**

```
$ colores[0]=rojo
$ colores[2]=verde
$ colores[1]=amarillo
$ colores=([2]=verde [0]=rojo [1]=amarillo)
$ colores=(rojo amarillo verde)
$ colores=(rojo [10]=amarillo verde)
```

- **Los arrays se pueden declarar vacíos explicitament mediante**

```
$ declare -a colores
```

- **Los atributos establecidos para el array (read-only) se aplican a todos los elementos**
- **Para referenciar un elemento:** `${array[i]}`

```
$ echo 'No pasar' es ${colores[0]}
```

- **Para referenciar todos los elementos** `${name[*]}` **o** `${name[@]}`

```
$ echo ${colores[*]}
```

Comando `set`

- **El comando `set` despliega las variables de shell junto a sus valores**

- **Permite definir el comportamiento del bash (opciones)**

- **Sintaxis:**

```
set [-abdefhkmntuvxBCHP] [(-|+)o opcion] [argumento...]
```

`-n` Lee comandos pero no los ejecuta, útil para depurar sintácticamente scritps
`-v` Muestra las líneas de entrada del shell tal cual son leídas
`-x` Muestra una traza de un comando y sus argumentos luego de aplicar expansiones

- **Su uso habitual en shell scripts es para establecer los parámetros posicionales**

```
$ set `date`
$ echo Hora actual: $4
Hora actual: 08:40:25
```

Comando `getopts`

- **bash provee `getopts` para tratar con opciones múltiples y complejas**
- **puede utilizarse como una condición en un bucle `while`, dada la especificación del formato de opciones (validez y argumentos), en el cuerpo del `while` se procesan**
- **Sintaxis:**

  `getopts cadena variable`

  - `cadena` conteniendo letras (opciones) y :'s (argumentos)

  - `variable` que almacena el argumento de la opción que está analizándose

- **Ejemplo:**

```
while getopts ":ab:c" opt; do
    case $opt in
        a ) procesar la opción -a ;;
        b ) procesar la opción -b
            $OPTARG es el argumento de la opción ;;
        c ) procesar la opción -c ;;
        \? ) echo 'usage: alice [-a] [-b barg] [-c] args...'
        exit 1
    esac
done
```

Módulo 4

Operadores

El comando `test`

- **Evalúa una expresión condicional**

```
$ test opcion expresion
```

- **`test` retorna un 0 (`true`) o un 1 (`false`) luego de la evaluación**
- **Una manera más concisa es mediante `[...]`**

```
$ [ opcion expresion ]
```

- **Ejemplos:**

```
$ test 1 -gt 2
$ [ 1 -gt 2 ]      # espacios obligatorios [_ y _]
```

- **bash introdujo el comando extendido de `test`, `[[...]]`, con un comportamiento más familiar para los programadores**

```
$ [ -f /etc/passwd && -f /etc/group ]     # no aceptado
$ [[ -f /etc/passwd && -f /etc/group ]]   # ok
```

- **Las construcciones `((...))` y `let...` evalúan expresiones aritméticas y retornan 0 (`true`) si el resultado es distinto de cero y 1 (`false`) en caso contrario**

```
$ ((1 > 2))
$ let 1 > 2
```

`true` y `false`

- **TRUE**
  - **`true`: comando que retorna siempre un `exit status` exitoso (cero) sin hacer nada**

```
$ true
$ echo $?
0
```

  - **También evalúan a true: 0, 1, -1, "abc".**

- **FALSE**
  - **`false`: comando que retorna siempre un `exit status` no exitoso (distinto de cero) sin hacer nada**

```
$ false
$ echo $?
1
```

  - **También evalúan a false: NULL, variable no inicializada, variable nula.**

Asignación y Operadores aritméticos

- `variable=asignacion`

  Inicializa o cambia el valor de una variable

  Funciona tanto para enteros como para cadenas

  - **<u>Advertencia:</u> No confundir con el operador de comparación =**

```
$ var=27
$ animal=tigre    # Sin espacios entre el signo =
```

- **Operadores aritméticos**

`+`	**suma**
`-`	**resta**
`*`	**producto**
`/`	**cociente**
`**`	**exponenciación**
`%`	**módulo o mod**
`+=`	**más-igual**
`-=`	**menos-igual**
`*=`	**por-igual**
`/=`	**dividido-igual**
`%=`	**módulo-igual**

Operadores de bits

<< **left shift (multiplica por 2)**
<<= **left-shift-igual**

```
$ let "var <<= 2"        # como un producto de 2^n
```

>> **right shift (divide por 2)**
>>= **right-shift-igual (inverso de <<=)**

```
$ let "var >>= 3"        # como una división de 2^n
```

& **and**
&= **and-igual**
| **or**
|= **or-igual**
~ **negación**
! **negación**
^ **xor (o-exclusivo)**
^= **xor-igual**

```
$ var1=24       # 00000000 00000000 00000000 00011000
$ var2=10       # 00000000 00000000 00000000 00001010
$ let "var1 &= var2"
$ echo $var1
8
```

Operadores de comparación de enteros

			JCL
-eq	es igual a	["$a" -eq "$b"]	EQ
-ne	es distinto a	["$a" -ne "$b"]	NE
-gt	es mayor que	["$a" -gt "$b"]	GT
>	es mayor que	(("$a" > "$b"))	
-ge	es mayor o igual que	["$a" -ge "$b"]	GE
>=	es mayor o igual que	(("$a" >= "$b"))	
-lt	es menor que	["$a" -lt "$b"]	LT
<	es menor que	(("$a" < "$b"))	
-le	es menor o igual que	["$a" -le "$b"]	LE
<=	es menor o igual que	(("$a" <= "$b"))	

- **Ejemplos:**

```
$ var1=21; var2=22
$ (("$var1" <= "$var2"))
$ echo $?
0
$ [ "$var1" -le "$var2" ]
$ echo $?
0
$ (("$var1" > "$var2"))
$ echo $?
1
```

Operadores de comparación de cadenas

`=`	es igual a	`[ "$a" = "$b" ]`
`==`	es igual a	`[ "$a" == "$b" ]`
`!=`	es distinto a	`[ "$a" != "$b" ]`
`<`	es menor en orden lexicográfico	`[[ "$a" < "$b" ]]`
		`[ "$a" \< "$b" ]`
`>`	es mayor en orden lexicográfico	`[[ "$a" > "$b" ]]`
		`[ "$a" \> "$b" ]`

- **Nota:** `"<"` y `">"` necesitan ser escapados dentro de `[]`

`${#var}`	retorna la longitud del valor de `var`
`expr length $var`	idem
`expr "$var" : '.*'`	idem
`-z`	la cadena es "null" (tiene longitud 0)
`-n`	la cadena no es "null"

- **Nota:** veremos detalles del comando `expr` más adelante

Condiciones sobre archivos

```
-e          existe el archivo
-f          es un archivo regular
-r          tiene permiso de lectura
-w          tiene permiso de escritura
-x          tiene permiso de ejecución
-s          no tiene cero bytes (no es vacío)
-d          es un directorio
-b          es un block device (floppy, cdrom)
-c          es un character device (keyboard, modem, sound card)
-p          es un pipe nominado (FIFO)
-h          es un hard link
-L          es un symbolic link
-S          es un socket
-O          soy el propietario del archivo?
-G          el GID del archivo es igual al mío?
-N          se modificó desde su última lectura?
f1 -nt f2   f1 es más nuevo que f2, en relación a la actualización
f1 -ot f2   f1 es más viejo que f2, en relación a la actualización
f1 -ef f2   f1 y f2 son hard links al mismo archivo
!           "no" o negación (invierte el sentido del test, retorna
            true si la condicón no está presente)
```

Operadores lógicos

- **negación** `!expr`
- **and lógico** `expr1 && expr2`
- **or lógico** `expr1 || expr2`
- **asociatividad** `( expr )` utilizado para forzar evaluación

- **Ejemplos:**

```
$ echo -n "Hola" && echo " Mundo"
Hola Mundo
$ false && echo " Mundo"
(nada, evalúa a 1)
$ true && echo " Mundo"
 Mundo

$ echo -n "Hola" || echo " Mundo"
Hola
$ false || echo " Mundo"
Mundo
$ true || echo " Mundo"
(nada, evalua a 0)
```

Operadores misceláneos

- **Operador coma**

- Permite encadenar dos o más operaciones aritmética

- Se evalúa cada operación (con posibles efectos laterales), pero sólo se retorna la última operación

- Similar al operador coma de C

- **Ejemplos:**

```
$ let "t1 = ((5 + 3, 7 - 1, 15 - 4))"
$ echo "t1 = $t1"
t1 = 11

$ let "t2 = ((i = 9, 15 / 3))"    # se establece i (efecto lateral)
$ echo "t2 = $t2 i = $i"
t2 = 5 i = 9
```

Módulo 5

Construcciones condicionales

Sentencia `if` (1/3)

- **Sintaxis:**

```
if condición1 ;
then
    comandos1
elif condición2 ;        # /                    \
then                     # |                    |
    comandos2            # | Esto es opcional   |
else                     # |                    |
    comandos3            # \                    /
fi
```

- **Realiza acciones (ejecuta comandos) dependiendo del valor de verdad de una condición (un comando)**
- **`elif` es una abreviatura de `elseif`**
- **Las condiciones evaluan valores de variables, características de archivos, si un comando se ejecuta o no correctamente, etc**
- **Ejecución:**
  1. **Se evalúa `condición1`**
  2. **Si su `exit code` es 0, se ejecutan los `comandos1` y termina el `if`**
  3. **De otra manera se evalúa `condición2`**
  4. **Si su `exit code` es 0, se ejecutan los `comandos2` y termina el `if`**
  5. **En caso contrario se ejecutan los `comandos3` y termina el `if`**

Sentencia `if` (2/3)

- **Errores comunes:**

- Omitir el punto y coma (;) antes de la sentencia `then` al escribir el `if` de manera lineal

```
if condicion; then comandos1 else comandos2 fi
```

- Utilizar `elseif` en lugar de `elif`

- Omitir la sentencia the `then` cuando se utiliza la sentencia `elif`

- Escribir `if` en lugar de `fi` al cerrar el cuerpo del `if`

- **Ejemplos:**

```
if grep saldo clientes > /dev/null       # grep -qs saldo clientes
then
    echo "clientes contiene al menos una ocurrencia de saldo."
elif grep deuda clientes > /dev/null
    echo "clientes contiene al menos una ocurrencia de deuda."
else
    echo "Las palabras saldo y deuda no estan presentes en clientes"
fi
```

Sentencia `if` (3/3)

- **Ejemplos:**

```
#!/bin/bash
xusers=`who | wc -l`

if [ $xusers -gt 1 ] ;
then
        echo "Hay más de un usuario conectado"
else
        echo "Sólo Ud. está conectado"
fi
```

```
var1=21
var2=22
[ "$var1" -ne "$var2" ] && echo "$var1 no es igual a $var2"

# El lo mismo que
if [ "$var1" -ne "$var2" ] ;
then
    echo "$var1 no es igual a $var2"
fi

# ¿Y esto a que es igual?
[ "$var1" -ne "$var2" ] || echo "$var1 no es igual a $var2"
```

Sentencia `case` (1/2)

- **Sintaxis:**

```
case palabra in
    patron1)comandos1 ;;
    patron2)comandos2 ;;
    ....          # el patron *) es (opcional) y se utiliza como "default"
esac
```

- `palabra` **es comparada frente a cada** *`patron`* **hasta que se encuentre una coincidencia o estos se acaben**
- **se ejecutan sólo los** `comandos` **asociados con el** *`patron`* **coincidente en caso que esté presente el** `;;` **y luego se sale del** `case`**, en caso contrario se ejecutarán los** `comandos` **del** *`patron`* **inferior, etc...**
- **Los** `;;` **hacen las veces del break de C**
- **Un** *`patron`* **es una cadena consistente en caracteres comunes y metacaracteres**
- **No existe límite para la cantidad máxima de patrones aunque el mínimo es uno**
- `case` **sirve para simplificar construcciones tipo** `if/elif/.../elif/fi`
- **Su verdadero poder radica en el uso de patrones**

Sentencia `case` (2/2)

- **Ejemplo:**

```
if [ "$fruta" = banana ] ; then
    echo "La banana es rica en potasio."
elif [ "$fruta" = kiwi ] ; then
    echo "El kiwi provee vitamina C."
elif [ "$fruta" = manzana ] ; then
    echo "La manzana es baja en calorías."
fi

# Con case
case $fruta in
    banana)    echo "La banana es rica en potasio." ;;
    kiwi)      echo "El kiwi provee vitamina C." ;;
    manzana)   echo "La manzana es baja en calorías." ;;
esac

# Otro ejemplo
case $archivo in
  *.conf)       echo "$archivo es un archivo de configuracion" ;;
  *.gz|*.bz2)   echo "$archivo es un archivo comprimido" ;;
  *.[gif|jpg]) echo "$archivo es un archivo de imagen" ;;
  *)            echo "$archivo es un archivo no clasificado" ;;
esac
```

Módulo 6

Construcciones iterativas o de repetición

Sentencia `for`

- **Es la construcción de iteración básica**
- **Sintaxis:**

```
for arg in lista ;
do
    comando(s)...
done
```

- **En c/iteración `arg` toma valores de `lista` y se ejecutan *`comando(s)...`***
- **`lista` puede contener metacaracteres** `for i in hoja*`
- **Los elementos en `lista` pueden tener argumentos**
- **Si `do` está en la misma línea que `for` es necesario el punto y coma**
- **Ejemplos:**

```
for planeta in Mercurio Venus Tierra Marte ;
do
    echo $planeta
done

for file in /var/*.bck ;
do
    rm $file
done
```

Sentencia `select` (1/2)

- **Sintaxis:**

```
select name [in lista];     # los [] están indicando opción
do                          # no es parte de la sintaxis
    comando(s)...
done
```

- **Provee una manera simple de crear menús numerados para que los usuarios seleccionen una de las opciones en `lista`**
- **Introducido por `ksh` posteriormente incorporado por `bash`**
- **Ejecución:**
  1. **Se muestra cada elemento de `lista` acompañado de un número**
  2. **Se muestra el prompt `PS3`, generalmente es `#?`**
  3. **Cuando el usuario ingresa un valor se almacena en `$REPLY`**
  4. **Si `$REPLY` contiene un número válido, `name` se establece a la variable asociada a dicho número.**
     Si `$REPLY` contiene un número NO válido, `name` será nula.
     Si se ingresa una línea vacía se muestra `lista` y el prompt nuevamente.
     En caso de EOF, se finaliza el `select`.
  5. **Se ejecutan los *`comando(s)...`* en los primeros dos casos.**
  6. **Si no se utiliza ningún mecanismo de control de bucles se continúa con el paso 1**

Sentencia `select` (2/2)

- **Ejemplo:**

```
#!/bin/bash
PS3='Elija su comida favorita: '  # Establece el prompt
echo
select food in "Asado" "Lasagna" "Paella" "Pizza"
    do
        [ -e "$food" ] && continue
        echo "Su comida favorita es $food."
        break
    done
exit 0
```

- **Nota: Si se omite `lista`, `select` utilizará los argumentos pasados al script (`$@`), o a la función que contiene el `select`**

- **Nota: Más adelante, en este mismo módulo veremos el significado de *break* y *continue***

Sentencia `while` (1/2)

- **Sintaxis:**

```
while comando ;          # comando ≈ condición ≈ comando test
do
    comando(s) . . .
done
```

- **Se ejecuta mientras que la condición sea verdadera**
- **Para repetir *comando(s) . . .* un número de veces no preestablecido**
- **Ejecución:**
  1. **Se ejecuta `comando`**
  2. **Si el `exit status` de `comando` es distinto de cero, finaliza el `while`**
  3. **Si el `exit status` de `comando` es cero, se ejecuta *comando(s) . . .***
  4. **Cuando finaliza *comando(s) . . .* se retorna al paso 1**
- **Sintaxis in-line:** `while comando ; do comando(s) . . . ; done`
- **Ejemplo:**

```
x=0
while [ $x -lt 10 ]
do
    echo -n $x
    x=$(( $x + 1 ))
done
```

```
# SALIDA #
0123456789
```

Sentencia `while` (2/2)

- **Bucles anidados**

```
while condición1 ;                  # bucle principal
    do
        comandos...
        while condición2 ;          # bucle anidado
        do
            comandos...
        done
        comandos...
    done
```

- **No hay restricciones relacionadas con el nivel de anidación**

```
x=0
while [ "$x" -lt 10 ] ;
do
    y="$x"
    while [ "$y" -ge 0 ] ;
    do
        echo "$y \c"
        y=$(( $y - 1 ))
    done
    echo $x
    x=$(( $x + 1 ))
done
```

```
        # SALIDA #
0



...
9
```

Sentencia `until`

- **Sintaxis:**

```
until comando ;        # comando ≈ condición ≈ comando test
do
    comando(s) . . .
done
```

- **Se ejecuta hasta que la condición sea verdadera**
- **Es la "negación" del bucle `while`**
- **Ejecución:**
  1. **Se ejecuta `comando`**
  2. **Si el `exit status` de `comando` es cero, finaliza el `until`**
  3. **Si el `exit status` de `comando` no es cero, se ejecutan *`comando(s) . . .`***
  4. **Cuando finalizan los *`comando(s) . . .`* se retorna al paso 1**
- **Sintaxis in-line:** `until comando ; do comando(s) . . . ; done`
- **Ejemplo:**

```
x=1
until [ $x -gt 10 ]          # while [ ! $x -gt 10 ]
do
    echo -n $x
    x=$(( $x + 1 ))
done
```

Comandos `break` y `continue`

- **Existe el riesgo de escribir bucles infinitos**

```
x=0
while [ x -lt 10 ]          # Error, va $x !!!
do
    echo $x
    x=$(( $x + 1 ))
done
```

- **`break` finaliza el bucle que la contiene**
- **`continue` provoca un salto hasta la próxima iteración, saltando el resto de los comandos de la iteración actual**
- **ambas construcciones pueden eventualmente tomar un parámetro numérico, `break N` ≈ finaliza los bucles hasta el nivel `N` de anidación**
- **Ejemplo:**

```
while :
do
    read CMD
    case $CMD in
        [qQ]|[qQ][uU][iI][tT]) break ;;
        *) process $CMD ;;
    esac
done
```

Módulo 7

Funciones

Introducción

- **Las funciones mejoran la programación de shell scripts:**
  - **Cuando se invoca una función, la misma estará previamente cargada en memoria → velocidad de ejecución**
  - **Modularización, depuración, reutilización de código**
- **Es una implementación limitada respecto a otros lenguajes**
- **Necesariamente deben estar definidas antes de su invocación**
- **Dentro de una función `$1`, `$2`, `$3`,... serán los parámetros pasados a la misma, no al script en sí**
- **Sintaxis permitidas:**

```
function nombre_funcion {
    comando...
}
```

```
nombre_funcion () {
        comando...
}
```

- **Ejemplo:**

```
#!/bin/bash

saludo () {
    echo -n "Hola "
}

saludo
echo "Mundo"
```

Variables locales y globales

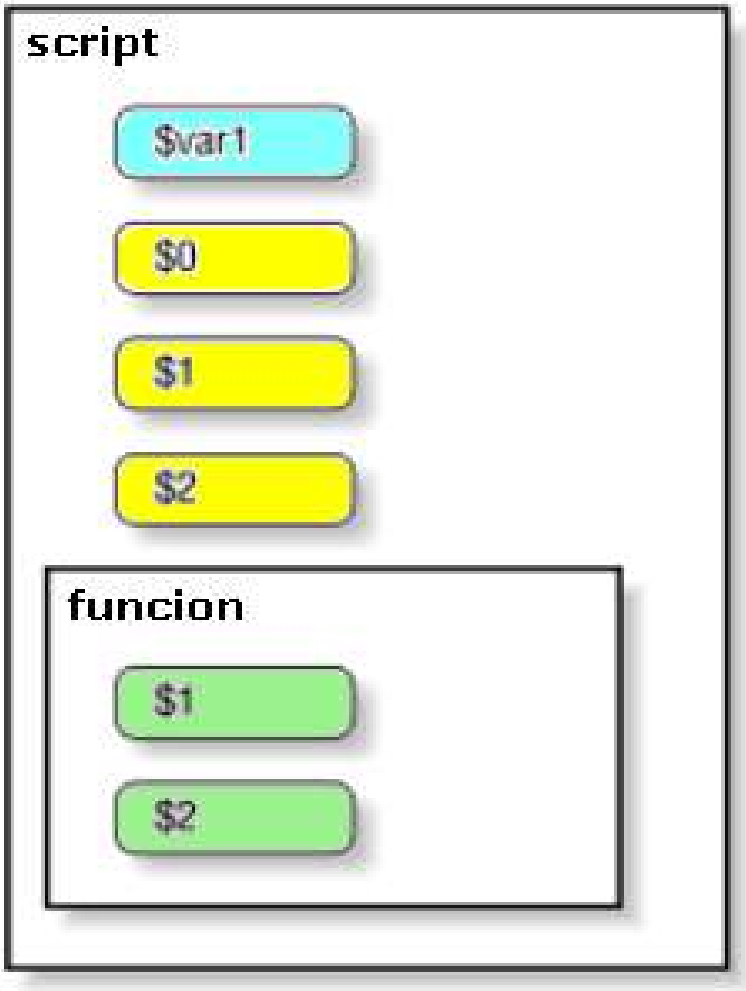
- **Una variable es considerada como local si sólo es visible dentro del bloque de código en el cual aparece, es decir tiene alcance (scope) local. En el caso de funciones las variables locales sólo tienen significado dentro de la función.**

- **`$var1` es visible tanto para `sript` como para `funcion`. Es una variable global**
- **`$0, $1, $2` son visibles sólo para `sript`**
- **`$1` y `$2` son visibles sólo para `funcion`**

- **<u>Nota:</u>**

`funcion` puede definir una variable local denominada `var1`, pero debe anteponer la palabra reservada local

```
local var1
```



Visibilidad y definiciones

- **las declaraciones de funciones pueden aparecer en casi cualquier lugar, inclusive donde iría un comando**

```
$ ls -l | f () { echo "22"; } # Correcto, aunque inútil
```

- **bash admite funciones anidadas**

```
#!/bin/bash
f1 () {
    f2 () { # funcion anidada
        echo "Funcion \"f2\", dentro de \"f1\"."
    }
}

f2      # Provoca un error
```

Modularización y terminación

- **Con source o con "." podemos incluir el código de otro script en el nuestro:**

```
#!/bin/bash
source funciones.sh     # en funciones.sh se define suma
suma 1 3
suma 12 12312
```

- **exit y retrun**

- **exit status**

  Es el valor que retorna la función.
  Puede ser especificado explícitamente por **retrun**, de otra manera será el exit status del último comando ejecutado. Se obtiene su valor mediante $?

- **retrun [n]**

  Termina una función, el argumento entero opcional es retornado al script invocante como el exit status de la función y se asigna a $?

- **Nota:** El entero positivo más grande que una función puede retornar es 255
  Las versiones superiores a 2.05b del bash no sufren de esta limitación.

Terminación ejemplo

```
#!/bin/bash
# Verificando el valor del máximo entero que puede retornarse
return_test () {
    return $1
}

return_test 27          # bien
echo $?                 # imprime 27
return_test 255         # bien aun
echo $?                 # imprime 255
return_test 256         # Error!
echo $?                 # imprime 1 (código "error miscelaneo")
```

Alternativa: uso de variable global

```
Return_Val= # Variable global para almacenar el valor de retorno
alt_return_test () {
    Return_Val=$1
    return # Returns 0 (success).
}

alt_return_test 256
echo "return value = $Return_Val"          # 256
```

Módulo 8

Entrada y Salida
Redirecciones

Introducción

- 'Redireccionar' significa capturar la salida de un archivo, programa, script o hasta de un bloque dentro de un script y enviarla como entrada a otro archivo, comando, programa o script
- Cuando se ejecuta un comando se abren y asocian con el mismo tres archivos representados por números bajos (file descriptors)

- Se puede asociar cualquier archivo con descriptores de archivos mediante el comando `exec` (facilitar la referencia)

```
$ exec 4>capitulo4.txt
```

- bash ofrece muchos operadores de redirección

Interacción con el usuario (1/3)

- **`read`**
- **Sintaxis:**

```
read [-a aname] [-p prompt] [-er] [variable...]
```

- **Lee el valor de una variable desde `STDIN`**
- **Con la opción `-a` permite leer variables tipo array**
- **Ejemplos:**

```
#!/bin/bash
echo -n "Ingrese su nombre: "
read name
echo "name = $name"

echo -n "Ingrese su fecha de nacimiento (dd mm aaaa): "
read dia mes anyo
```

- **Un `read` sin variable(s) asociada(s) provoca que se establezca `REPLY`**
- **Algunas opciones de `read`**

`-t`	Entrada temporizada
`-p` prompt	Permite establecer un prompt para guiar la entrada
`-n` N	Aceptar sólo N caracteres de la entrada

Interacción con el usuario (2/3)

- `echo`
- **Sintaxis:**

  `echo [-neE] [arg...]`
- **Imprime texot por `STDOUT`**
- **Por defecto imprime un carácter `nueva-linea`, con `-n` se suprime**
- **Con la opción `-e` interpreta caracteres especiales**

`\b`	**Backspace**
`\c`	**Suprime `nueva-linea`**
`\f`	**Formfeed**
`\n`	`nueva-linea`
`\r`	**Retorno de carro**
`\t`	**Tabulador**
`\\`	**Backslash**
`\nnn`	**Caracter cuyo valor ASCII es el octal `nnn`**

- **Ejemplos:**

```
$ a=`echo "HOLA" | tr A-Z a-z`
$ echo $a
hola
$ echo -e "Nueva \nLinea"
Nueva
Linea
```

Interacción con el usuario (3/3)

- **`printf`**
- **Sintaxis:**

  `printf cadena-de-formato... [parametro...]`
- **Es una mejora a `echo`, versión limitada de la función `printf()` de C**
- **Se utiliza para formatear texto de salida**

- **Generalmente ubicado en `/bin` o `/usr/bin`**

- **Ejemplos:**

```
PI=3.14159265358979
NRO=1234
Palabra1=Shell
Palabra2=Scripts

printf "Pi con dos cifras decimales = %1.2f" $PI
printf "Numero = \t%d\n" $NRO
printf "[%s\t%s]\n" $Palabra1 $Palabra2
```

Tabla de operadores de redirección (1/2)

`cmd1 | cmd2`	**Pipe, toma STDOUT de `cmd1` como STDIN del `cmd2`**
`> arch`	**STDOUT a `arch`, si `arch` existe lo reemplaza**
`< arch`	**Toma STDIN desde `arch`**
`>> arch`	**STDOUT a `arch`, escribe al final si `arch` existe**
`>| arch`	**Fuerza STDOUT a `arch` aun si `noclobber` está establecido**
`n>| arch`	**Idem para el archivo cuyo descriptor de archivo es `n`**
`<> arch`	**Usa `arch` como STDIN tanto como STDOUT**
`n<> arch`	**Usa `arch` como STDIN tanto como STDOUT para el descr. arch `n`**
`<< label`	**Here document**
`n> arch`	**Direcciona el descriptor de archivo `n` a `arch`**
`n< arch`	**Toma el descriptor de archivo `n` desde `arch`**
`n>> arch`	**Direcciona el descriptor de archivo `n` a `arch`, agrega datos al final de `arch` si ya existe**

Tabla de operadores de redirección (2/2)

`n>&`	**Duplica STDOUT del descriptor de archivo `n`**
`n<&`	**Duplica STDIN del descriptor de archivo `n`**
`n>&m`	**El descriptor de archivo `n` se transforma en una copia del descriptor de archivo de salida `m`**
`n<&m`	**El descriptor de archivo `n` se transforma en una copia del descriptor de archivo de entrada `m`**
`&>arch`	**Direcciona STDOUT y STDERR a `arch`**
`<&-`	**Cierra STDIN**
`>&-`	**Cierra STDOUT**
`n>&-`	**Cierra STDOUT del descriptor de archivo `n`**
`n<&-`	**Cierra STDIN del descriptor de archivo `n`**

Ejemplos

```
$ cat >> .bashrc
alias ll='ls -lha'
alias rm='rm -i'
^D
```

```
$ cat < archivo1 > archivo2
# similar a cp archivo1 archivo2
```

```
$ lpr << MIS_URLS
http://sun.sunsolve.com
http://docs.sun.com
http://www.sunfreeware.com
MIS_URLS
```

```
$ echo "Mensaje" 1>&2
```

Módulo 9

Herramientas misceláneas

Introducción

- **Herramientas útiles para la confección de scrpits**

- **Comandos propios del shell (built-in)**

```
eval
: (dos puntos)
type
```

- **Comandos externos**

```
sleep
find
xargs
expr
bc
```

- **Los comandos built-in se ejecutan de manera más eficiente que los externos pues no existen accesos a disco para su ejecución**

Comando `eval`

- **Puede utilizarse cuando se desea que el shell reprocese la línea de comandos por segunda vez**
- **Sintaxis:**

```
eval cualquier_comando
```

- **Ejemplo:**

```
$ SALIDA="> out.file"
$ echo hola $SALIDA
hola > out.file
$ eval echo hola $SALIDA
$    # se creo el archivo out.file
```

- **Luego se puede cambiar el valor original de `SALIDA` y así afectar a todas las líneas que comiencen con eval y contengan `$SALIDA`**
- **Útil para el caso en que se desea componer una línea de comandos utilizando metacaracteres contenidos en variables o producidos debido a sustitución de comandos**

Comando :

- **No hace nada, retorna 0 indicando ejecución de comando exitoso**
- **Puede utilizarse con seguridad en cualquier lugar donde se requiera un comando por cuestiones puramente sintácticas**
- **Ejemplos:**

```
if [ -x $CMD ]
then
    :
else
    echo Error: $CMD no es ejecutable >&2
fi
```

- **Construcción de bucles infinitos**

```
while :                          # más eficiente que while true
do
    echo "Ingrese una palabra: \c"
    read ENTRADA
    [ "$ENTRADA" = "basta" ] && break
done
```

- **El uso de : provoca que el shell evalúe sus argumentos, esto es útil para invocar sustituciones de variables**

Comando `type`

- **Brinda el path absoluto de uno o más comandos**
- **Sintaxis:**

```
type comando...
```

- **Si `comando` no es un comando externo al shell, `type` responderá**
  - **Es un comando built-in**
  - **Es una palabra reservada del shell**
  - **Es un alias**

- **Si es un alias de un comando, `type` también dará el comando subyacente**

- **Ejemplo:**

```
$ type true vi case eval history
true is /bin/true
vi is /usr/bin/vi
case is a keyword
eval is a shell builtin
history is an exported alias for fc -l
```

Comando `sleep`

- **Provoca una pausa por un número determinado de segundos**
- **Sintaxis:**

  `sleep n`

- **Algunos tipos de `UNIX` permiten otras unidades de tiempo**
- **Puede ser usado para ofrecer una cantidad de segundos al usuario para que lea un pantalla antes de que esta desaparezca**
- **Ejemplos:**

```
echo -e "Ingrese un valor!\a"
sleep 1
echo -ne "\a"
sleep 1
echo -ne "\a"
```

```
while :
do
    echo ========================================
    date
    echo ========================================
    who
    sleep 300          # 5 minutos
done >> logfile
```

Comando `find` (1/2)

- **Comando muy poderoso, provee un mecanismo muy flexible para crear una lista de archivos que verifican ciertos criterios**
- **Sintaxis:**

  `find directorio_partida opciones acciones`

- **Ejemplos:**

```
$ find / -name core -print          # -print es considerado por defecto
$ find / -name '*core*'             # * ? [caracteres] [!caracteres]
$ find / -type d
$ find / -size +2000                # 2000 bloques de 512 bytes
$ find / -mtime -5
```

  +n Modificados hace más de n días atrás
  n Modificados exactamente hace n días
  -n Modificados hace menos de n días

  -mtime ↔ **Modificación**
  -atime ↔ **Acceso**
  -ctime ↔ **Creación, permisos, propiedad**

```
$ find / -atime +100 -exec rm -i {} \;
```

Comando `find` (2/2)

- **Combinación de criterios**

- And

```
$ find / -size +50 -mtime -3 -print
$ find / -size +50 -print -mtime -3      # se ignora -mtime -3
```

- Or

```
$ find / \( -size +50 -o -mtime -3 \)
```

- Not

```
$ find /dev ! \( -type b -o -type c -o type d \)
```

- **Si mediante `-exec` hay que procesar muchos archivos como en**

```
$ find / -name core -exec rm -i {} \;
```

utilizar en su lugar `xargs` como en

```
$ find / -name core | xargs rm -i
```

Este comando también borra los archivos `core` pero mucho más rápido y con menos sobrecarga que `-exec` que llama `rm -i` para cada archivo

Comando `xargs`

- **Acepta una serie de palabras desde la `STDIN` y las dispone como argumentos para un comando dado**
- **Ejemplo:**

```
$ cat lista_archivos | xargs rm
```

- **Si el número de archivos es demasiado grande `xargs` ejecuta `rm` varias veces**
- **Puede especificarse cuantos argumentos de `STDIN` serán procesados como argumentos para el comando mediante la opción `-n`**

```
$ cat lista_archivos | xargs -n 20 rm    # borrar de 20 en 20
$ ls
capitulo1     capitulo2     caratula
notas         prologo
$ ls | xargs -n 2 echo --
-- capitulo1 capitulo2
-- caratula notas
-- prologo
```

- **Problema**

```
$ rm a*
rm: arg list too long
$ ls | grep '^a' | xargs -n 20 rm
```

Comando `expr`

- **Útil para realizar operaciones aritméticas enteras simples en notación infija**
- **Sintaxis:**

  `expr operando1 operador operando2`

- **Operadores:**
  - `+` **Suma**
  - `-` **Resta**
  - `\*` **Multiplicación (operador quoted)**
  - `/` **División entera**
  - `%` **Resto de la división**
- `expr` **requiere espacios para separar sus argumentos**
- **Ejemplos:**

```
$ expr 3 + 2
5
$ expr "ABRIL" : '[A-G]*'
2
$ mail=diego@mail.com"
$ [ expr "$mail" : '.*' -eq expr "$mail" : '.*@.*\..*']

//En scritps CNT=´expr $CNT + 1´
```

Comando `bc`

- **Potente utilidad para realizar cálculos numéricos no restringida a los entros**

```
$ bc
scale=4
8/3
2.6666
2.5 * 4.1/6.9          # Los espacios son opcionales
1.4855
quit
```

- **Operadores:**
  - `+` **Suma**
  - `-` **Resta**
  - `*` **Multiplicación (operador quoted)**
  - `/` **División entera**
  - `%` **Resto de la división**
  - `^` **Exponenciación**

- **Opera de manera precisa con números de cualquier tamaño**

```
9238472938742937 * 2938472947298472
27147002688730233964784462089226 4
```

Comando `bc`

- **Usos típicos en scripts**

```
AVERAGE=´echo "scale=4; $PRICE/$UNITS" | bc´
PI=$(echo "scale=10; 4*a(1)" | bc -l)     # a: atan(x)
                                          # -l: mathlib
```

- **Permite realizar conversiones en diferentes bases**

  Es importante establecer primero la base output

```
$ bc
obase=16                # base output = hexadecimal
ibase=8                 # base input  = octal
400
100
77
3f
10*3
18
quit
```

Módulo 10

Filtros

Filtros

- **Comandos que leen alguna entrada, realizan una transformación de la misma o calculan valores mediante esta y escriben alguna salida**

- **Comandos de filtrados típicos**

- **`sort`:** Ordena líneas de archivos de texto (**`sort -k`**, **`sort -f`**)
- **`tr`:** Traduce o elimina caracteres (trabaja con **`STDIN`**)
- **`uniq`:** Elimina líneas duplicadas en un archivo ordenado (**`sort -u`**)
- **`head`:** Muestra las primeras líneas de un fichero
- **`tail`:** Muestra las últimas líneas de un fichero
- **`wc`:** Estadísticas simples para archivos de texto
- **`rev`:** Invierte cada línea de una archivo y la envía a **`STDOUT`**
- **`cat`:** Concatena el contenido de uno o más archivos (**`cat -n`**)
- **`cut`:** Imprime partes seleccionadas de los archivos de entrada (**`cut -b`**)

Módulo 11

Lenguaje **awk**

Introducción

- **Aho - Weinberger - Kernighan**
- **La versión original fue escrita en 1977 en los laboratorios AT&T**
- **En 1985 se escribió una versión más potente (funciones, expresiones regulares)**
- **`awk` es idóneo para obtener informes o resúmenes a partir de "datos crudos" o de la salida de otros programas**
- **Está más relacionado a un lenguaje de programación (C) que a un editor, como lo está `sed`**
- **Los programas `awk` son breves en general permitiendo su fácil composición con otros comandos y programas**

Generalidades

- **Forma general**

```
awk 'programa' archivo ...
```

- **Un `programa awk` consiste en una o más reglas de la forma**

```
patrón { acción }    # las llaves separan la acciones de los patrones
patrón { acción }
...
```

- **Perspectiva `awk` de la entrada**

  `archivo` = secuencia de líneas o `registros`

  `registros` = secuencias de `campos`

- `awk` **consume cada** `archivo ...` **registro por registro analizando si entre sus campos "está presente" el** `patrón`**, si es así se aplica la** `acción` **sobre este registro**

- `awk` **es case sensitive**

- `awk` **NO altera al contenido de los** `archivo ...`

Primeros ejemplos

```
$ awk '/abc/ {print $0}' archivo
```

- **Nota:** Barras **/** y comillas **'**
- **El patrón o la acción pueden omitirse pero no ambos a la vez**
  - `patrón` **omitido → se realiza la acción para cada línea**
  - `acción` **omitida → la acción por defecto es replicar la línea**

- **Ejemplos:**

```
awk '{print $0}' archivo   ≡   cat archivo
awk '/abc/' archivo        ≡   grep abc archivo
```

- **Ejemplos más elaborados:**

```
$ awk '/100/ {print $0}
       /200/ {print $0}' cuotas_pagas impuestos
```

Nota: si en una línea aparece tanto 100 como 200 se imprimirá dos veces!

```
$ ls -l | awk '$6=="Jun" { sum+=$5 } END { print sum }'
```

Ejecución de programas `awk`

- **En línea de comandos, eventualmente como parte de un pipeline**

```
awk 'programa' archivo…
```

- **Como script**

```
awk –f script.awk archivo…
```

- **Sin archivos de entrada aplica el programa a la entrada estándar (`Ctrl-d`) para terminar**

```
awk 'programa'    o     awk -f sript.awk
```

- **Como ejecutable**

```
#! /bin/awk –f
# Primer script
BEGIN {print "Hola Mundo"}
```

Sentencias y líneas

- **Cada linea de un programa `awk` es una sentencia o regla independiente**

- **Una sentencia puede ocupar más de una línea dividiéndola con nueva-línea detrás de alguno de los caracteres:**

  `,  {  ¿  :  ||  &&  do  else`

  de otra manera la nueva-línea indicaría fin de sentencia

- **Se puede cortar también con el carácter `\` en cualquier punto**

```
awk '/oferta de fin de mes/ \
  {print $0}' lista-precios
```

- **El carácter `;` se utiliza para colocar más de una sentencia en una misma línea**

```
awk '/100/ {print $0} ; /200/ {print $0}' impuestos
```

Interpretación de la entrada

- **awk consume la entrada en registros, divididos a su vez en campos**
- **Con las variables built-in se puede cambiar esto**

- variable `RS` (`Record Separator`): separador de registros

```
awk 'BEGIN {RS="/"}; {print $0}' fechas
//cambia RS antes de consumir entrada
```

- `RS=""` indica que los registros serán separados por líneas en blanco

- variable `FS` (`Field Separator`): separador de campos

- variable `FNR`: almacena la cantidad de registros leídos hasta el momento del archivo actual, este valor se reinicia con cada archivo

- variable `NR` (`Number of Records`) guarda el número total de registros leídos (no se reinicia automáticamente)

- variable `NF` (`Number of Fields`) guarda la cantidad de campos en un registro

- **Referenciando campos:**
  - `$0`: Registro completo
  - `$1`: Primer campo del registro
  - `$2`: Segundo campo del registro
  - ...
  - `$NF`: Último campo del registro

Patrones (1/2)

`/expresión regular/`

- **El texto del registro de entrada concuerda con la expresión regular**
- **Ejemplos:**

`/texto/`	**Concordancia con `texto`**
`/x+/`	**Una o más ocurrencias de `x`**
`/x?/`	**Una o ninguna ocurrencia de `x`**
`/x|y/`	**Tanto `x` como `y`**
`(string)`	**Agrupa una cadena para usar con `+` o `?`**
`$1 ~ /exp-reg/`	**Concordancia de campo**

`expresión`

- **Vale cuando su valor es distinto de 0 (si representa un número) o no nulo (si representa una cadena)**
- `expresión` **se evalua cada vez que la regla se chequea contra un nuevo registro**
- **Puede ser una expresión de la forma `$N`, que dependa del registro actual**

Patrones (2/2)

`patron1, patron2`

- **Especifican un rango de registros**
- `patron1` **(patrón de inicio),** `patron2` **(patrón de fin)**
- **Cuando un registro concuerda con** `patron1` **comienzan a chequearse los siguientes registros contra** `patron2`**, luego realiza la misma tarea desde el ppio**

```
$ awk '$1 == ”17/05/2005”, $2 == ”20/06/2005”'
```

`BEGIN  END`

- **Son patrones especiales, no se utilizan como filtro**
- **Brindar la posibilidad de realizar tareas antes y después de procesar la entrada**
- **Tanto** `BEGIN` **como** `END` **se ejecutan sólo una vez**
- **No pueden ser usados en rangos o con operadores booleanos**
- **Deben estar acompañados por accion(es) pues no existen acciones por defecto para estas reglas**

`null`

- **Especifican cualquier secuencia de caracteres, es decir, todos los registros concuerdan con este**

Operando con patrones

- **Expresiones de comparación de patrones**

`x < y`	Verdad si `x` es menor que `y`
`x <= y`	Verdad si `x` es menor o igual que `y`
`x > y`	Verdad si `x` es mayor que `y`
`x >= y`	Verdad si `x` es mayor o igual que `y`
`x == y`	Verdad si `x` es igual a `y`
`x != y`	Verdad si `x` es distinto de `y`
`x ~ y`	Verdad si `x` concuerda con la expresión regular `y`
`x !~ y`	Verdad si `x` no concuerda con la expresión regular `y`

- **Operadores booleanos de patrones**

`!`	no
`||`	o
`&&`	y

- **Ejemplos:**

```
$ awk '$1 == 5000 { print $1, $2}'

$ awk '/100/ && /200/'

$ awk '! /100/'

$ awk '/sa/ || /srl/'
```

Acciones

- **Una acción = una o más sentencias `awk` y se encierra entre llaves `{}`**
- **Las sentencias son separadas por caracteres nueva-linea o ;**
- **Tipos de sentencias**

- Expresiones

Invocación de funciones, asignación de variables, variables y funciones built-in

- Sentencias de control

Similares a las del lenguaje C: `if`, `for`, `while`, `do-while`, `break`, `continue`

- Sentencias compuestas

Consisten en una o más sentencias encerradas entre llaves, `bloques`

- Control de entrada

`getline` y `next`

- Sentencias de salida

`print` y `printf`

- Sentencia de borrado

Para eliminar elementos de un array

Sentencias `print` y `printf` (1/2)

- **Son de las más utilizadas como acción**
- `print (item1, item2, ...)` # los paréntesis son necesarios a veces
- `print` ≡ `print $0`
- `print ""` ≡ línea en blanco
- **Separadores de salida**
  - **OFS Output Field Separator** **(espacio por defecto)**
  - **ORS Output Record Separator** **(nueva-linea por defecto)**
- **Ejemplos:**

```
$ awk 'patron { print }' a_ent > a_sal
// $ grep patron a_ent > a_sal
$ awk 'BEGIN { print "linea uno\nlinea dos" }'
linea uno
linea dos
$ date | awk '{ print $1,$2}'
Ene 14
$ date | awk '{ print $1,$2}'
Ene14
$ date | awk '{  print "Precio Unitario"
                  print "====== ========" }' factura
$ awk 'BEGIN { OFS = ";"; ORS = "\n\n" }
              { print $1, $3, $NF }' listado
```

Sentencias `print` y `printf` (2/2)

- `printf formato item1, item2, ...` # los ( ) son necesarios a veces
- `formato` **es una cadena que indica como deben imprimirse los ítems**
- **es una versión inferior a la función `printf()` de C**
- **no coloca automáticamente un carácter nueva-linea al final**

```
printf "%f\n", $2
```

- **Ejemplos:**

```
$ awk '{ printf "(%-10s, %d, %c)\t", $1, $3, $4 }' inventario
$ awk 'BEGIN { format = "%-10s %s\n"
               printf format, "Nombre", "Número"
               printf format, "------", "------" }
       { printf format, $1, $2 }' archivo_con_datos
```

- **Redirecciones**

```
$ awk '{ print $2 > "lista-telefonos"
         print $1 > "lista-nombres" }' archivo_con_datos

$ awk '{ print $1 > "nombres-desordenados"
         print $1 | sort -r > "nombres-ordenados" }' archivo_con_datos
```

“one-liners” útiles (1/2)

- **awk 'NF > 0'** # sólo líneas no vacías

- **ls -l archivos |**
  awk '{ x += $4 } ; END { print "total bytes: " x }'

cantidad de bytes ocupados en un directorio

- **awk '{ if (NF > max) max = NF }**
  END { print max }'

imprime el número máximo de campos entre los registros

- **awk '{ nlines++ } ; END { print nlines }'**
- **awk 'END { print NR }'**

cantidad de líneas en archivo(s), **wc -l**

- **awk '{ print NR, $0 }'**

numera toda línea de entrada, **cat -n**

“one-liners” útiles (2/2)

- **`awk 'BEGIN { FS = ":" }`**
  `{ print $1 | "sort" }' /etc/passwd`

```
# imprime una lista ordenada de los nombres de usaurio
```

- **`awk 'length($0) > 80'`**

```
# imprime las líneas que contengan más de 80 caracteres
```

Ejemplos varios

-
```
awk '{ i = 1                          # imprime los primeros tres campos
       while (i <= 3) {               # de cada registro
          print $i
          i++                         # i=$i+1
        }
  }'
```

-
```
awk '{ for (i = 1; i <= 3; i++)  # idem anterior
          print $i
  }'
```

- Bash script con código awk embebido

```
#!/bin/bash

PASSWORD_FILE=/etc/passwd
n=1    # Numero de usuario

for name in $(awk 'BEGIN{FS=":"}{print $1}' < "$PASSWORD_FILE" )
do
    echo "USER #$n = $name"
    let "n += 1"
done
```

Módulo 12

Lenguaje `sed`

sed

- **Es un "editor de flujo" (stream editor)**
- **Deriva del editor `ed`**
- **Consume cada línea de los archivos de entrada aplicando comandos a las mismas, sin alterar a los mismo**
- **Sintaxis:**

```
sed opciones 'comando_ed' archivos…
```

- **Nota: las comillas " son generalmente indispensables debido a los metacaracteres (evitar expansiones)**
- **Opciones:**

-n: indica que se suprima la salida estándar

-e script: indica que se ejecute el script que viene a continuación.
Si no se emplea la opción -f se puede omitir -e.

-f archivo: indica que los comandos se tomarán de un archivo

- **Los comandos son de la forma `/patron/ accion`**
  - `patron` **es una expresión regular**
  - `accion` **es uno de los siguientes comandos (hay más)**

`p`	**Imprime la linea**
`d`	**Borra la linea**
`s/p1/p2/`	**Sustituye la primer ocurrencia de `p1` con `p2`**

Ejemplos de uso de sed

```
# Comportamiento por defecto: imprimir la entrada a STDOUT
sed kq ≡ head -k          # Del ppio a la línea k-ésima
sed -n '1p' archivo       # Devuelve la 1er línea, -n sin el texto original
sed -n '4,6p' archivo     # Devuelve las líneas 4 a la 6
Otra manera de hacer lo anterior: sed 10,20\!d
sed -n '4,$p' archivo     # Devuelve las líneas 4 al final ≡ tail -4
sed -n /^E/ archivo       # Devuelve toda línea que empieza con "E"
sed -n '/^E/,$p' arch     # Desde la 1er línea que empiece con "E"
                            al final
sed '3d' archivo          # Borra la 3ra línea de archivo
sed 'a\Linea nueva' archivo
                          # Añade la línea con el contenido "Linea
                            nueva" después de cada línea de archivo

$ sed '/0\.[0-9][0-9]$/p' precios.txt
$ sed -n '/0\.[0-9][0-9]$/p' precios.txt
$ sed '/^[Bb]lanco/d' colores.dat > colores-nuevo.dat
```

Ejemplos de uso de sed

- **Sustituciones**

/patron/s/patron1/patron2

- patron1 es reemplazado por patron2 en toda línea que concuerde con patron
- se realiza un reemplazo por línea (comportamiento por defecto)
- si se omite patron se habilita el reemplazo para todas las líneas

```
$ sed 's/Celeste/Celsete/' colores
```

- para realizar más de un reemplazo se utiliza g (global)

```
$ sed 's/mas/más/g' carta
```

- sólo el reemplazo tiene efecto en las líneas 1 a la 3

```
$ sed '1,3s/sa/SA/g' carta
```

- sustituciones homólogas: 1→x, 2→y, 3→z

```
$ sed 'y/[123][xyz]/g/' listado
```

- reutilizando una expresión regular

```
$ sed 's/ *[0-9][0-9]*\.[0-9][0-9]$/\$&/' precios.txt
```

- comandos múltiples

```
sed -e 'comando1' -e 'comando2' ... -e 'comandoN' archivos
```

Módulo 13

Depuración de scripts
Ejercitación

Depuración de scripts (1-2)

- Debugging = detección y eliminación de problemas ligados a la correctitud (sintáctico-lógica) y el desempeño (performance)

- "Síntomas" de un script con errores:
  - Arroja un error sintáctico, detiene su ejecución
  - Se ejecuta, pero no trabaja como debería, error lógico
  - Se ejecuta como era esperado, pero realiza efectos laterales no deseados (bomba lógica)

- echo's es la primer opción, obviamente rudimentaria

TRAZA

- Comunmente se inicia un subshell con la opción `-x` establecida para ejecutar un script en modo debug

```
$ bash -x script.sh
```

Esto produce una "traza", cada comando con sus argumentos en `script.sh` se imprimen por STDOUT, luego de realizar expansiones y antes de ejecutarlos

Depuración de scripts (2-2)

- **Opciones de debugging**

`set -f` `set -o noglob`

Desactiva generación de nombres de archivo usando metacaracteres (**globbing**)

`set -v` `set -o verbose` **I**

Imprime comandos luego de ejecutarlos. **Visualizar que produjo la salida**

`set -x` `set -o xtrace`

Imprime la traza de los comandos antes de ejecutarlos

`set -n` `set -o noexec`

Lee comandos pero no los ejecuta. **Detección de errores de sintaxis**

`set -u` `set -o nounset`

Da mensajes de error al intentar utilizar variables no definidas

- **Activar y desactivar modos mediante – y + respectivamente**
- **Pueden establecerse opciones de debugging en parte de un script**

```
set -x          # activa debugging desde aquí
# comandos del script ...
set +x          # desactiva debugging desde aquí
# comandos del script ...
```

- **Control de eficiencia mediante comando `time`**

Ejercicios y Programas

- **`Nivel inicial`**

- Interfaz para el comando `cal`

- Efectuar un backup (`.tar.gz`) de los archivos de su home que no han sido modificados desde la última semana

- Imprimir todos los números primos entre 1 y 500

- Invertir el contenido de un archivo (`tac`)

- **`Nivel medio`**

- Interfaz (exhaustiva) para el comando `cp`

- Interfaz (exhaustiva) para el comando `find`

- Parseo del archivo `/etc/passw` y desplegar su contenido de manera más fácil de leer

- Ecuación cuadrática $ax^2+bx+c=0$

Módulo 14

Conclusiones

Cuando NO utilizar shell scripts

- **Tareas de uso intensivo de recursos, especialmente cuando la velocidad es un factor importante** (sorting, hashing)
- **Procedimientos que involucren cálculos matemáticos pesados, en especial los que involucren aritmética de punto flotante**
- **Portabilidad** (utilizar C en su lugar)
- **Aplicaciones complejas donde se requiere programación estructurada** (typechecking, prototipos de funciones)
- **Aplicaciones muy críticas**
- **Necesidad de operaciones de archivo intensivas**
- **Necesidad de arrays multi-dimensionales**
- **Necesidad de estructuras de datos como listas enlazadas o árboles**
- **Necesidad de generar o manipular GUIs**
- **Necesidad de acceso directo a hardware**
- **Necesidad de realizar E/S a través de puertos o sockets**
- **Aplicaciones de código-cerrado** (Los shell scripts son necesariamente Open Source)