Shell Programming

Put distinctive simple tools together to accomplish your goal…

Outline

- Variable pre-operations
- args, argc in Shell Scripts
- Arithmetic and Logics (Arithmetic is described previously!)
  - Test commands
- Control Structures: if-else, switch-case, for/while loops
- Input/output: Read from screen or file
- Defining Functions & Parsing Arguments
- Error Handling and Debug tool (sh -x)
- A Shell Script Sample: Failure Detection on Servers

- Appendix: Regular Expression
- Appendix B: sed and awk

Shell variables (1)

❑ Assignment

	Bourne Shell	C Shell
Local variable	my=test	set my=test
Global variable	export my	setenv my test

- Example:
  - $ export PAGER=/usr/bin/less
  - % setenv PAGER /usr/bin/less

  - $ current_month=`date +%m`
  - % set current_month =`date +%m`

Shell variables (2)

- Declaration is needed!

There are two ways to call variable…

"${var}"… why? I

- Access
  - % echo "$PAGER"
  - % echo "${PAGER}"
- Use {} to avoid ambiguity
  - % temp_name="haha"
  - % temp="hehe"
  - % echo $temp
    - hehe
  - % echo $temp_name
    - haha
  - % echo ${temp}_name
    - hehe_name
  - % echo ${temp_name}
    - haha

More clear…

Shell variable operator (1)

"${var}"… why? II: value assignment

※ BadCond == !GoodCond

BadCond : var is not set or the value is null
GoodCond : var is set and is not null

operator	description
${var:=value}	If !GoodCond, use the value and assign to var
${var:+value}	If GoodCond, use value instead else null value is used but not assign to var
${var:-value}	If !GoodCond, use the value but not assign to var
${var:?value}	If !GoodCond, **print** value and shell exits

Print → stderr

The command stops immediately

"Parameter Expansion" in sh(1)

Shell variable operator (2)

Samples

❑Ex:

```
#!/bin/sh

var1="haha"
echo "01" ${var1:+"hehe"}
echo "02" ${var1}
echo "03" ${var2:+"hehe"}
echo "04" ${var2}
echo "05" ${var1:="hehehe"}
echo "06" ${var1}
echo "07" ${var2:="hehehe"}
echo "08" ${var2}
echo "09" ${var1:-"he"}
echo "10" ${var1}
echo "11" ${var3:-"he"}
echo "12" ${var3}
echo "13" ${var1:?"hoho"}
echo "14" ${var1}
echo "15" ${var3:?"hoho"}
echo "16" ${var3}
```

❑Result:

```
01 hehe
02 haha
03
04
05 haha
06 haha
07 hehehe
08 hehehe
09 haha
10 haha
11 he
12
13 haha
14 haha
hoho
16
```

Shell variable operator (3)

operator	description
${#var}	String length
${var#pattern}	Remove the smallest prefix
${var##pattern}	Remove the largest prefix
${var%pattern}	Remove the smallest suffix
${var%%pattern}	Remove the largest suffix

These operators do not change var. value…

```
#!/bin/sh

var="Nothing happened end closing end"

echo ${#var}
echo ${var#*ing}
echo ${var##*ing}
echo ${var%end*}
echo ${var%%end*}
```

Results:

```
32
happened end closing end
end
Nothing happened end closing
Nothing happened
```

Predefined shell variables

Similar to C program's "Int main(argc, args)" – **arguments of program**, e.g. ls –a ~

- Environment Variables
- Other useful variables:

sh	csh	description
$#	$#	Number of positional arguments
$0	$0	Command name
$1, $2, ..	$1, $2, .. $argv[n]	Positional arguments
$*	$*, $argv[*]	List of positional arguments (useful in for loop)
$?	$?	Return code from last command
$$	$$	Process number of current command (pid)
$!	$!	Process number of last background command

Usage of $* and $@

- The difference between $* and $@
  - $* : all arguments are formed into a long string
  - $@ : all arguments are formed into separated strings

- Examples: test.sh

```
for i in "$*" ;  do
    echo $i
done

% test.sh 1 2 3
1 2 3
```

```
for i in "$@" ;  do
    echo $i
done

% test.sh 1 2 3
1
2
3
```

test command

Checking things for us… e.g. file status, statements

- test(1)
  - test, [→ two condition evaluation utility
  - test expression
  - [expression]
  - Test for: file, string, number
- Test and return 0 (true) or 1 (false) in $?
  - % test –e News ; echo $? → $? To obtain the return code
    - If there exist the file named “News”
  - % test "haha" = "hehe" ; echo $?
    - Whether “haha” **equal** “hehe”
  - % test 10 -eq 11 ; echo $?
    - Whether 10 **equal** 11

Details on the capability of test command – File test

- -e file
  - True if file exists (regardless of type)
- -s file
  - True if file exists and has a size greater than zero
- -b file
  - True if file exists and is a block special file
- -c file
  - True if file exists and is a character special file
- -d file
  - True if file exists and is a directory
- -f file
  - True if file exists and is a regular file
- -p file
  - True if file is a named pipe (FIFO)
- -L file
  - True if file exists and is a symbolic link
- -S file
  - True if file exists and is a socket
- -r file
  - True if file exists and is readable
- -w file
  - True if file exists and is writable
- -x file
  - True if file exists and is executable
- -u file
  - True if file exists and its set user ID flag is set
- -g file
  - True if file exists and its set group ID flag is set
- -k file
  - True if file exists and its sticky bit is set
- -O file
  - True if file exists and its owner matches the effective user id of this process
- -G file
  - True if file exists and its group matches the effective group id of this process
- file1 -nt file2
  - True if file1 exists and is newer than file2
- file1 -ot file2
  - True if file1 exists and is older than file2
- file1 -ef file2
  - True if file1 and file2 exist and refer to the same file

Hard links to same file..

Details on the capability of test command – String test

- -z string
  - True if the length of string is zero
- -n string
  - True if the length of string is nonzero
- string
  - True if string is not the null string
- s1 = s2
  - True if the strings s1 and s2 are identical
- s1 != s2
  - True if the strings s1 and s2 are not identical
- s1 < s2
  - True if string s1 comes before s2 based on the binary value of their characters
- s1 > s2
  - True if string s1 comes after s2 based on the binary value of their characters

Details on the capability of test command – Number test

- n1 -eq n2 ==, !=, >, <, >=, <= fashion does not apply here…
  - True if the integers n1 and n2 are algebraically equal
- n1 -ne n2
  - True if the integers n1 and n2 are not algebraically equal
- n1 -gt n2
  - True if the integer n1 is algebraically greater than the integer n2
- n1 -ge n2
  - True if the integer n1 is algebraically greater than or equal to the integer n2
- n1 -lt n2
  - True if the integer n1 is algebraically less than the integer n2
- n1 -le n2
  - True if the integer n1 is algebraically less than or equal to the integer n2

test command – combination

- ! expression
  - True if expression is false.
- expression1 -a expression2
  - True if both expression1 and expression2 are true.
- expression1 -o expression2
  - True if either expression1 or expression2 are true.
  - The -a operator has higher precedence than the -o operator.
- (expression)
  - True if expression is true

test command – short format (when used in sh and csh)

- test command short format using [] or ()
  - % test "haha" = "hehe" ; echo $?
  - “Logical, arithmetical and comparison operators” in tcsh(1)

```
if test “haha” = “hehe” ; then
        echo “haha equals hehe”
else
        echo “haha do not equal hehe”
fi
```


```
if [ “haha” = “hehe” ] ; then
        echo “haha equals hehe”
else
        echo “haha doesn’t equal hehe”
fi
```

```
if ( “haha” == “hehe” )  then
        echo “haha equals hehe”
else
        echo “haha doesn’t equal hehe”
endif
```

expr command

Similar to test cmd. but behaves differently…
DO "experiment" on an expression….!!

- Evaluate arguments and return 0 (true) or 1 (false) in $?
- Operators: +, -, *, /, %, =, !=, <, <=, >, >= → Different from test…
- Example:

```
$ a=10
$ a=`expr $a + 10` ; echo $a

% set a=10
% set a=`expr $a + 10`; echo $a
```

<Short format>

- "Arithmetic Expansion" in sh(1)
  - $((expression))
- "Builtin commands" @ in tcsh(1)
  - % @ a = $a + 10 ; echo $a

```
% a=10                    \* to escape for *…
% a=`expr $a \* 2`; echo $a

% expr 4 = 5 ; echo $?
  ➔ 0                     Print immediately
    1
% expr 5 = 5 ; echo $?
  ➔ 1                     But return reversely..
    0
```

if-then-else structure

```
if [ test conditions ] ; then
        command-list
elif
        command-list
else
        command-list
fi
```

```
if ( test conditions ) then
        command-list
else if
        command-list
else
        command-list
endif
```

tell interpreter to load specific shell…

```
#!/bin/sh

a=10
b=12

if [ $a != $b ] ; then
        echo "$a not equal $b"
fi
```

```
#!/bin/tcsh

set a=10
set b=12

if ( $a != $b ) then
        echo "$a not equal $b"
endif
```

switch-case structure (1)

```
case $var in
    value1)
        action1
    ;;
    value2)
        action2
    ;;
    value3|value4)
        action3
    ;;
    *)
        default-action
    ;;
esac
```

OR

default

```
switch ( $var )
    case value1:
        action1
        breaksw
    case value2:
        action2
        breaksw
    case value3:
    case value4:
        action3
        breaksw
    default:
        default-action
        breaksw
endsw
```

switch-case structure (2)

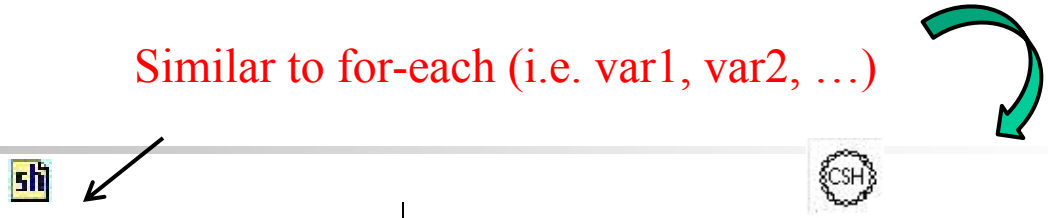
- Example

```
case $# in
    0)
        echo  “Enter file name:”
        read argument1
        ;;
    1)
        argument1=$1
        ;;
    *)
        echo  “[Usage] comm file”
esac
```

```
switch ($#)
    case 0:
        echo  “Enter file name:”
        # read argument1
        breaksw
    case 1:
        argument=$1
        breaksw
    default:
        echo  “[Usage] comm file”
endsw
```

For loop

Similar to for-each (i.e. var1, var2, …)



for var in var1 var2 ...
do
    action
done

```
for dir in bin doc src
do
    cd $dir
    for file in *
    do
        echo $file
    done
    cd ..
done
```

A List

foreach var (var1 var2 ...)
    action
end

```
foreach dir ( bin doc src )
    cd $dir
    foreach file ( * )
        echo $file
    end
    cd ..
end
```

While loop

```
while [...]
do
        action
done
```

```
month=1
while [ ${month} -le 12 ]
do
    echo $month
    month=`expr $month + 1`
done
```

print from 1 to 12…


```
while (...)
   action
end
```

```
set month=1
while ( ${month} <= 12 )
    echo $month
    @ month += 1
end
```

Until loop

like do-while

until [...]

do

action

done

```
month=1
until [ ${month} -gt 12 ]
do
    echo $month
    month=`expr $month + 1`
done
```

Read from stdin

like “readline”, “scanf”…

Usage: read var

```
#!/bin/sh

echo "hello! How are you ?“

read line

if [ "$line" = "fine, thank you" ] ; then
        echo "right answer"
else
        echo "wrong answer, pig head"
fi
```

```
#!/bin/tcsh

echo "hello! How are you ?"

set line=$<

if ( "$line" == "fine, thank you" ) then
        echo "right answer"
else
        echo "wrong answer, pig head"
endif
```

Read from file

Method1: set file to a file descriptor

```
#!/bin/sh

exec 3< "file"

while read line <&3 ; do
        echo "$line"
done
```

```
#!/bin/sh

while read line
do
        echo "$line"
done < "file"
```

```
#!/bin/tcsh

set lc=1

while ( 1 )
        set line=`sed -n $lc,${lc}p "file"`
        if ( "$line" == "" ) then
                break
        endif

        echo $line
        @ lc ++
end
```

Need a “line counter”.. read lines one-by-one until “” line is met

plus one during each iteration

Method2: read line one by one, set input file in the end of while loop

Shell functions (1)

- Define function

```
function_name ( ) {
    command_list
}
```

```
dir ( ) {
        ls –l | less
}
```

Also -- when only one line

```
dir ( )  ls –l | less
```

- Removing function definition

```
unset function_name
```

- Function execution

```
function_name
```

- Function definition is local to the current shell

※ Define the function before first use…

Shell functions (2)

example

```
#!/bin/sh

function1 () {
    result=`expr ${a:=0} + ${b:=0}`
}

a=5
b=10

function1

echo $result
```

No main function, no scope problems of C-Style local-global scopes..

Parsing arguments (1)

sh programs, e.g. portmaster (thousands of code)
carefully take a look on its coding style is recommed…

- Use shift and getopt

```
#!/bin/sh
while [ "`echo $1 | cut –c1`" = "-" ] ;
do
   case $1 in
     -a|-b|-c)
        options="${options} $1" ;;
     *)
        echo "$1: invalid argument" ;;
    esac
    shift
done
```

- shift: a b c → b c → c
- "shift n" is also abailiable
- $0 is the command name

How about "ls –ail" …?

Can take a, b, o options
":" means additional argu.

getopt, a program
Transform:
1. -ab → -a –b
2. -k → --k

```
#!/bin/sh
args=`getopt abo: $*`
if [ $? -ne 0 ]; then
    echo "Usage: getopt.sh [-a] [-b] [-o file]"
    exit 2
fi
set -- $args          <-- Set getopt results
for i                 <-- Iterator, a number
do
   case "$i" in       <-- "$i"
     -a|-b)
        echo flag $i set; sflags="${i#-}$sflags";
        shift;;
     -o)
        echo oarg is ""$2""; oarg="$2"; shift;
        shift;;
     --)
        shift; break ;;
    esac
done
echo "Do something about remainder ($*)"
```

Parsing arguments (2)

Too difficult? Try "sh builit-in getopts" …☺

- Use getopts (recommended)

```
#!/bin/sh

while getopts abcf:o op        <-- op: var. name
# The 'f' followed by ':' indicates the option takes an argument
do
   case $op in
      a|b|c)   echo "OPT=ABC";;
      f)    echo  $OPTARG;;  # $OPTARG is the following argument
      o)    echo  "OPT=o";;
      *)    echo  "Deafult";;
   esac
done
shift `expr $OPTIND - 1`    # The index of the first non-option argument
echo "The left arguments $*"
```

- $OPTARG: arguments
- $OPIND: the number of the index of the first non-option arguments

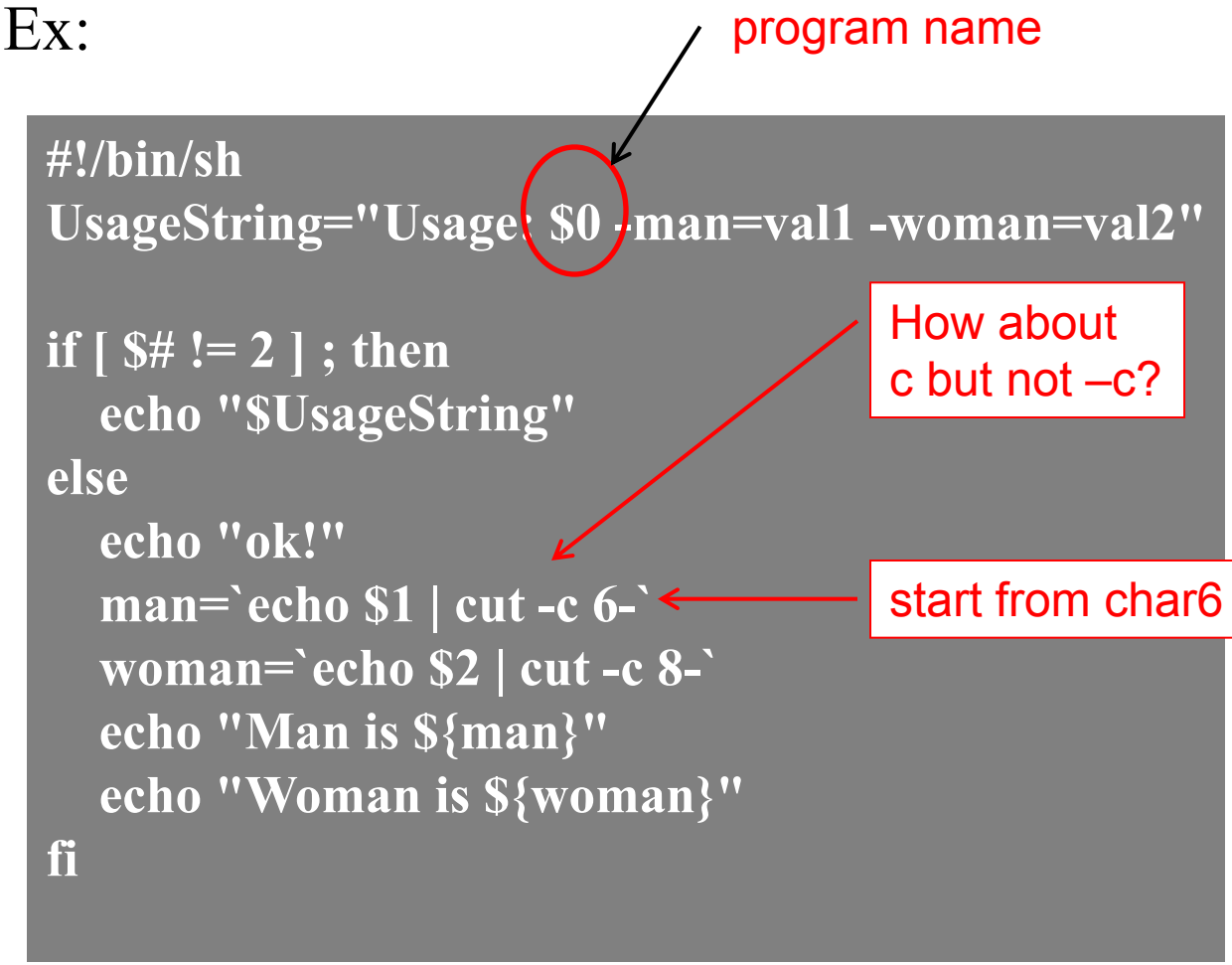
- "Built-in Commands" in sh(1): getopts

Handling Error Conditions

- Internal error ← program crash
  - Caused by some command's failing to perform
    - User-error
      - Invalid input
      - Unmatched shell-script usage
    - Command failure
- External error ← signal from OS
  - By the system telling you that some system-level event has occurred by sending signal

Handling Error Conditions – Internal Error

- Ex:



→ Handling the errors yourself…

Handling Error Conditions – External Error (1)

- Using trap in Bourne shell
  - trap [command-list] [signal-list]
    - Perform command-list when receiving any signal in signal-list

Usag: trap "[commands]" list of signals looking for…

```
trap "rm tmp*; exit0" 1 2 3 14 15
```

```
trap "" 1 2 3
```

Ignore signal 1 2 3

Handling Error Conditions – External Error (2)

#	Name	Description	Default	Catch	Block	Dump core
1	SIGHUP	Hangup	Terminate	☑	☑	⊘
2	SIGINT	Interrupt (^C)	Terminate	☑	☑	⊘
3	SIGQUIT	Quit	Terminate	☑	☑	☑
9	SIGKILL	Kill	Terminate	⊘	⊘	⊘
10	SIGBUS	Bus error	Terminate	☑	☑	☑
11	SIGSEGV	Segmentation fault	Terminate	☑	☑	☑
15	SIGTERM	Soft. termination	Terminate	☑	☑	⊘
17	SIGSTOP	Stop	Stop	⊘	⊘	⊘
18	SIGTSTP	Stop from tty (^Z)	Stop	☑	☑	⊘
19	SIGCONT	Continue after stop	Ignore	☑	⊘	⊘

Handling Error Conditions – External Error (3)

- Using onintr in C shell
  - onintr label
    - Transfer control to label when an interrupt (CTRL-C) occurs
  - onintr -
    - Disable interrupt
  - onintr
    - Restore the default action

Looking for the occurrence of interrupt (CTRL-C)

```
onitr catch
…
Do something in here
…
exit 0

catch:
        set nonomatch
        rm temp*
        exit 1
```

Debugging Shell Script

Debug tools in sh…

❑Ex:

```
#!/bin/sh -x

var1="haha"
echo "01" ${var1:+"hehe"}
echo "02" ${var1}
echo "03" ${var2:+"hehe"}
echo "04" ${var2}
echo "05" ${var1:="hehehe"}
echo "06" ${var1}
echo "07" ${var2:="hehehe"}
echo "08" ${var2}
echo "09" ${var1:-"he"}
echo "10" ${var1}
echo "11" ${var3:-"he"}
echo "12" ${var3}
echo "13" ${var1:?"hoho"}
echo "14" ${var1}
echo "15" ${var3:?"hoho"}
echo "16" ${var3}
```

Debug mode

❑Result:

```
+ var1=haha
+ echo 01 hehe
01 hehe
+ echo 02 haha
02 haha
+ echo 03
03
+ echo 04
04
+ echo 05 haha
05 haha
+ echo 06 haha
06 haha
+ echo 07 hehehe
07 hehehe
+ echo 08 hehehe
08 hehehe
+ echo 09 haha
09 haha
+ echo 10 haha
10 haha
+ echo 11 he
11 he
+ echo 12
12
+ echo 13 haha
13 haha
+ echo 14 haha
14 haha
hoho
```

Debug msgs.
print out the
substitution results…

Shell Script Examples

檢查某一台機器是否當掉 (1)

- Useful details

Ping three times…

- /sbin/ping –c 3 bsd1.cs.nctu.edu.tw

```
PING bsd1.cs.nctu.edu.tw (140.113.235.131): 56 data bytes
64 bytes from 140.113.235.131: icmp_seq=0 ttl=60 time=0.472 ms
64 bytes from 140.113.235.131: icmp_seq=1 ttl=60 time=0.473 ms
64 bytes from 140.113.235.131: icmp_seq=2 ttl=60 time=0.361 ms

--- bsd1.cs.nctu.edu.tw ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.361/0.435/0.473/0.053 ms
```

Ping statistics

檢查某一台機器是否當掉 (2)

```
#!/bin/sh
# [Usage] isAlive.sh ccbsd1

Usage="[Usage] $0 host"
temp="$1.ping"
Admin="liuyh"
count="20"

if [ $# != 1 ] ; then
  echo $Usage
else
  /sbin/ping -c ${count:=10} $1 | /usr/bin/grep 'transmitted' > $temp
  Lost=`awk –F" " '{print $7}' $temp | awk –F"%" '{print $1}' `

  if [ ${Lost:=0} -ge 50 ] ; then
    mail –s  "$1 failed" $Admin < $temp
  fi
  /bin/rm $temp
fi
```

default 10 times

Grep “tran…”

wrtie to the temp file

- awk on $temp using space as delimeter
- How many % packet loss?

Mail and del. $temp

Appendix A: Regular Expression

matching string according to rules…

Regular Expression (1)

- Informal definition
  - Basis:
    - A single character "a" is a R.E.
  - Hypothesis
    - If r and s are R.E.
  - Inductive
    - Union: r + s is R.E
      - Ex: a + b
    - Concatenation: rs is R.E.
      - Ex: ab
    - Kleene closure: r* is R.E.
      - Ex: a*
- Example:
  - (1+2+3+4+5+6+7+8+9) (1+2+3+4+5+6+7+8+9)*
  - Letter: (A + B + C + … + Z + a + b + c + … + z)
  - Digit: (0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9)

Regular Expression (2)

❑ Pattern-matching

- Contain letters, number and special operators

operator	Description
.	Match any single character
[]	Match any character found in []
[^]	Match any character not found in []
^	Match following R.E. only if occurs at start of a line
$	Match following R.E. only if occurs at end of a line
*	Match zero or more occurrence of preceding R.E.
?	Match zero or one occurrence of preceding R.E.
+	Match one or more occurrence of preceding R.E.
{m,n}	Number of times of preceding R.E. At least m times and at most n times
{m,}	Number of times of preceding R.E. At least m times.
{m}	Number of times of preceding R.E. Exactly m times.
\	Escape character

Regular Expression (3)

- Example:
  - **r.n**
    - **Any 3-character string that start with r and end with n**
      - **r1n, rxn, r&n will match**
      - **r1xn, axn will not match**
  - **..Z..**
    - **Any 5-character strings that have Z as 3rd character**
      - **aeZoo, 12Zos will match**
      - **aeooZ, aeZooa will not match**
  - **r[a-z]n**
    - **Any 3-character strings that start with r and end with n and the 2nd character is a alphabet**
      - **rxn will match**
      - **r1n, r&n will not match**
  - **[A-Za-z][0-9]**
    - **Any 2-character strings that 1st character is a alphabet and 2nd is a number**
      - **A2 will match**
      - **2c, 22, A2A will not match**

Regular Expression (4)

- **^Windy**
  - **Any string starts with Windy**
    - Windy is great ➔ match
    - My Windy is great ➔ not match
- **^..Z..**
  - **Any string ..Z.. and ..Z.. starts in a line**
- **[Ee][Nn][Dd]$**
  - **Any string ends with any combination of "end"**
- **^$**
  - **Match blank line**
- **ZA*P**
  - **"A" can be appeared 0 or more times**
    - **ZP, ZAP, ZAAP, …**
- **ZAA*P**
  - **ZAP, ZAAP, …**
- **[A-Za-z] [A-Za-z]***
  - **String of characters**
- **[+-][1-9] [0-9]***
  - **Integer with a preceding + or -1**

operator	Description
.	Match any single character
[]	Match any character found in []
[^]	Match any character not found in []
^	Match following R.E. only if occurs at start of a line
$	Match following R.E. only if occurs at end of a line
*	Match zero or more occurrence of preceding R.E.
?	Match zero or one occurrence of preceding R.E.
+	Match one or more occurrence of preceding R.E.
{m,n}	Number of times of preceding R.E. At least m times and at most n times
{m,}	Number of times of preceding R.E. At least m times.
{m}	Number of times of preceding R.E. Exactly m times.
\	Escape character

Regular Expression (5)

- **[+-]{0,1}[1-9][0-9]***
  - **Match any legal integer expression**
- **[+-]{0,1}[1-9][0-9]*\.{0,1}[0-9]*** Escape of "."
  - **Match any real or integer decimal**
- **[A-Z]{2}Z[0-9]{2}**
  - **Two capital characters followed by Z followed by two numbers**

- "Shell Patterns" in sh(1)
- "REGULAR EXPRESSIONS" in grep(1)
- …

Appendix B: sed and awk

Details on using sed and awk…

sed – Stream EDitor (1)

- sed(1)
- Syntax sed cmds..
  - sed –e "command" –e "command"… file
  - sed –f script-file file sed script
    - Sed will (1) read the file line by line and (2) do the commands, then (3) output to stdout
    - e.g. sed -e '1,10d' -e 's/yellow/black/g' yel.dat
- Command format
  - [address1[,address2]]function[argument]
    - From address 1 to address 2
    - Do what action
- Address format
  - n ➔ line number
  - /R.E./ ➔ the line that matches R.E

sed – Stream EDitor (2)

- Example of address format
  - sed –e 10d
  - sed –e /man/d
  - sed –e 10,100d
  - sed –e 10,/man/d
    - Delete line from line 10 to the line contain "man"

sed – Stream EDitor
Function: substitution (1)

- substitution
  - Syntax

    replacement in whole file

    [address] s/pattern/replace/flags
  - Flags
    - N: Make the substitution only for the N'th occurrence
    - g: replace all matches
    - p: print the matched and replaced line
    - w: write the matched and replaced line to a file

sed – Stream EDitor
Function: substitution (2)

- Ex:
  - sed –e ‘s/liuyh/LIUYH/2’ file
  - sed –e ‘s/liuyh/LIUYH/g’ file
  - sed –e ‘s/liuyh/LIUYH/p’ file
  - sed –n –e ‘s/liuyh/LIUYH/p’ file
  - sed –e ‘s/liuyh/LIUYH/w wfile’ file

file
I am jon
I am john
I am liuyh
I am liuyh
I am nothing

sed – Stream EDitor
Function: delete

- delete
  - Syntax:

    [address]d
- Ex:
  - sed –e 10d
  - sed –e /man/d
  - sed –e 10,100d
  - sed –e 10,/man/d

sed – Stream EDitor
Function: append, insert, change

- append, insert, change
  - insert → insert before the line
  - change → replace whole line
  - Syntax:

[address]a\	[address]i\	[address]c\
text	text	text

- Ex:
  - sed –f sed.src file

sed.src

```
/liuyh/i \
Meet liuyh, Hello
```

file

```
I am jon
I am john
I am liuyh
I am liuyh
I am nothing
```

Results:

```
I am jon
I am john
Meet liuyh, Hello
I am liuyh
Meet liuyh, Hello
I am liuyh
I am nothing
```

sed – Stream EDitor
Function: transform

- transform One-by-one transformation
  - Syntax:

    [add1,addr2] y/xyz.../abc.../
- Ex:
  - sed –e ‘y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/’ file
    - Lowercase to uppercase

sed – Stream EDitor
Function: print

- print
  - Syntax:

    [addr1, addr2]p
- Ex:
  - sed -n -e '/^liuyh/p'

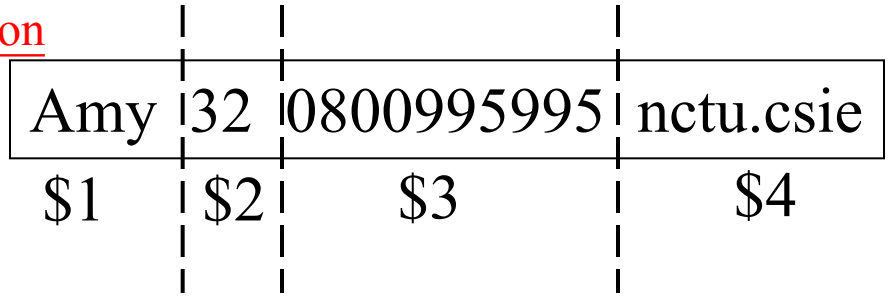
Print out the lines that begins with liuyh

-n: By default, each line of input is echoed to the standard output after all of the commands have been applied to it. The -n option suppresses this behavior.

awk

- awk(1)
- Syntax e.g. -F: space by default
  - awk [-F fs] ['awk_program' | -f program_file] [data_file ……]
    - awk will read the file line by line and evaluate the pattern, then do the action if the test is true
    - Ex:
      - awk '{print "Hello World"}' file
      - awk '/MA/ {print $1}' list

      → sed style line selection



- Program structure
  - 'pattern1 {action1}
    pattern2 {action2}
    ……'

awk – Pattern formats

- pattern formats
  - Relational expression
    - ==, <, <=, >, >=, !=, ~, !~
    - A ~ B means whether A contains substring B

e.g. A = abb; B = bb

  - Regular Expression
    - awk '/[0-9]+/ {print "This is an integer" }'
    - awk '/[A-Za-z]+/ {print "This is a string" }'
    - awk '/^$/ {print "this is a blank line."}'
  - BEGIN
    - It will be true when the awk start to work before reading any data
      - awk ' BEGIN {print "Nice to meet you"}'
  - END → Do before reading file…
    - It will be true when the awk finished processing all data and is ready to exit
      - awk ' END {print "Bye Bye"}' → After reading file…

awk – action format

file
I am jon
I am john
I am liuyh
I am liuyh
I am nothing

- Actions
  - **Print**
  - **Assignment**
  - **if(expression) statement [; else statement2]**
    - **awk '/liuyh/ { if($2 ~ /am/) print $1}' file**
  - **while(expression) statement**
    - **awk 'BEGIN {count=0} /liuyh/ {while (count < 3) {print count;count++}}' file**
    - **awk 'BEGIN {count=0} /liuyh/ {while (count < 3) {print count;count++};count=0}' file**
  - **for (init ; test ; incr) action**
    - **awk '/liuyh/ {for (i=0;i<3;i++) print i}' file**

if $2 exists “am”, print “I”

var usage: no need for “$”

reset count after printing

Achieving same goal impl. using for loop…

awk – built-in variables (1)

- $0, $1, $2, ...
  - Column variables
- NF
  - Number of fields in current line
- NR
  - Number of line processed
- FILENAME
  - the name of the file being processed
- FS
  - Field separator
- OFS
  - Output field separator

awk – built-in variables (2)

- Ex:
  - awk ‘BEGIN {FS=“:”} /liuyh/ {print $3}’ /etc/passwd
    - 1002
  - awk 'BEGIN {FS=":"} /^liuyh/{print $3 $6}' /etc/passwd
    - 1002/home/liuyh
  - awk 'BEGIN {FS=":"} /^liuyh/{print $3 " " $6}' /etc/passwd
    - 1002 /home/liuyh
  - awk 'BEGIN {FS=":" ;OFS="=="} /^liuyh/{print $3 ,$6}' /etc/passwd
    - 1002==/home/liuyh

separator; insert assigned delimiter (space by default)

awk – Reference

- awk(1) is a powerful utility
- Let us man awk
- AWK Tutorial Guide
  - http://lmgtfy.com/?q=awk+tutorial+guide