Linux Programming
쉘 프로그래밍

sisong@ut.ac.kr
한국교통대학교 컴퓨터공학전공
송석일

강의 내용

- 쉘(Shell) 개요
- 파이프(Pipe) & 리디렉션(Redirection)
- 쉘 프로그래밍 및 실행 방법
- 쉘 문법
- 정규 표현식(Regular Expression)
- 실습

쉘 프로그램의 개요

- 쉘 이란
  - 사용자를 리눅스 또는 유닉스 시스템에 이어주는 인터페이스
  - 쉘을 통해 명령어를 입력하면 쉘은 해당 명령을 운영체제가 실행할 수 있도록 전달해주는 역할 수행
  - MS-DOS의 명령어 처리기인 command.com과 비슷
  - 리눅스를 사용 하면서 쓰고 있는 것들의 대부분이 쉘 스크립트로 작성됨
- 쉘의 종류
  - Bash
    - Unix의 Bourne Shell 문법을 모두 만족하며 그외에 더 나은 C Shell의 일부분도 포함을 하고 있는 보다 발전적인 Shell
  - Tcsh
    - bash와 유사
    - bash의 문법으로 짜여져 실행된 Shell 프로그램은 tcsh에서도 실행이 가능
    - tcsh에서 짜여진 Shell 프로그램은 bash에서는 실행되지 않음
    - 막강한 기능들이 있으므로 응용할 수 있으면 상당히 편리
  - 기타 : Korn shell (ksh), C shell (csh)

재지향(Redirection) & 파이프(Pipe)

- 재지향 : ">", ">>"
  - 어떤 프로그램의 출력 정보를 다른 곳(일반적으로 파일)으로 다시 향하게 함
  - ">" : 새로운 파일을 생성, 기존에 같은 이름의 파일이 있었다면 그 파일은 지워지게 됨
  - ">>" : 기존에 같은 이름의 파일이 있다면 그 파일의 마지막 부분에 추가, 같은 이름의 파일이 없다면 새로운 파일 생성
  - 예
    - grep printf hello.c > out.txt
    - grep main hello.c >> out.txt
- 파이프 : |
  - 프로세스간 데이터 통신을 위한 방법 : 하나의 프로그램을 실행시켜서 발생된 표준출력 데이터를 다른 프로그램에 표준입력으로 전달
  - 예
    - grep printf hello.c | wc -l

쉘 프로그래밍 및 실행 방법

- 대화모드
  - 별도의 스크립트 파일을 만들지 않고, 쉘 상에서 직접 실행
  - 대화식으로 수행이 되며 바로 결과 확인 가능
  - 예)

```
[user00@localhost lect]$ for file in *
> do
> if grep -l printf $file
> then
> wc -l $file
> fi
> done
bill.c
6 bill.c
fred.c
6 fred.c
hello.c
7 hello.c
```

쉘 프로그래밍 및 실행 방법

- 스크립트 파일 생성
  - 스크립트를 별도의 파일을 생성하여 저장
  - 스크립트 파일을 실행이 가능한 파일로 모드 변경
- 실습
  - vim을 이용해 왼편의 내용을 first라는 파일에 저장하라

```
#!/bin/sh

#first
#이 파일은 현재 디렉토리의 모든 파일에서
#문자열 printf를 찾고 해당 파일의 이름을
#표준 출력으로 표시한다.

for file in *
do
        if grep -l printf $file
        then
                wc -l $file
        fi
done

exit 0
```

```
[user00@localhost] sh first
[user00@localhost] /bin/sh first
[user00@localhost] chmod +x first
[user00@localhost] ./first
```

쉘 프로그램 문법

- 변수 : 문자열, 숫자, 환경변수, 매개변수
- 조건 : 쉘 부울(Boolean)
- 제어 : if, elif, for, while, until, case
- 리스트
- 함수
- 쉘 내장명령
- 명령 실행
- 히어 도큐먼트(Here Document)

변수

- 쉘 변수의 특징
  - 변수를 사용하기 전에 선언하지 않음
  - 변수의 값은 항상 문자열로 간주됨
  - 대소문자를 구별함
  - $를 변수 앞에 붙여서 변수의 값을 접근
  - read 명령어를 이용하여 사용자입력을 변수에 저장 가능
- 변수와 따옴표
  - 변수의 값이 하나이상의 공백문자(공백, 탭, 줄바꿈)를 포함하면 따옴표를 이용해 표시해야 함
  - " " 와 ' ' 모두 사용가능
    - " " : 변수값에 $변수명 이 포함될때 변수를 저장된 값으로 대체
    - ' ' : $변수명을 그대로 출력
    - ₩ : $의 의미를 제거

실습

아래처럼 프롬프트 상에서 명령어를 수행해 보라.

```
[seokil@database lp]$ greeting=Hello
[seokil@database lp]$ echo $greeting
Hello
[seokil@database lp]$ greeting="Hello Boy"
[seokil@database lp]$ echo $greeting
Hello Boy
[seokil@database lp]$ greeting=7+6
[seokil@database lp]$ echo $greeting
7+6
[seokil@database lp]$ echo greeting
greeting
```

Vim을 이용하여 아래와 같은 내용의 second 파일을 생성하고 실행이 가능하도록 모드를 수정한 후 실행해 보라.

```
#!/bin/sh

myvar="안녕하세요"

echo $myvar
echo "$myvar"
echo '$myvar'
echo "\$myvar"

echo 글자를 입력하세요
read myvar

echo '$myvar' now equals $myvar
exit 0
```

변수

- 환경 변수
  - 쉘 스크립트가 시작할 때 환경으로부터 얻은 값으로 초기화 되는 변수
  - 일반적으로 대문자 사용
  - 계정 사용자에 의해서 추가 가능
  - 기본적인 환경 변수
    - $HOME : 현재 사용자의 홈 디렉토리
    - $PATH : 명령을 검색할 디렉토리들을 콜론(:)으로 구분해 놓은 목록
    - $PS1 : 명령 프롬프트. 보통 $ 지만 다른 값으로 설정 가능
    - $PS2 : 두번째 프롬프트. 추가적인 입력을 받아들이는 프롬프트(일반적으로 >)
    - $IFS : 입력 필드 구분자. 쉘이 입력을 읽을때 단어들을 구분하는 문자 목록
    - $0 : 쉘 스크립트의 이름
    - $# : 전달된 매개변수의 개수
    - $$: 쉘 스크립트의 프로세스 ID. 주로 임시 파일 이름을 유일하게 생성하기 위해 사용됨

변수

- 매개 변수
  - 스크립트를 실행할 때 전달되는 변수
  - 매개변수가 전달되지 않더라도 $# 은 0
  - 매개변수 종류
    - $1, $2, … : 스크립트에 주어진 매개변수
    - $* : 모든 매개변수들의 목록을 하나의 변수로 나타냄. 환경 변수 IFS에 저장된 첫번째 문자로 매개변수 구분
    - $@ : $* 의 확장형. IFS가 비어 있다면 매개 변수들은 함께 나열됨

실습

아래처럼 프롬프트 상에서 명령어를 수행해 보라.

```
[user00@localhost lect]$ IFS=''
[user00@localhost lect]$ set foo bar bam
[user00@localhost lect]$ echo "$@"
foo bar bam
[user00@localhost lect]$ echo "$*"
foobarbam
[user00@localhost lect]$ unset IFS
[user00@localhost lect]$ echo "$*"
foo bar bam
[user00@localhost lect]$
```

Vim을 이용하여 아래와 같은 내용의 vartest 파일을 생성하고 실행이 가능하도록 모드를 수정한 후 실행해 보라.

```
#!/bin/sh

greeting="Hello"
echo $greeting
echo "The program $0 is now running"
echo "The first parameter was $1“
echo "The second parameter was $2"
echo "The third parameter was $3"
echo "The parameter list was $*"
echo "The user's home directory is $HOME"

echo "Please enter a new greeting"
read greeting

echo $greeting
echo "The script is now complete"

exit 0
```

조건

- test, [

```
if test -f fred.c
then
...
fi

if [ -f fred.c ]
then
...
fi
```

  - [는 test 의 축약형
  - [와 확인할 조건 사이에 **공백을 반드시 넣어야** 함

조건

- 문자열 비교
  - [string] - string이 빈 문자열이 아니라면 참
  - [string1 = string2] - 두 문자열이 같다면 참
  - [string1 != string2] - 두 문자열이 다르면 참
  - [-n string] - 문자열이 null(빈 문자열) 이 아니라면 참
  - [-z string] - 문자열이 null(빈 문자열) 이라면 참
- 산술 비교
  - [expr1 -eq expr2] - 두 표현식 값이 같다면 참 ('EQual')
  - [expr1 -ne expr2] - 두 표현식 갑이 같지 않다면 참 ('Not Equal')
  - [expr1 -gt expr2] - expr1 > expr2 이면 참 ('Greater Then')
  - [expr1 -ge expr2] - expr1 >= expr2 이면 참 ('Greater Equal')
  - [expr1 -lt expr2] - expr1 < expr2 이면 참 ('Less Then')
  - [expr1 -le expr2] - expr1 <= expr2 이면 참 ('Less Equal')
  - [! expr] - expr 이 참이면 거짓, 거짓이면 참
  - [expr1 -a expr2] - expr1 AND expr2 의 결과 (둘다 참이면 참, 'And')
  - [expr1 -o expr2] - expr1 OR expr2 의 결과 (둘중 하나만 참이면 참, 'Or')

조건

- 파일 조건
  - [-b FILE] - FILE 이 블럭 디바이스 이면 참
  - [-c FILE] - FILE 이 문자 디바이스 이면 참.
  - [-d FILE] - FILE 이 디렉토리이면 참
  - [-e FILE] - FILE 이 존재하면 참
  - [-f FILE] - FILE 이 존재하고 정규파일이면 참
  - [-g FILE] - FILE 이 set-group-id 파일이면 참
  - [-h FILE] - FILE 이 심볼릭 링크이면 참
  - [-L FILE] - FILE 이 심볼릭 링크이면 참
  - [-k FILE] - FILE 이 Sticky bit 가 셋팅되어 있으면 참
  - [-p FILE] - True if file is a named pipe.
  - [-r FILE] - 현재 사용자가 읽을 수 있는 파일이면 참
  - [-s FILE] - 파일이 비어있지 않으면 참

조건

- 파일 조건
  - [-S FILE] - 소켓 디바이스이면 참
  - [-t FD] - FD 가 열려진 터미널이면 참
  - [-u FILE] - FILE 이 set-user-id 파일이면 참
  - [-w FILE] - 현재 사용자가 쓸 수 있는 파일(writable file) 이면 참
  - [-x FILE] - 현재사용자가 실행할 수 있는 파일(Executable file) 이면 참
  - [-O FILE] - FILE 의 소유자가 현재 사용자이면 참
  - [-G FILE] - FILE 의 그룹이 현재 사용자의 그룹과 같으면 참
  - [FILE1 -nt F - : FILE1이 FILE2 보다 새로운 파일이면 (최근 파일이면) 참
  - [FILE1 -ot F - : FILE1이 FILE2 보다 오래된 파일이면 참
  - [FILE1 -ef F - : FILE1 이 FILE2의 하드링크 파일이면 참

if 문

- 조건에 따라 수행할 부분을 결정하는 제어문

if condition
then
  statements
Fi
	if condition
	then
	  statements
	else
	  statements
	fi
		if condition
		then
		  statements
		elif condition
		then
		  statements
		else
		  statements
		fi

if 문

- 중첩된 if문

```
if condition
then
    if condition
    then
            statements
    else
            statements
    fi
elif condition
then
    statements
else
    statements
fi
```

실습

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

if [ $timeofday = "yes" ]; then
  echo "Good morning"
else
  echo "Good afternoon"
fi

exit 0
```

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

if [ $timeofday = "yes" ]
then
  echo "Good morning"
elif [ $timeofday = "no" ]; then
  echo "Good afternoon"
else
  echo "Sorry, $timeofday not recognized. Enter yes or no"
  exit 1
fi

exit 0
```

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

if [ "$timeofday" = "yes" ]
then
  echo "Good morning"
elif [ "$timeofday" = "no" ]; then
  echo "Good afternoon"
else
  echo "Sorry, $timeofday not recognized. Enter yes or no"
  exit 1
fi

exit 0
```

for 문

- 패턴 리스트 수에 따라 반복을 수행

for variable in values
do
  statements
  …
done
	for variable
	do
	  statements
	  …
	done

while 문

- 조건이 맞는 동안 반복을 수행하는 반복문

```
while condition
do
        statements
        …
done
```

실습

```
#!/bin/sh

for foo in bar fud 43
do
  echo $foo
done

exit 0
```

```
#!/bin/sh

for file in $(ls f*); do
    echo $file
done

exit 0
```

```
#!/bin/sh

echo "Enter password"
read trythis

while [ "$trythis" != "secret" ]; do
  echo "Sorry, try again"
  read trythis
done
exit 0
```

```
#!/bin/sh

foo=1

while [ "$foo" -le 20 ]
do
  echo "Here we go again"
  foo=$(($foo+1))
done

exit 0
```

until 문

- 조건이 맞을 때까지 반복을 수행

```
until condition
do
        statements;
        …
done
```

case 문

- 값에 따라 수행될 부분을 결정하는 제어문

```
case variable in
pattern1)
        statements
        statements
        ;;
pattern2)
        statements
        statements
        ;;
*)
        statements
        statements
        ;;
esac
```

리스트

- 명령어를 일렬로 연결

- AND 리스트
  - statement1 && statement2 && statement3 && …
  - 왼쪽에서 시작해서 각 statement가 false를 반환할 때 까지 실행

- OR 리스트
  - statement1 || statement2 || statement3 || …
  - 왼쪽에서 시작해서 각 statement가 true를 반환할 때 까지 실행

실습

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

case "$timeofday" in
   "yes") echo "Good Morning";;
   "no" ) echo "Good Afternoon";;
   "y" ) echo "Good Morning";;
   "n" ) echo "Good Afternoon";;
   * ) echo "Sorry, answer not recognised";;
esac

exit 0
```

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

case "$timeofday" in
   yes | y | Yes | YES ) echo "Good Morning";;
   n* | N* )         echo "Good Afternoon";;
   * )      echo "Sorry, answer not recognised";;
esac

exit 0
```

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

case "$timeofday" in
   yes | y | Yes | YES )
         echo "Good Morning"
         echo "Up bright and early this morning?"
         ;;
   [nN]* )
         echo "Good Afternoon"
         ;;
   * )
         echo "Sorry, answer not recognised"
         echo "Please answer yes or no"
         exit 1
         ;;
esac

exit 0
```

함수

- 함수 정의

```
function_name () {
   statements
   …
}
```

- 함수 호출

```
function_name
```

실습

```
#!/bin/sh

foo() {
    echo "Function foo is executing"
}

echo "script starting"
foo
echo "script ended"

exit 0
```

함수

- 매개변수 전달
  - function_name 매개변수1, …
  - function_name 함수에서는 매개변수를 $*, $1, $2,… 로 접근
- 값 반환하기
  - return 반환값

실습

```
#!/bin/sh

yes_or_no() {
  echo "Is your name $* ?"
  while true
  do
    echo -n "Enter yes or no: "
    read x
    case "$x" in
      y | yes ) return 0;;
      n | no )  return 1;;
      * )       echo "Answer yes or no";;
    esac
  done
}

echo "Original parameters are $*"

if yes_or_no "$1"
then
  echo "Hi $1, nice name"
else
  echo "Never mind"
fi

exit 0
```

명령 수행

- 쉘 스크립트에서는 두가지 형식의 명령 수행이 가능
  - 외부명령 : 명령 프롬프트에서 수행할 수 있는 명령
  - 내부명령 : 쉘 내부적으로 구현되어 명령 프롬프트에서는 실행되지 않는 명령
- 명령어들

break : 반목문 종료
: : 널 명령, true에 대한 별칭
continue : 반복 계속
echo
eval : 인자의 연산
exec : 현재 쉘을 다른 프로그램으로 대체
exit n : 종료 코드 반환
- 0 : 성공
- 1 ~ 125 : 에러코드
- 126 : 파일 실행 가능
- 127 : 명령 찾을 수 없음
- 128 이상 : 신호 발생

export : 특정 변수를 하위 쉘의 환경 변수로
expr : 표현식 연산
printf : echo 의 향상된 버전
return : 함수의 값 반환
set : 쉘에 매개변수 변수를 설정
shift : 매개 변수위 순서를 이동 $1->$2
trap : 신호를 받았을때 취할 작동 지정
find : 파일을 검색할때 사용
grep : 파일에서 문자열 검색할때 사용

export

export2

```
#!/bin/sh

echo "$foo"
echo "$bar"
```

export1

```
#!/bin/sh

foo="The first meta-syntactic variable"
export bar="The second meta-syntactic variable"

./export2
```

expr, printf

- expr
  - 표현식 수행
    - x=`expr $x + 1`
    - x=$(expr $x+1)
  - expr 로 수행할수 있는 표현식
    - |, &, =, >, >=, <, <=, !=, +, -, *, /, %
- printf
  - printf “형식 문자열” 매개변수1 매개변수2 …
  - 이스케이프 시퀀스
    - ₩₩, ₩a, ₩b, ₩f, ₩n, ₩r, ₩t, ₩v
  - 변환지정자
    - d, c, s, %
  - 예
    - printf “%s₩n” hello
    - printf “%s %d₩t%s” “Hi There” 15 people

find

- 컴퓨터에서 파일 검색
  - find [경로] [옵션] [테스트] [작동]
  - 예) find / -name wish –print
- 주요 옵션
  - -depth : 디렉토리 자체를 살펴보기전에 디렉토리의 내용을 검색
  - -follow : 심볼릭 링크를 따라가서 검색
  - -maxdepths –N : 최대 N 수준의 디렉토리를 검색
  - -mount (or –xdev) : 다른 파일 시스템의 디렉토리는 검색하지 않음
- 주요 테스트
  - -atime N : 파일이 N일 이전에 마지막으로 액세스 된것을 검색
  - -mtime N : N일 이전에 수정된 것 검색
  - -name “패턴” : 경로를 제외한 파일의 이름이 주어진 패턴에 일치하는 것 검색
  - -newer otherfile : otherfile보다 최신인 파일 검색
  - -type C : 파일 형식이 C 인것 (f : 파일 , d : 디렉토리)
  - -user 사용자 이름 : 주어진 사용자가 소유한 파일 검색

find

- 연산자
  - !, -not
  - -a, -and
  - -o -or
- 예
  - find . –newer first –print
  - find . –newer first –type f –print
  - find . ₩(-name "_*" –or –newer first ₩) –type f –print

- 작동
  - -exec : 특정 명령 실행
  - -ok : exec 과 동일, 사용자에게 명령 실행 여부 물어봄
  - -print : 화면에 출력
  - -ls : 현재 파일에 대해서 ls a명령 실행

grep

- 파일에서 문자열 검색
  - grep [옵션] 패턴 [파일]
- 옵션
  - -c : 일치하는 줄을 모두 출력하지 않고, 일치하는 줄의 개수를 출력
  - -E : 확장 표현식 적용
  - -h : 각 출력 줄에 파일이름을 붙이는 작업을 수행하지 않음
  - -i : 대소문자 구별 안함
  - -l : 일치된 줄에 해당하는 파일 명을 나열. 일치된 줄은 출력하지 않음
  - -v : 일치되지 않은 줄만을 선택
- 예
  - grep print second
  - grep –c print first second
  - grep –c –v print first second

명령 실행

- 쉘 스크립트 안에서 명령을 실행하고 그 결과를 쉘 스크립트에서 사용
  - $(명령)
  - ‘명령’
- 예

```
#! /bin/sh

echo 현재디렉토리는 $PWD
echo 현재 사용자는 $(who)

exit 0
```

```
#! /bin/sh

echo 현재디렉토리는 $PWD
whoisthere=$(who)
echo 현재 사용자는 $whoisthere

exit 0
```

산술 확장, 변수 확장

- 산술확장 : $((...))
  - x=`expr $x + 1`
  - x=$(expr $x+1)
  - ➔ x=$(($x+1))
- 변수확장
  - 변수의 일부를 잘라내거나, 변수에 문자를 붙일 수 있음
  - 확장 방법
    - ${변수:-default} : 변수가 널이라면 default 값으로 대체
    - ${#변수} : 변수의 길이
    - ${변수%단어} : 변수 끝에서 단어에 일치하는 변수의 가장 짧은 부분 제거
    - ${변수%%단어} : 변수 끝에서 단어에 일치하는 변수의 가장 긴 부분 제거
    - ${변수#단어} : 변수 처음에서 단어에 일치하는 변수의 가장 짧은 부분 제거
    - ${변수##단어} : 변수 처음에서 단어에 일치하는 변수의 가장 긴 부분 제거

변수확장 예제

```
#!/bin/sh

for i in 1 2
do
  echo user${i}
done

exit 0
```

```
#!/bin/sh

unset foo
echo ${foo:-bar}

foo=fud
echo ${foo:-bar}

foo=/usr/bin/X11/startx
echo ${foo#*/}
echo ${foo##*/}

bar=/usr/local/etc/local/networks
echo ${bar%local*}
echo ${bar%%local*}

exit 0
```

디버깅

- 쉘 스크립트의 디버깅은 상대적으로 매우 쉬움
- 특별히 도움이 될만한 도구는 제공되지 않음
- 일반적인 디버깅 방법
  - 쉘 스크립트의 에러가 발생하면 에러를 포함하고 있는 줄의 번호 출력
  - 에러 원인을 알수 없을 때는 echo 를 이용해서 변수의 내용을 출력하거나 대화모드 쉘 실행 방법을 이용해서 의심 나는 부분을 실행해 봄
- sh 의 실행 옵션을 이용
  - sh –n 스크립트 : 문법 에러만 검사. 명령 실행 안함
  - sh –v 스크립트 : 명령 실행 전에 스크립트 내용 에코
  - sh –x 스크립트 : 명령 실행 후 명령 에코

실습 (과제)

- 현재 디렉토리에서 f로 시작하는 파일이나 디렉토리에 대해서 파일인 경우에는 “파일명 : 일반파일”, 디렉토리인 경우에는 “디렉토리명 : 디렉토리” 를 출력하도록 쉘 프로그램을 작성하시오.

- 현재 디렉토리에 user001, user002, …, user100 의 이름을 갖는 디렉토리를 생성하는 쉘 프로그램을 작성하시오.