제 8 장
쉘 프로그래밍

ACS30021
고급 프로그래밍

나보균 (bkna@kpu.ac.kr)

컴퓨터 공학과
한국산업기술 대학교

학습 목표

- 쉘 스크립트
- 쉘 변수 관리
- 입력 받기
- 연산자
- 제어문
- 함수
- 디버깅

8.1 왜 쉘 프로그래밍을 해야 하나?

- 시스템 관리자
  - ✓ 리눅스 머신이 부팅될 때
    - ➢ 부팅이 되면 시스템 설정 정보들을 읽어 들이고 서비스를 구동하기 위해서 /etc/rc.d에 있는 쉘 스크립트를 실행
- 복잡한 어플리케이션의 프로토타입
  - ✓ 아주 복잡한 어플리케이션을 작성하기 전에 "빠르고 간단한" 프로토타입
    - ➢ 스크립트가 원래 하려고 하던 기능보다 제한된 기능만 제공하고 속도가 느리더라도 이는 프로젝트 개발의 첫 단계에 있어 아주 유용하다. 이렇게 하면 실제로 C, C++, 자바, 펄 등으로 마지막 코딩에 들어가기에 앞서 전체 동작 상태를 점검해 볼 수 있기 때문에 전체 구조상의 중요한 결함을 발견할 수도 있다.
    - ➢ 예,

      인공지능 Deep learning에서 AlphaGo 등 많은 프로그램들이 프로토타입은 python으로 구현하고, 실전용은 C++로 구현
- 복잡한 일들을 작은 단위로 나누어 처리하거나 여러 요소들과 유틸리티를 묶어 처리(고전적인 유닉스 철학)

쉘의 기능

- 명령어 해석기 기능
  - ✓ 사용자와 커널 간 명령을 해석하여 전달
  - ✓ 사용자가 입력한 명령이나 파일에서 읽어 들인 명령을 해석하고 적절한 프로그램을 실행
- 프로그래밍 기능
  - ✓ 프로그래밍 작성가능
  - ✓ 여러 명령을 사용해 반복적 수행 작업을 한 프로그램으로 제작 가능
  - ✓ 쉘 스크립트라고 부름
- 사용자 환경설정 기능
  - ✓ 초기화 파일 기능을 이용해 사용자 환경 설정
  - ✓ 경로 설정, 파일의 기본 권한 설정, 환경변수 설정 등 사용자 별로 사용 환경의 특성을 초기화 파일에 설정 가능
  - ✓ 로그인할 때 초기화 파일이 실행되어 사용자의 초기 환경이 설정됨

쉘 스크립트 개념

- 스크립트?
  - 인터프리터라 불리는 다른 프로그램에 의해 실행되는 프로그램
  - Python, Perl, 자바 스크립트, Tcl/Tk, Make …
  - 파이썬은 스크립트 언어지만 예외적으로 취급
- 쉘 스크립트
  - 쉘이 실행하는 프로그램
  - 유닉스 명령 + 쉘이 제공하는 프로그램 구성 요소
  - 쉘 스크립트 파일 이름은 키워드나 앨리어스, 내장 명령과 같은 이름을 쓰지 않는 것이 바람 직 함
  - 예 : test_script

```
#!/bin/bash
#  My First Script Program

echo I love UNIX !
pwd
```

쉘 스크립트를 쓰면 안 될 때

- 리소스에 민감한 작업들, 특히 속도가 중요한 요소일 때(정렬, 해쉬 등등)
- 강력한 산술 연산 작업들, 특히 임의의 정밀도 연산(arbitrary precision)과 복소수를 써야 할 때(C++ 사용)
- 플랫폼간 이식성이 필요할 때(C 사용)
- 구조적 프로그래밍이 필요한 복잡한 어플리케이션(변수의 타입체크나 함수 프로토타입 등이 필요할 때)
- 업무에 아주 중요하거나 회사의 미래가 걸렸다는 확신이 드는 어플리케이션
- 보안상 중요해서 시스템의 무결성을 보장하기 위해 외부의 침입이나 크래킹, 파괴 등을 막을 필요가 있을 때
- 서로 의존적인 관계에 있는 여러 부분으로 구성된 과제

쉘 스크립트를 쓰면 안 될 때

- 과도한 파일 연산이 필요할 때(Bash는 제한적인 직렬적 파일 접근을 하고 , 특히나 불편하고 불충분한 줄단위 접근만 가능)
- 다차원 배열이 필요할 때
- 링크드 리스트나 트리같은 데이타 구조가 필요할 때
- 그래픽이나 GUI를 생성/변경하는 등의 일이 필요할 때
- 시스템 하드웨어에 직접 접근해야 할 때
- 포트나 소켓 I/O가 필요할 때
- 예전에 쓰던 코드를 사용하는 라이브러리나 인터페이스를 써야 할 필요가 있을 때
- 독점적이고 소스 공개를 안 하는 어플리케이션을 제작 할 때(쉘 스크립트는 필연적으로 오픈 소스이다.)

대체: 펄이나 Tcl, 파이썬 등 스크립팅 언어 또는 C, C++, 자바

기본 사용법 익히기

쉘 ?

- 사용자와 커널 사이의 중간 역할
- 사용자가 입력한 명령을 처리 하고 실행 결과를 알려줌
- 편리한 사용을 위해 다양한 기능 제공

쉘의 종류

- 본쉘(sh), C쉘(csh), 콘쉘(ksh), 배시쉘(bash),…

- 종류 확인하기
  - 프롬프트로 확인 : C쉘은 %, 본쉘/콘쉘/배시쉘은 $
  - 명령으로 확인

프롬프트 모양이 # 인 경우?
시스템관리자(root) 계정임

```
$ echo $SHELL
/usr/bin/bash
$
```

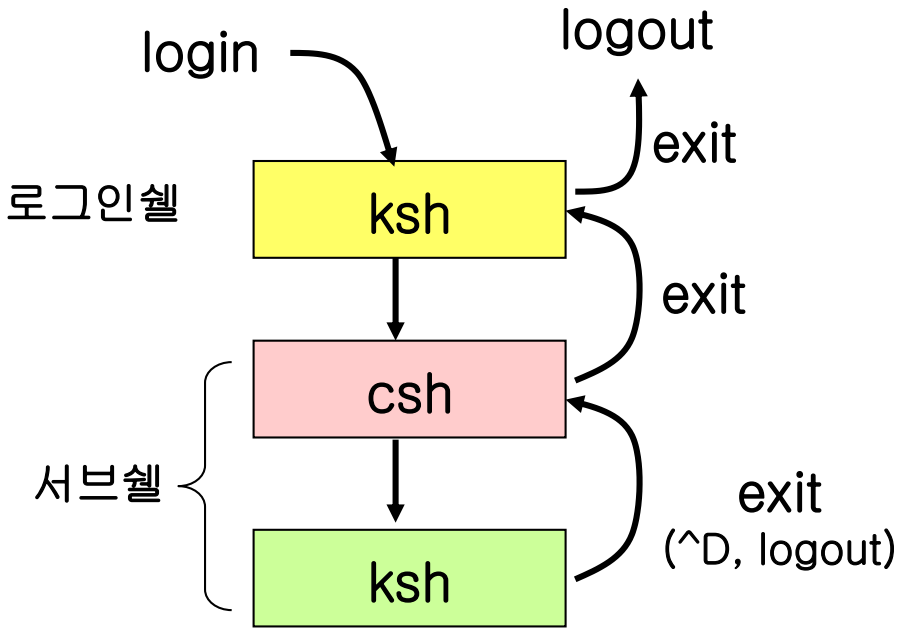
bash 쉘!

쉘 변경하기

로그인 쉘과 서브쉘

- ✓ 로그인 쉘 : 사용자가 로그인한 직후 자동 생성되는 쉘
- ✓ 서브쉘 : 사용자가 직접 실행한 쉘

```
$ csh
% ksh
$ exit
% exit
$
```



환경설정파일 – 배시 쉘

- 시스템 초기화 파일
  - /etc/profile 사용

- 사용자 초기화 파일
  - 홈디렉토리 아래의
  - .bash_profile (이 파일이 없으면 아래 파일 검색)
    .bash_login (이 파일이 없으면 아래 파일 검색)
    .profile (이 파일이 없으면 아래 파일 검색)
    .bash_rc
  - .bashrc
    - .bash_profile에 BASH_ENV=~/.bashrc가 설정되어 있어야 실행
  - .bash_logout
    - 로그아웃할 때 실행

배시 쉘 – 환경설정파일

- 경로설정
  - ✓ 본쉘/콘쉘과 같은 방식으로 설정
  - ✓ .bash_profile에 주로 설정
  - ✓ 예 : PATH=/bin:/usr/bin:/usr/ucb:/usr/local/bin:
- 프롬프트 설정

```
$ PS1="$(uname -n) >"
bkna >
```

- 환경설정파일 적용
  - source명령이나 마침표(.)명령 사용

```
$ source .bash_profile
$ . bash_profile
```

배시 쉘 – 환경설정실습

- .bash_profile에 설정

1) 경로에 /usr/local/bin 추가
2) 프롬프트는 히스토리 번호가 나오도록 수정
3) 저장하고 설정내용 적용토록 실행

- .bashrc에 설정

1) 앨리어스 설정
   - ls가 ls -aF를 실행한다.
   - c는 clear 명령을 실행한다.
   - h는 history 명령을 실행한다.
   - rm은 rm -i를 실행한다.
2) 저장하고 설정내용 적용토록 실행

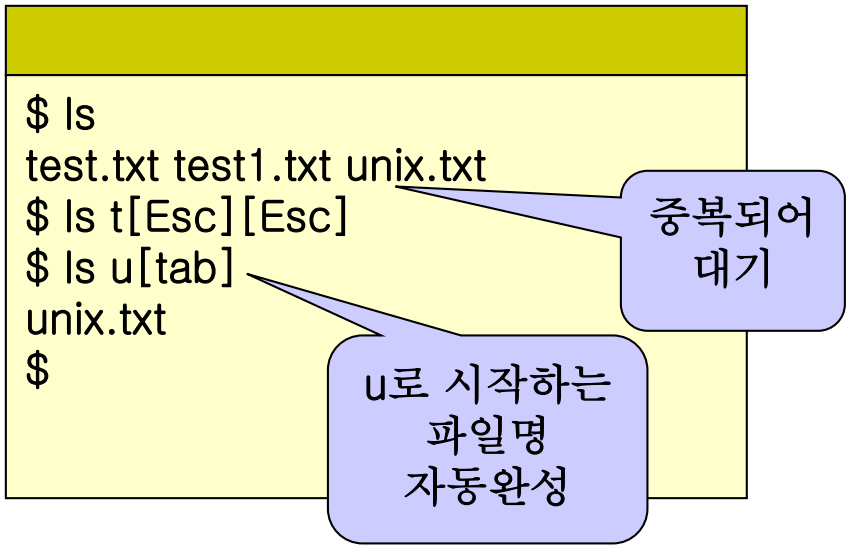
배시 쉘 – 히스토리 기능

- 방향키 기능
  - ✓ 방향키로 히스토리 목록을 불러서 사용

방향키	기능
⬆	히스토리 목록을 위쪽 방향으로 출력 (^p 와 동일)
⬇	히스토리 목록을 아래쪽 방향으로 출력 (^n)
⬅	출력된 히스토리 목록의 왼쪽으로 커서를 이동 (^b)
➡	출력된 히스토리 목록의 오른쪽으로 커서를 이동 (^f)
Backspace	커서위치 한 자 지움 (^d)
	커서 이후 지움 (^k)
	명령어 맨 처음으로 커서 위치 (^a)
	명령어 맨 마지막으로 커서 위치 (^e)

배시 쉘 – 파일명 자동완성기능

- 디렉토리에서 파일명의 앞부분 일부를 입력하고 Esc키를 두 번 입력하거나 탭키를 입력하면 자동으로 나머지 부분을 완성
  - ✓ 이 때 입력한 앞부분이 중복되면 중복된 부분까지만 출력
- 사용법



8.2 기본 쉘 스크립트 환경

- 예 1:

```
# cleanup
# 루트로 실행시키세요.
cd /var/log
cat /dev/null > messages
cat /dev/null > wtmp
echo "로그를 정리했습니다.“
```

- 단순히 콘솔이나 터미널에서 쉽게 실행 시킬 수 있는 명령어들의 조합
- 명령어들을 스크립트 상에서 실행시키는 이유:
  - ✓ 이 명령어들을 반복적으로 입력하지 않아도 된다는 것
  - ✓ 스크립트는 특정한 응용이나 취향에 맞게 수정하고 일반화 가능

쉘 스크립트 파일 내용 – 쉘 명령어

- 쉘이 실행할 수 있는 모든 명령어 사용 가능
- 여러 명령을 반복 수행해야 할 때 파일로 작성하여 실행
- 예 : find_script

```
#!/bin/ksh
#  find_script : /bin, /usr/bin에 있는 쉘 스크립트 검색

cd  /bin
file  *  |  grep  “스크립트”

cd  /usr/bin
file  *  |  grep “스크립트”
```

- 실행 결과:

```
$ find_script
alias:            실행할 수 있는 /bin/ksh 스크립트
appletviewer:  실행할 수 있는 /bin/ksh 스크립트
arch:             실행할 수 있는 /usr/bin/sh 스크립트
basename:      실행할 수 있는 /usr/bin/sh 스크립트
```

쉘 스크립트 파일 내용 – 예제 2:

```
#!/bin/bash
# cleanup, version 2
# 루트로 실행시키세요.
LOG_DIR=/var/log
ROOT_UID=0          # $UID가 0인 유저만이 루트 권한을 갖습니다.
LINES=50            # 기본적으로 저장할 줄 수.
E_XCD=66            # 디렉토리를 바꿀 수 없다?
E_NOTROOT=67        # 루트가 아닐 경우의 종료 에러.

if [ "$UID" -ne "$ROOT_UID" ]
then
  echo "이 스크립트는 루트로 실행시켜야 됩니다."
  exit $E_NOTROOT
fi
if [ -n "$1" ]        # 명령어줄 인자가 존재하는지 테스트(non-empty).
then
  lines=$1
else
  lines=$LINES      # 명령어줄에서 주어지지 않았다면 디폴트값을 씀.
fi
```

예 2:

```
# E_WRONGARGS=65 # 숫자가 아닌 인자.(틀린 인자 포맷)
#
# case "$1" in
# "" ) lines=50;;
# *[!0-9]*) echo "사용법: `basename $0` 정리할파일"; exit $E_WRONGARGS;;
# * ) lines=$1;;
# esac
#
#* 이것을 이해하려면 "루프" 절을 참고하세요.

cd $LOG_DIR
if [ `pwd` != "$LOG_DIR" ] # 혹은 if [ "$PWD" != "LOG_DIR" ]
                           # /var/log 에 있지 않다?
then
  echo "$LOG_DIR 로 옮겨갈 수 없습니다."
  exit $E_XCD
fi # 로그파일이 뒤죽박죽되기 전에 올바른 디렉토리에 있는지 두번 확인함.

# 더 좋은 방법은:
# cd /var/log || {
# echo "필요한 디렉토리로 옮겨갈 수 없습니다." >&2
# exit $E_XCD; # }
```

예 2:

```
tail -$lines messages > mesg.temp # message 로그 파일의 마지막 부분을 저장.
mv mesg.temp messages # 새 로그 파일이 됨.
# cat /dev/null > messages
#* 위의 방법이 더 안전하니까 필요 없음.
cat /dev/null > wtmp # > wtemp 라고 해도 같은 결과.
echo "로그가 정리됐습니다."

exit 0
# 스크립트 종료시에 0을 리턴하면
#+ 쉘에게 성공했다고 알려줌.
```

8.3 스크립트 실행하기

- 스크립트 작성 – scriptname.sh
- 실행
  - sh scriptname 이나,
  - bash scriptname 이라고 입력
  - sh <scriptname 은 스크립트가 표준입력(stdin)에서 읽는 것을 사실상 막기 때문에 권장 안함
  - 더 편한 방법은

    chmod 를 써서 스크립트 자체를 실행 파일로 전환 후 파일 자체 실행
    - chmod 555 scriptname
    - scriptname

쉘 스크립트 종료하기 – exit

- 스크립트의 종료
  - ✓ 파일의 마지막 명령을 실행
  - ✓ exit 실행
- 종료 상태
  - ✓ $? 변수에 저장

```
exit [종료상태]
```

- 예 : test_exit

```
#!/bin/bash
#  test_exit : exit and $? Test Script

exit  20
```

```
$ test_exit
$ echo $?
20
$
```

#! - 매직 넘버"(magic number)

- #! 은 스크립트의 제일 앞에서 이 파일이 어떤 명령어 해석기의 명령어 집합인지를 시스템에게 알려주는 역할
  - ✓ "매직 넘버"(magic number)로서, 실행 가능한 쉘 스크립트라는 것을 나타내는 특별한 표시자
- #! 바로 뒤에 경로명
  - ✓ 스크립트에 들어있는 명령어들을 해석할 프로그램의 위치를 표현
  - ✓ 프로그램이 쉘인지, 프로그램 언어인지, 유틸리티인지를 표현

#! – 매직 넘버

```
#!/bin/sh
#!/bin/bash
#!/usr/bin/perl
#!/usr/bin/tcl
#!/bin/sed -f
#!/usr/awk -f
```

- 기본 쉘인 /bin/sh이나 기본쉘(리눅스에서는 **bash**), 혹은 다른 명령어 해석기를 호출
  - ✓ 유닉스에서 기본 본쉘인 **#!/bin/sh**을 쓰면 다른 머신에 쉽게 이식(port) 가능 (단, Bash 만 가지고 있는 몇몇 기능들은 사용 불가)
  - ✓ 이렇게 작성된 스크립트는 POSIX **sh** 표준을 따름
- "#!" 뒤에 경로는 정확하게
  - ✓ 틀리면 스크립트 실행 시 "Command not found"라는 에러 메시지
- 스크립트에서 내부 쉘 지시자를 안 쓰고 일반 시스템 명령들만 사용하면 #!는 생략 가능

주석

```
echo "이 # 은 주석의 시작이 아닙니다."
echo '이 # 은 주석의 시작이 아닙니다.'
echo 이 # 은 주석의 시작이 아닙니다.        # # 는 특수문자 표시
echo 이 # 은 주석의 시작을 나타냅니다.
echo ${PATH#*:}                              # 매개변수 치환으로, 주석이 아니죠.
echo $(( 2#101011 ))                         # 진법 변환, 주석이 아닙니다.
```

- 패턴 매칭

```
#!/bin/bash
# length.sh
E_NO_ARGS=65
if [ $# -eq 0 ]  # 이 스크립트에서는 명령어줄 인자가 필요합니다.
then
echo "하나 이상의 명령어줄 인자가 필요합니다."
exit $E_NO_ARGS
fi

var01=abcdEFGH28ij
echo "var01 = ${var01}"
echo "var01 의 길이 = ${#var01}"
echo "스크립트로 넘어온 명령어줄 인자 갯수 = ${#@}"
echo "스크립트로 넘어온 명령어줄 인자 갯수 = ${#*}"
exit 0
```

쉘 특수문자

- ✓ 쉘이 특별하게 처리하는 문자들
- ✓ 잘 활용하면 명령을 입력할 때 매우 편리

메타문자	기능	예제
*	임의의 문자열	$ ls h*
?	임의의 한 문자	$ ls dir?
[ ]	대괄호안에 포함된 문자 중 하나	$ ls [a-f]*
~	홈 디렉토리	$ cd ~user1
-	이전 작업 디렉토리	$ cd -
;	명령 연결. 왼쪽부터 차례로 실행	$ date;cal;ls
|	왼쪽 명령의 실행 결과를 오른쪽 명령의 입력으로 전달	$ ls -l /etc | less
‘ ‘	모든 쉘 특수문자 무시	$ print ‘$SHELL’
“ “	$, `, \를 제외한 모든 쉘 문자 무시	$ print “$SHELL”
` `	쉘 명령 수행	$ print `date`
\	특수문자 기능 제거	$ print “\ $SHELL”
>, >> <, <<	입출력 방향 변경	

특수 문자

- ; – 명령어 구분자
  - ✓ 두 개 이상의 명령어를 한 줄에서 같이 쓸 수 있게
  - ✓ echo hello; echo there
- ;; – case 옵션 종료자

```
case "$variable" in
abc)
  echo "$variable = abc" ;;
xyz)
  echo "$variable = xyz" ;;
esac
```

- . – "점"(dot) 명령어.
  - ✓ source 명령어와 동일

특수 문자

- " – 부분 쿼우팅(partial quoting)
  - ✓ **"문자열"** 이라고 하면 쉘이 **문자열**에 들어 있는 거의 대부분의 특수 문자를 해석하지 못함 ($, `(backquote), ₩(이스케이프)를 제외)
- ' – 완전 쿼우팅(full quoting)
  - ✓ **'문자열'** 이라고 하면 쉘이 **문자열**에 들어 있는 모든 특수 문자를 해석하지 못함
  - ✓ "보다 더 강한 형태의 쿼우팅
- , – 콤마 연산자
  - ✓ **콤마 연산자** 는 연속적인 산술 연산을 하려고 할 때 쓰임
  - ✓ 모든 계산이 이루어진 뒤, 마지막에 계산된 결과만 반환
  - ✓ let "t2 = ((a = 9, 15 / 3))" # "a"를 세트하고 "t2"를 계산
- \ – 이스케이프(escape)
  - ✓ **\X**라고 하면 **X** 문자를 "이스케이프" 시키고, **'X'** 라고 인용 시키는 것과 동일한 효과
  - ✓ \는 "나 '이 문자 그대로 해석되도록 인용 할 때 쓰일 수도 있음

특수 문자

- / – **파일명 경로 구분자**
  - 파일명에 등장하는 각 요소들을 구분
  - /home/bozo/projects/Makefile
  - 나누기 산술 연산자
- ` – **명령어 치환(command substitution)**
  - **`명령어`** 라고 하면 **명령어**의 결과를 변수값으로 설정
  - echo It is now `date` => It is now Thu Nov 10 01:18:27 GMT 2011
- : – **널 명령어(null command)**
  - 쉘의 "NOP"(*no op*, 아무 동작도 않함)
- !
  - **테스트나 종료 상태의 의미를 반대나 부정**
  - 예를 들어, "equal"(=)을 "not-equal" (!=)로 해석
  - 다른 상황에서는 간접 변수 참조의 의미로도 쓰임

특수 문자

- *** – 와일드 카드**
  - ✓ * 문자는 정규 표현식에서 0개 이상의 문자를 표현
  - ✓ 이중 별표, **, 는 수학의 누승(累乘, exponentiation) 연산자
- **? – 와일드 카드(하나의 문자)**
  - ✓ ? 문자는 확장 정규 표현식에서 한 문자를 나타내는 것과 마찬가지로 글로빙(globbing)에서 파일명 확장을 나타내는 한 문자짜리 "와일드 카드"의 역할을 합니다.
  - ✓ ?은 이중 소괄호에서 C 스타일의 삼중 연산자
- **$ – 변수 치환**

```
var1=5 var2=3skid
echo $var1 # 5
echo $var2 # 3skid
```

  - ✓ $은 정규 표현식에서 줄의 끝을 표현
  - ✓ ${}매개변수 치환.
  - ✓ $*, $@위치(positional) 매개변수.

쉘 특수문자 및 명령 처리

- **인용부호 : 쉘 특수문자의 의미를 없애기 위해 사용**

인용 부호	기능	사용법
작은 따옴표 (‘ ‘)	모든 특수문자들이 해석되는 것을 막음	$ echo ‘$test’ $test
큰 따옴표 (“ “)	변수나 명령의 대체만 허용	$echo “$test” 100
역슬래시 (₩)	단일 문자가 해석되는 것을 막음	$echo \$test $test

- **명령 대체 : 명령 실행 결과를 문자열로 변환**

기호	사용법
백쿼트 (` `)	$ echo `date` Sunday, April 15, 2012 11:05:06 AM KST
$(명령)	$ echo $(date) Sunday, April 15, 2012 11:15:11 AM KST

8.4 배시쉘 환경 설정

- **변수**
  - ✓ 시스템이나 사용자에 의해 사용되는 정보를 저장하기 위한 저장소
  - ✓ 쉘 변수는 관례적으로 대문자를 사용
- **쉘 변수**
  - ✓ 현재의 쉘에서만 사용 가능한 변수
  - ✓ 확인 명령 : set
- **환경 변수**
  - ✓ 모든 쉘에서 사용 가능한 변수
  - ✓ 확인 명령 : env

변수 정의

- 변수 정의
  - 변수명과 문자열 사이에 공백이 있으면 안됨
  - 쉘 변수 : 변수=값
  - 환경변수
    - 변수=값; export 변수명
    - export 변수=값
- 변수 값 확인
  - 변수명 앞에 $를 붙여 표시
  - set, env 명령으로 확인
  - echo $변수명
- 변수 정의 해제
  - unset 변수명

텔넷 cookook.co.kr

```
$ ATEST=test
$ set
ATEST=test
...
```

```
$ BTEST=test
$ export BTEST
$ echo $BTEST
test
$
```

```
$ unset ATEST
$ echo $ATEST
$ set
```

변수 설정방법

- 쉘변수
  - ✓ 현재 쉘에서만 사용하는 변수로 소문자를 사용
  - ✓ '변수명=값'의 형태로 정의
  - ✓ export를 실행하지 않음
- 환경변수의 출력
  - ✓ env나 set명령을 사용
- 변수 설정 해제
  - ✓ unset 명령

```
$ env
_=/usr/bin/env
LANG=ko
_INIT_UTS_RELEASE=5.7
HZ=100
PATH=/usr/bin:
LOGNAME=user1
HOME=/export/home/user1
…


$ unset PATH
$ echo $PATH
```

자료형

- Bash 변수는 타입이 없다(untyped)
- Bash 는 다른 프로그래밍 언어들과는 달리, 변수를 "타입"으로 구분하지 않음
- Bash 변수는 본질적으로 문자열이지만 Bash 가 문맥에 따라서 정수 연산이나 변수를 비교
  - ✓ 결정적 요소는 그 변수값이 숫자로만 되어 있느냐 아니냐

예약된 환경 변수

- HOME
  - ✓ 사용자 홈디렉터리의 절대 경로명
- LOGNAME
  - ✓ 사용자의 로그인 명
- PATH
  - ✓ 자동 검색 경로명
- PS1
  - ✓ 주 프롬프트 기호 설정 변수
- PS2
  - ✓ 부 프롬프트 기호 설정 변수 (명령어가 1줄을 초과하는 경우 다음 줄 처음에 나오는 프롬프트 기호)
- IFS(Internal Field Separator)
  - ✓ 명령어 줄에서 명령어들을 구분하는 구분자 변수. (공백문자, 탭, 리턴)

예약된 환경 변수

- SHELL
  - ✓ 기본 쉘의 경로와 이름을 설정
- TERM
  - ✓ 단말기 유형을 설정
- TZ(Time Zone)
  - ✓ 시간 영역을 설정하는 변수
  - ✓ 한국은 ROK로 설정
- MAIL
  - ✓ 메일을 저장할 경로와 파일 이름을 지정하는 변수
  - ✓ MAILPATH 변수가 설정되면 이 변수는 무시됨

실습:

- 다음의 출력을 보이는 쉘명령어들로 만들어진 쉘스크립트를 작성하라.

  한국산업기술 대학교
  저는 로그인네임
  저의 홈 디렉터리는 /..../....
  현재 시간은 20xx년 xx월 xx일 .....

쉘 변수 사용하기

- 변수값 활용하기

형식	의미
$name	name의 값으로 대체
$(name)	name의 값으로 대치. 변수 이름이 다른 구문과 인접해 있을 때 사용
${name-word}	name이 정의되어 있으면 그 값을, 그렇지 않으면 word값 사용
${name+word}	name이 정의되어 있으면 word 값을 사용, 그렇지 않으면 NULL
${name=word}	name이 정의되어 있으면 그 값을 출력, 정의되지 않았으면 name 변수에 word를 대입하고 그 값 사용
${name?word}	name이 정의되어 있으면 그 값을 사용하고 그렇지 않으면 word **출력 후 종료**

쉘 변수 사용하기

- 사용 예제

```
$ test=test
$ echo $test
test
$ echo ${test}
test
$ echo ${test-word}
test
$ echo ${test1-word}
word
$ echo ${test+word}
 word
$ echo ${test=word}
test
$ echo ${test1=word}
word
$ echo ${test?word}
test
$ echo ${test2:?word}
bash: test2: word
$
```

1) test 변수에 "test" 문자열 저장
2) test 변수값 출력
3) test 변수값 출력
4) test 변수가 정의되어 있으므로 그 값을 출력
5) tset1 변수가 없으므로 문자열 word 출력
6) test 변수가 정의되어 있으므로 word 출력
7) test가 정의되어 있으므로 그 값을 출력
8) test1 변수가 없으므로 word를 그 값으로 저장
9) tset 변수가 정의되어 있으므로 그 값 출력
10) tset2 변수가 없으므로 word 출력 후 스크립트 종료

변수 문자열 처리

- 변수의 값이 문자열 일 때 문자열 내 패턴을 찾아 일부분을 제거하는 방법

표현식	기능
${vairable%pattern}	variable의 뒤부터 패턴과 일치하는 첫번째 부분을 찾아서 제거
${variable%%pattern}	variable 값의 뒤부터 패턴과 일치하는 최대 부분을 찾아서 제거
${variable#pattern}	variable 값의 앞부터 패턴과 일치하는 첫 번째 부분을 찾아서 제거
${variable##pattern}	variable 값의 앞부터 패턴과 일치하는 최대 부분을 찾아서 제거

변수 문자열 처리

- 사용법
  - ✓ %

```
$ path1="/usr/bin/local/bin"
$ echo ${path1%/bin}
/usr/bin/local
$ echo ${path1%%/bin*}
/usr
$
```

  - ✓ #

```
$ path2="/export/home/user1/.profile"
$ echo ${path2#/export/home}
/usr1/.profile
$ echo ${path2##*/}
.profile
$
```

변수 속성 지정 – typeset

- 변수의 속성
  - ✓ 대소문자, 폭, 정렬 방향, 읽기 전용, 정수타입

명령	기능
typeset	모든 변수 출력 (지역, 환경변수)
typeset –x	export 된 모든 변수 출력 (환경변수)
typeset a b c	지역변수 a b c 정의
typeset –r d	변수 d를 읽기 전용으로 지정
typeset –i num	num 변수에는 정수만 저장
typeset –u name	name의 값을 대문자로 변환하여 저장
typeset –l name	name의 값을 소문자로 변환하여 저장
typeset –L# name	name의 값의 처음 #개 문자만 남기고 삭제
typeset –R# name	name 값의 마지막 #개 문자만 남기고 삭제
typeset –z# name	name을 #개 문자로 구성된 문자열을 만들고 끝에 null문자 추가

명령행 인자 처리

- 명령행 인자
  - ✓ 스크립트를 실행할 때 인자로 주어진 값
- 위치 매개 변수
  - ✓ 명령행 인자를 저장하는 스크립트 변수
  - ✓ 인자의 위치에 따라 이름이 정해짐

명령행 인자	의미
$0	쉘 스크립트의 이름
$1 – $9	명령행에 주어진 첫번째부터 9번째까지 인자
$(10)	10번째 인자 ($10은 왜 틀릴까?)
$#	전체 인자 개수
$*	모든 인자
$@	$*과 같은 의미
“$*”	“$1 $2 $3” 하나의 문자열 취급
“$@"	“$1” “$2” “$3” 각각의 독립적 문자열 취급
$?	최근 실행된 명령의 종료값

명령행 인자처리

- 예 : test_position

```
#!/bin/bash
#  test_position: 명령행 인자를 테스트

 echo '$# : ' $#
 echo $1 $2 $3
 echo '$* : ' $*
 echo '$@ : ' $@
 echo '$? : ' $?
```

- 실행 결과:

```
$ test_position  one two  three
$# : 3
one  two  three
$* : one two three
$@ : one two three
$? : 0
$
```

명령행 인자처리 – shift

```
shift  [n]
```

- 위치 매개변수의 항목을 지정한 개수만큼 왼쪽으로 이동
- 개수를 지정하지 않으면 하나씩 이동

- 예 : test_shift

```
#!/bin/bash
#  test_shift: shift 명령 테스트

echo $*
shift
echo $*
shift
echo $*
```

```
$ test_shift  1  2  3  4  5
1 2 3 4 5
2 3 4 5
3 4 5
$
```

8.5 입출력

- 쉘은 항상 기본적으로
  - ✓ 표준입력(stdin, 키보드),
  - ✓ 표준출력(stdout, 스크린),
  - ✓ 표준에러(stderr, 콘솔 스크린에 에러 메세지)

  "파일들"을 열어 놓는다

표준 출력 명령 – /bin/bash

- echo
  - ✓ 쉘 명령과 유틸리티 모두 제공
  - ✓ 지정한 문자열 출력
  - ✓ echo “Hello” # echo 명령 수행 후 다음 행으로 커서 이동
  - ✓ echo –n “Hello” # no newline
- print
  - ✓ 쉘 명령
  - ✓ 옵션이 제공되어 echo 보다 편리
- printf
  - ✓ printf “Hello”
  - ✓ printf “Hello ₩n”
  - ✓ printf “Hello I am studying %s” cybernetics

```
$ echo 'test'
test
$ /usr/bin/echo test
test
$ print 'I love UNIX!!'
I love UNIX!!
$
```

입력 받기 – read

- 쉘 내장 명령으로 터미널이나 파일로부터 입력 처리
- 사용 형식

형식	의미
read x	표준입력에서 한 행을 입력 받아 x에 저장
read first last	표준입력에서 한 행을 입력받아 첫번째 단어를 first에 저장하고 나머지 모두를 last에 저장
read -p prompt	prompt를 출력하고 입력을 기다린다. 입력된 값은 REPLY 변수에 저장
read -p prompt var	prompt를 출력하고 입력을 기다린다. 입력된 값은 var 변수에 저장

입력 받기 – read

- 예 : test_read

```
#!/bin/bash
# 키보드 입력 처리를 테스트 하는 스크립트

echo "Input x : "
read x                                  # 아무 메시지 없이 사용자 입력을 기다림
echo  The value of x is $x              # 사용자가 임의의 값을 입력하면 출력

read –p "Input y : " y                  # Input y :를 출력한 후 입력 기다림
echo x is $x  y is $y                   # x, y 값 출력

echo –n "Please enter your name : "    # Please~ 문을 출력. 줄 안바꿈
read  first last                        # 첫 단어는 first, 나머지는 last에 저장
echo Hi $first                          # first name 만 출력

printf “Put your ID such as %d₩n” $x   # C 언어의 형식으로 표현
```

사용자로부터 입력 받기 – read

- **사용 예 : test_read**

```
#!/bin/bash
# 키보드 입력 처리를 테스트 하는 스크립트

read   x                          # 아무 메시지 없이 사용자 입력을 기다림
echo  “x : $x”                     # 사용자가 임의의 값을 입력하면 출력

read  x   y                       # 첫 단어는 x, 나머지는 y에 저장
echo “x is $x  y is $y”           # x, y 값 출력

read  -p “Input : “               # Input : 을 출력한 후 입력 기다림
echo “input : $REPLY”             # $REPLY에 자동 저장된 입력값 출력
```

실습:

- read 문을 사용하여 파일을 복사하는 간단한 쉘 프로그램을 작성하시오.

8.6 연산자 – 괄멤단산시 관비논삼대 콤

- 프로그램에서 연산 시 자료를 처리하는 방법
- 산술, 비교, 논리, 비트 연산자 제공
- 수치 연산자 사용시 let 또는 (( )) 사용해야 함

연산자	의미	사용예
-	음수 (단항연산)	-5
!	논리 부정(not)	((! x < y))
~	비트 반전 (not)	~y
* / %	곱셈, 나눗셈, 나머지 연산	let y=3 * 5
+ -	덧셈, 뺄셈	let x=x+1
<< >>	비트왼쪽 시프트, 비트오른쪽 시프트	((y = x << 3))
<= >= < > == !=	비교 연산	((x < y))
& ^ |	비트 AND, XOR, OR 연산	let "z = x ^ y"
&& ||	논리 AND, OR	((x<y || x==3))
=	변수값 지정	let z=1
*= /= %= += == <<= >>= &= ^= |=	단축 연산	let z+=1

산술 연산자 (arithmetic operators)

- 연산자는 (( )), let 이나 expr(산술, 관계, 논리, 문자열 연산만 가능) 식에서 사용
  - ✓ echo `expr 5 % 3` # expr 은 반드시 공백이 연산 사이에 필요
  - ✓ ((c = $a * $b)) ; echo $c
  - ✓ let “a = b + c“; echo $a #let a=b+c; echo $a와 동일
    # 공백 여부 주의!
- 누승 연산자
  - ✓ **누승(exponentiation)
  - ✓ Bash 2.02 버전에서 누승 연산자인 "**"가 소개

```
let "z=5**3"
echo "z = $z" # z = 125
```

연산 예문

```
a=5
echo $a
let b = 20      # let 에서는 공백 사용 못함
let “b = 20”   # 공백을 사용하려면 “ “ 를 사용
echo $b
(( c = 30 ))   # 공백 가능
echo $c
a=$c*5         # let, expr, (( ))을 사용 안하면 문자열
echo $a
echo $((7*8))   # 계산 결과 바로 출력
echo $(( ! 2 + 3  * 4 ))   # !2 는 0
echo $(( 2 << 1 ))
echo $(( 3 ^ 5 ))   # XOR 연산
```

1) 5
2) bash: =: 할당하려면 lvalue가 필요
3) 20
4) 30
5) 30*5
6) 56
7) 12
8) 4
9) 6

연산 예문

```
echo 3 + 4 = `expr 3 + 4`  # echo 3 + 4 = $((3 + 4))
x=20
echo “3 x $x = `expr $x ₩* 3`”  # echo “3 x $x = $(($x * 3))”
```

1) 3 + 4 = 7
2) 3 x 20 = 60

숫자 상수(Numerical Constants)

```
#!/bin/bash
# numbers.sh: 숫자 표시법.

# 10진수
let "d = 32"
echo "d = $d"
# 별로 특별한 게 없다.

# 8진수: '0' 다음에 나오는 숫자
let "o = 071"
echo "o = $o"
# 결과는 10진수로 나타난다.

# 16진수: '0x'나 '0X' 다음에 나오는 숫자
let "h = 0x7a"
echo "h = $h"
# 결과는 10진수로 나타난다.
```

숫자 상수

```
# 다른 진법: 진수#숫자
# 진수는 2 와 36 사이가 올 수 있다.
let "b = 32#77"
echo "b = $b"
#
# 이 표기법은 아주 제한된 범위의 숫자(2 - 36)에서만 동작한다.
# ... 10 개의 숫자 + 26 개의 알파벳 문자 = 36.
let "c = 2#47" # 범위 초과 에러:
# numbers.sh:
let: c = 2#47: value too great for base (error token is "2#47")
echo "c = $c"
echo
echo $((36#zz)) $((2#10101010)) $((16#AF16))
exit 0
```

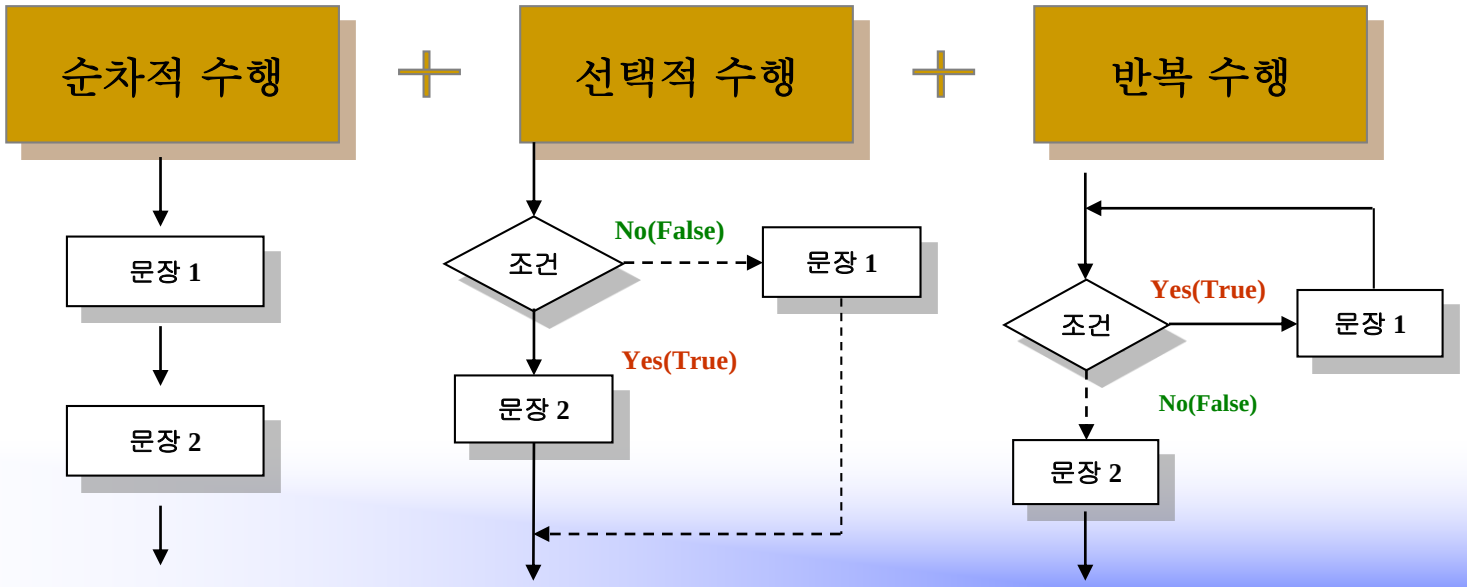
실습:

- 두 개의 숫자를 명령행 인수 $1과 $2에 입력받아 사칙연산을 수행하는 calculator라는 쉘프로그램을 작성하시오.
  - ✓ calculator 200 100 이라고 입력하면

200 + 100 = 300
200 – 100 = 100
200 * 100 = 20000
200 / 100 = 2

8.7 제어문

- 프로그램내의 문장 실행 순서를 제어하는 것
- 선택문
  - ✓ 프로그램 실행문을 조건에 따라 선택적으로 실행
  - ✓ if, select
- 반복문
  - ✓ 프로그램 실행문을 정해진 횟수나 조건에 따라 반복 실행
  - ✓ while , do , for



선택문 – if~then~else

- 주어진 조건의 참, 거짓 여부에 따라 명령 실행

```
if   조건명령
then
     명령
[ else
     명령 ]
fi
```

반드시 “if와 then을 다음 줄에 써야 한다. 아니면 아래처럼;

```
if [  x ]; then
    명령
else
    명령
fi
```

- 예 : test_if

```
#!/bin/bash
#  test_if: if 문을 테스트하는 스크립트

echo –e “Input x: ₩c”; read x  # x 값을 입력 받음
echo –e “Input y: ₩c”; read y  # y 값을 입력 받음

if (( x < y ))
then
   echo “x is less than y $x $y”
else
   echo “y is less than x $x $y”
fi
```

선택문 – if~then~elif~else

- 조건이 실패일 때 새로운 분기 명령 실행

```
if   조건명령1
then
     명령
elif 조건명령2
then
     명령
else
     명령
fi
```

- 예 : test_elif

```
#!/bin/bash
#  test_elif: if-elif 문 테스트

echo "점수를 입력하세요 : "
read score

if (( $score > 90 ))
then
   echo 훌륭합니다.
elif (( $score >= 80 ))
then
   echo 잘 하셨네요
else
   echo 좀 더 열심히 하세요
fi
```

조건식 test – 조건식이 참/거짓 판단

- 형식: test 조건식

  또는 [조건식] # 대괄호와 조건식 사이에 **공백**

- [[ ]] 는 쉘 상에서 [ ]과 동일
  - ✓ **[…]** 말고 **[[…]]** 를 쓰면 많은 논리적 에러들을 막을 수 있다.
  - ✓ 예를 들어 &&, ||, <, > 연산자들은 [ ] 에서 에러, [[ ]] 에서는 잘 동작

```
file=/etc/passwd
if [[ -e $file ]]
then
   echo "비밀번호 파일이 존재합니다."
fi
```

- let 이나 (( )) 문은 산술식에 사용

조건 테스트 – 테스트 문

- 조건 명령에 사용하는 문자열 연산자
- 내장 명령 [[ ]], [ ], 또는 (( )) 사용
  - ✓ 연산자 양쪽에 반드시 공백
  - ✓ 수치 연산자는 (( ))
    - ➢ ((...)) 와 let ... 은 자신이 계산한 산술식이 0이 아닌 값을 가질 경우에 종료 상태 0을 리턴
- 예

```
#!/bin/bash
# test_string: 문자열 테스트 스크립트

echo "Are you OK (y/n) ? "
read answer # answer 변수에 값 저장

if [[ $answer = [Yy]* ]]          # 문자열 비교. y로 시작하는 문자열인가
then
    echo Happy to hear it.         # y로 시작하면
else
    echo That is too bad.          # y로 시작하지 않으면
fi
```

조건 테스트 – 문자열 연산자

- 조건 명령에 사용하는 문자열 연산자
- 내장 명령 [[ ]] 사용

문자열연산자	동작
string = pattern string == pattern	string이 pattern과 일치. = 연산자 양쪽에 공백
string != pattern	string이 pattern과 일치하지 않음
string	string이 널이 아님
-z string	string의 길이가 0
-n string	string의 길이가 0이 아님
-l string	string의 길이

조건 테스트 – 문자열 연산자

- **예 : test_string**

```
#!/bin/bash
# test_string: 문자열 테스트 스크립트

echo -n "Are you OK (y/n) ? "
read  ans                              # ans 변수에 값 저장

if [[ $ans = [Yy]* ]]                  # y로 시작하는 문자열인가
then
   echo Happy to hear it.              # y로 시작하면
else
   echo That is too bad.               # y로 시작하지 않으면
fi
```

조건 테스트 – test 플래그

- 파일 관련 테스트
- 배쉬쉘, 콘쉘 전용 플래그

test 플래그	기능
-a file	파일이 존재
-e file	파일이 존재
-L file	심볼릭 링크 파일
-O file	사용자가 file의 소유자
-F file	사용자의 그룹 ID가 파일의 그룹 ID와 같음
-S file	소켓 파일

조건 테스트 – test 플래그

- 본쉘과 공통 사용 플래그

test 플래그	기능
-r file	읽기 가능
-w file	쓰기 가능
-x file	실행 가능
-f file	일반 파일
-d file	디렉토리 파일
-b file	블록 장치 특수 파일
-c file	문자 장치 특수 파일
-p file	파이프 파일
-u file	setuid 권한 부여 파일
-g file	setgid 권한 부여 파일
-k file	sticky bit 접근 권한 부여 파일
-s file	파일의 크기가 0이 아님

조건 테스트 – test 플래그

- 예 : test_file

```
#!/bin/bash
# test_file: 파일 연산자 테스트

echo "파일 이름을 입력하세요 : "
read file

if [[ ! -a $file ]]
then
    echo 해당 파일이 존재하지 않습니다. 파일 이름을 다시 확인하세요.
elif [[ -f $file ]]
then
    echo 일반 파일입니다.
elif [[ -d $file ]]
then
    echo 디렉토리 파일입니다.
else
    echo 특수 파일입니다.
fi
```

```
$ test_file
파일 이름을 입력하세요 : /dev/null
특수 파일입니다
$ test_file
파일 이름을 입력하세요 : /etc/passwd
일반 파일입니다
$ test_file
파일 이름을 입력하세요 : .
디렉토리 파일입니다 $
```

조건 테스트 – test 플래그

예 : test_file

```
#!/bin/bash
# test_file: 파일 연산자 테스트

echo -n "파일 이름을 입력하세요 : "
read file

if [[ -G $file ]]
then
   echo 그룹ID가 같습니다.
else
   echo 그룹ID가 같지 않습니다.
fi

if [[ ! -a $file ]]
then
   print  파일이 존재하지 않습니다. 파일 이름을 다시 확인하세요.
elif [[ -f $file ]]
then
   print 일반 파일입니다.
elif [[ -d $file ]]
then
   print 디렉토리 파일 입니다.
else
   print 특수 파일입니다.
fi
```

```
$ ./test_file
파일 이름을 입력하세요 : /dev/null
그룹ID가 같지 않습니다.
특수파일입니다
$ ./test_file
파일 이름을 입력하세요 : .
그룹ID가 같습니다.
디렉토리 파일입니다.
$ ./test_file
파일 이름을 입력하세요 : /etc/hosts
그룹ID가 같지 않습니다.
일반 파일입니다.
$
```

조건 테스트 – 정수 비교 연산자(이진)

-eq

같음

if ["$a" -eq "$b"]

-ne

같지 않음

if ["$a" -ne "$b"]

-gt

더 큼

if ["$a" -gt "$b"]

-ge

더 크거나 같음

if ["$a" -ge "$b"]

-lt

더 작음

if ["$a" -lt "$b"]

-le

더 작거나 같음

if ["$a" -le "$b"]

<

더 작음(이중 소괄호에서)

((“$a”< “$b”))

<=

더 작거나 같음(이중 소괄호에서)

((“$a” <= “$b”))

>

더 큼(이중 소괄호에서)

((“$a” > “$b”))

>=

더 크거나 같음(이중 소괄호에서)

((“$a” >= “$b”))

조건 테스트 – 문자열 비교 연산자(이진)

=

같음

if ["$a" = "$b"]

==

같음

if ["$a" == "$b"]

= 와 동의어입니다.

[[$a == z*]] # $a 가 "z"로 시작하면 참 (패턴 매칭)

[[$a == "z*"]] # $a 가 z* 와 같다면 참

[$a == z*] # 파일 globbing이나 # 낱말 조각남이 일어남

["$a" == "z*"] # $a 가 z* 와 같다면 참

!=

같지 않음

if ["$a" != "$b"]

[[...]] 에서 패턴 매칭을 사용

<

아스키 알파벳 순서에서 더 작음

if [["$a" < "$b"]]

if ["$a" ₩< "$b"]

"<" 가 [ ] 에서 쓰일 때는 이스케이프를 시켜야 하는 것에 주의

>

아스키 알파벳 순서에서 더 큼

if [["$a" > "$b"]]

if ["$a" ₩> "$b"]

">" 가 [ ] 에서 쓰일 때는 이스케이프를 시켜야 하는 것에 주의

-z

문자열이 "null"임. 즉, 길이가 0

-n

문자열이 "null"이 아님.

산술 비교와 문자열 비교

```
#!/bin/bash
a=4
b=5
# 여기서 "a"와 "b"는 정수나 문자열 양쪽 모두로 해석될 수 있다. Bash 변수는 타입
# 에 대해 관대하기 때문에 산술 비교와 문자열 비교에는 약간 애매한 부분이 있다.
# Bash 는 숫자로만 이루어진 변수에 대해서 산술 비교도 허용하고 문자열 비교도
# 허용 한다. 주의해서 쓰기 바란다.
if [ "$a" -ne "$b" ]
then
echo "$a 와 $b 는 같지 않습니다."
echo "(산술 비교)"
fi
echo
if [ "$a" != "$b" ]
then
echo "$a 는 $b 와 같지 않습니다."
echo "(문자열 비교)"
fi
# 여기서 "-ne" 와 "!=" 는 둘 다 동작합니다.
echo
exit 0
```

조건 테스트 – 복합 비교 연산자(이진)

-a

논리 and

exp1 -a exp2 는 exp1 과 exp2 **모두** 참일 경우에만 참을 리턴합니다.

-o

논리 or

exp1 -o exp2 는 exp1 이나 exp2 중 **하나라도** 참일 경우에 참을 리턴합니다.

중첩된 if/then 조건 테스트

- if/then를 쓴 조건 테스트는 중첩될 수도 있고, 그 결과만 보면 위에서 살펴본 && 복합 비교 연산자를 쓴 것과 똑같다.

```
if [ condition1 ]
then
   if [ condition2 ]
   then
      do-something # "conditino1"과 "condition2"가 모두 참일 경우에만
   fi
fi
```

실습:

- 시스템 전체에서 쓰이는 xinitrc 파일은 X 서버 실행에 쓰일 수 있다. 이 파일에는 아주 많은 **if/then** 테스트가 나오는데 다음은 그 중 일부분이다.

```
if [ -f $HOME/.Xclients ]; then
    exec $HOME/.Xclients
elif [ -f /etc/X11/xinit/Xclients ]; then
     exec /etc/X11/xinit/Xclients
else
     # 아마 이 부분은 절대 실행되지 않겠지만 혹시 모르니까.
     # (Xclients 에도 역시 안전 장치가 걸려 있다) 전혀 해가 없다.
     xclock -geometry 100x100-5+5 &
     xterm -geometry 80x50-50+150 &
     if [ -f /usr/bin/netscape -a -f /usr/share/doc/HTML/index.html ]; then
        netscape /usr/share/doc/HTML/index.html &
     fi
fi
```

- 위 코드에 나오는 “if-then 조건 테스트"를 설명해 보시오.

선택적 실행문 – case 문

- 주어진 변수의 값에 따라 실행할 명령 따로 지정
- 변수의 값이 value1 이면 value1부터 ;;을 만날 때 까지 명령 실행, 그리고 esac 이후 명령 실행
- 값의 지정에 패턴 표시 특수기호, | (or연산자) 사용 가능
- 일치하는 값이 없으면 기본값인 * 부터 실행

```
case 변수  in
Value1)
    명령 ;;
Value2)
    명령 ;;
*)
    명령 ;;
esac
```

선택문 – case 문

- 예 : test_case

```
#!/bin/bash
# test_case: case 테스트 스크립트

echo "명령을 선택하세요 : "
read  cmd

case $cmd in
[0-9])                # 0부터 9까지 임의의 숫자
   date
   ;;
[aA-C]*)              # 소문자 a, 대문자 A,B,C로 시작하는 임의의 문자열
   pwd
   ;;
"cd" | "CD")          # cd 또는 CD
   echo $HOME
   ;;
*)
   echo Usage : 명령을 선택하세요
   ;;
esac
```

반복문 – for

- 리스트 안의 각 값들에 대해 지정한 명령을 순차 실행

```
for 변수 in `cat list`
do
    명령
done
```

[Tip] `cat list` 대신 $(< list)을 사용하면 외부 파일의 내용을 입력으로 받아서 처리!

- 예 : test_for

```
#!/bin/bash
#  test_for: for 테스트 스크립트

for num in 0 1 2
do
   echo number is $num
done
```

```
$ test_for
number is 0
number is 1
number is 2
$
```

반복문 – for

- 명령행 인자 처리 가능
- 예 : test_for3

```
#!/bin/bash
#  test_for3: 명령행 인자 처리

 for  person  in  $*
 do
    mail  $person <  letter
    echo  ${person}에게 메일을 보냈습니다.
 done
 echo  모든 메일을 보냈습니다.
```

```
$  ./test_for3 user1 user2 user3
user1에게 메일을 보냈습니다.
user2에게 메일을 보냈습니다.
user3에게 메일을 보냈습니다.
----------------------
모든 메일을 보냈습니다.
$
```

반복문 – for

```
#!/bin/bash
# 떠돌이별 목록.
for planet in Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune Pluto
do
   echo $planet
done
echo

# 따옴표로 묶인 전체 '목록'은 한 개의 변수를 만들어 냅니다.
for planet in "Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune Pluto"
do
   echo $planet
done
exit 0
```

실습:

- 오른쪽에 주어진 파일 list.dat을 for 반복문의 외부파일 내용 참조로 활용하여 화면에 출력 하시오.

```
1 2 3 4
```

```
#!/bin/bash
# FILE: forLoop.sh

for num in `cat $1`
do
    echo $num
done

echo '==================='

for num in $(<$1)
do
   echo $num
done
```

실습

- 구구단 프로그램을 아래의 조건에 맞추어 작성하시오.
  - ✓ 프로그램 파일 이름은 multiplicationTable.sh

1. 명령행 인자로 몇 단인지 입력
   a. 예, multiplicationTable_PositionalParam.sh 3 4 5
2. 표준입력 장치로부터 몇 단인지 입력
   a. 반복하여 출력, q를 입력하면 종료
   b. 예, $ multiplicaitonTable_stdio.sh

```
$ 구구단의 몇단을 출력할까요? 5
$ 구구단의 몇단을 출력할까요? 4
$ 구구단의 몇단을 출력할까요? q
$
```

반복문 – while

- 조건 명령이 정상 실행되는 동안 명령 반복

```
while  조건 명령
do
    명령
done
```

```
#무한 반복문
while :
do
    명령
done
```

- 예 : test_while

```
#!/bin/bash
# test_while: while을 이용해 1부터 10까지 합을 구하는 스크립트

count=1
sum=0

while (( count <= 10 ))
do
    (( sum += count ))
    let  count+=1
done

echo 1부터 10까지의 합 : $sum
```

```
$ test _while
1부터 10까지의 합 : 55
$
```

반복문 – until

- 조건 명령이 정상 실행될 때까지 명령 반복

```
until  조건 명령
do
    명령
done
```

- 예 : test_until

```
#!/bin/bash
# test_until: 지정한 사용자가 로그인하면 알리는 스크립트

echo "로그인 이름:"
read person                          # 유저 이름을 person에 저장

until  who | grep $person   # > /dev/null
do
   sleep  5                  # 유저가 접속 중이 아니면 5초 쉼
done
echo “₩007"                  # beep. 삑 소리를 냄
```

break [n]

- 반복 탈출 명령
- for, while, select 등 반복문에 사용
- break n
  - ✓ n번째 레벨로 반복 탈출

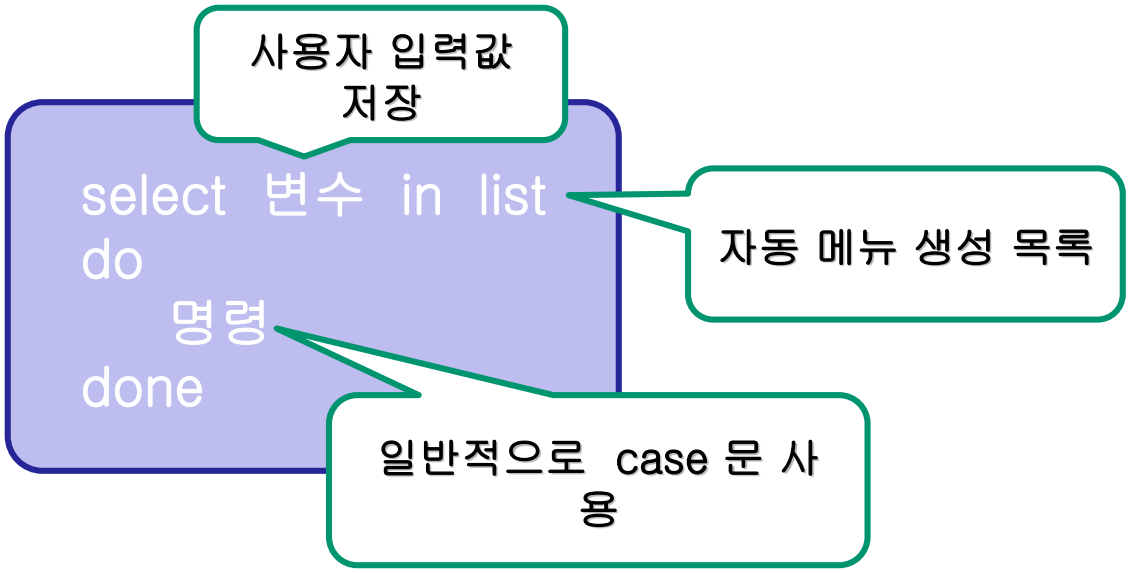
```
#!/bin/bash
# test_select: 사용자 입력에 따라 pwd,date 명령실행

PS3="명령을 입력하세요 : "

select cmd in pwd date quit
do
   case $cmd in
   pwd)
        pwd ;;
   date)
        date;;
   quit)
        break;;
   *)
        echo 잘못 입력하셨습니다. 번호를 선택하세요.
        ;;
   esac
   REPLY=
done
```

반복문 – select

- 메뉴를 생성할 수 있는 반복 실행문
- list에 지정한 항목을 선택 가능한 메뉴로 만들어 화면에 출력
  - ✓ 각 항목에 자동 부여된 번호 선택 가능
  - ✓ 사용자 입력은 select와 in 사이에 지정된 변수에 저장
  - ✓ 보통 case 문과 결합하여 입력 값 처리



반복문 – select

- 예 : test_select

```
#!/bin/bash
# test_select: 사용자 입력에 따라 pwd,date 명령실행.

PS3="명령을 입력하세요 : "

select cmd in pwd date quit
do
   case $cmd in
   pwd)
      pwd ;;
   date)
      date;;
   quit)
      break;;
   *)
      echo “잘못 입력하셨습니다. 번호를 선택하세요.”
      ;;
   esac
   REPLY=
done
```

```
$ test _select
1)  pwd
2)  date
3)  quit
명령을 입력하세요 : 1
/export/home/user1/unix/ch13
1)  pwd
2)  date
3)  quit
4)  명령을 입력하세요 : 3
$
```

반복문 - select

```
#!/bin/bash
PS3='제일 좋아하는 야채를 고르세요: '
# 프롬프트 문자열 세트.
echo
select vegetable in "콩" "당근" "감자" "양파" "순무"
do
   echo
   echo "제일 좋아하는 야채가 $vegetable 이네요."
   echo "깔깔~~"
   echo
   break # 여기에 'break'가 없으면 무한 루프를 돕니다.
done
exit 0
```

continue

- 루프 안에서 사용
- 이후 실행 순서를 무시하고 루프의 처음으로 돌아가는 명령
- 예 : test_cont

```
#!/bin/bash
#   test_cont: continue 테스트

for person in $(< list)    # `cat list`와 동일
do
    if [[ $person == gildong ]] then
       continue
    fi
    mail  $person <  letter
    printf  "${person}에게 메일을 보냈습니다. ₩n"
done
printf "모든 메일을 보냈습니다. ₩n"
```

[Tip] 파일 list에 들어있는 명단 중 gildong을 제외한 모두에게 eMail 전송

continue [n]

```
#!/bin/bash
#  test_cont: continue 테스트

for person in $1 $2 $3
do

  while [[ $person != park ]]
  do
    if [[ $person == kim ]] then
      continue 2      # 1이면 while문으로, 2이면 for문으로
    fi
    echo  ${person}에게 메일을 보냈습니다.
  done

  echo Park에게 메일을 안 보냈습니다.
done

echo 모든 메일을 보냈습니다.
```

break와 continue

- break와 continue 루프 제어 명령어는 다른 프로그래밍 언어들과 정확히 같은 의미의 작동
  - ✓ break 명령어는 자신이 속해 있는 루프를 끝내고
  - ✓ continue는 해당 루프 사이클 내에 남아 있는 나머지 명령어들을 건너 뛰고 다음 단계의 루프를 수행

실습:

```
#!/bin/bash
LIMIT=19
# 상한선
echo
echo "3,11을 제외하고 1부터 20까지 출력."
a=0
while [ $a -le "$LIMIT" ]
do
   a=$(($a+1))
   if [ "$a" -eq 3 ] || [ "$a" -eq 11 ] # 3,11을 제외
   then
      continue # 이번 루프의 나머지 부분을 건너뜀.
   fi
   echo -n "$a "
done
# 독자들을 위한 연습문제:
# 루프가 왜 20까지 찍을까요?
echo; echo
echo 1부터 20까지 출력하지만 그 다음에 무슨 일인가가 일어납니다.
```

```
##########################################################
# 똑같은 루프지만 'continue'를 'break'로 바꿨습니다.
a=0
while [ "$a" -le "$LIMIT" ]
do
   a=$(($a+1))
   if [ "$a" -gt 2 ]
   then
      break # 루프 나머지를 모두 건너뜀.
   fi
   echo -n "$a "
done
echo; echo; echo
exit 0
```

변수 간접 참조

- 어떤 변수값이 다음에 나올 변수의 이름이라고 가정해 보자. 그렇다면 그 어떤 변수로 다음에 나올 변수의 값을 알아낼 수 있을까?
- 예를 들어, *a=letter_of_alphabet* 이고 *letter_of_alphabet=z* 라고 할 때 *a*를 참조하면 *z*가 나올까?
- 이를 **간접 참조(dereference)**라고 부른다.
- 사용 형태

```
#!/bin/bash
#FILE: deref.sh
# test of variable dereference

var2="value”  # var2=value
var1="var2”    # var1=var2
echo ${!var1}  #echo $(!var1)
```

```
$sh deref.sh
value
```

- eval var1=₩$$var2 또는 var1=${!var2}

간접 참조

```
#!/bin/bash
# 변수 간접 참조.
a=letter_of_alphabet
letter_of_alphabet=z
echo
# 직접 참조.
echo "a = $a"

# 간접 참조.
eval a=₩$$a
echo "이제 a = $a"

echo
# 이제 2차 참조(second order reference)를 바꿔보도록 하자.
t=table_cell_3
table_cell_3=24
echo "₩"table_cell_3₩" = $table_cell_3"
echo -n "역참조(dereferenced)된 ₩"t₩" = "; eval echo ₩$$t
# 이 간단한 예제에서는,
# eval t=₩$$t; echo "₩"t₩" = $t"
# 라고 해도 되는데 왜 그럴까?

echo
```

간접 참조

```
t=table_cell_3 NEW_VAL=387
table_cell_3=$NEW_VAL
echo "₩"table_cell_3₩" 의 값을 $NEW_VAL 로 바꿉니다."
echo "₩"table_cell_3₩" 은 이제 $table_cell_3 이고,"
echo -n "역참조된 ₩"t₩" 는 "; eval echo ₩$$t
# "eval" 은 "echo"와 "₩$$t" 두 개의 인자를 받아서 $table_cell_3 과 똑같이 세트해줍니다.
echo

exit 0
```

실습:

- 1부터 100까지의 합을 구하는 쉘 스크립트를 작성하시오.
- 1부터 임의의 수까지 더하는 쉘 스크립트를 작성하시오.

8.8 함수 정의

- 하나의 목적으로 사용되는 명령들의 집합
- 앨리어스와의 차이점
  - ✓ 조건에 따라 처리 가능
  - ✓ 인자 처리 가능
  - ✓ bash 불가
  - ✓ c, tcsh 가능
- 예 : trash

```
$ mkdir  ~/.TRASH
$ function  trash {
>  mv  $*  ~/.TRASH
>  }
$
```

함수 정의

```
함수이름 ()
{
    명령어들
}
```

```
function 함수이름
{
    명령어들
}
```

```
function 함수이름 {
    명령어들
}
```

- 쉘 함수의 정의된 내용은 주기억 장소에 상주
  - ✓ 쉘 프로그램 보다 빠르게 처리
- 쉘함수 실행
  - ✓ 함수 이름만
  - ✓ 괄호 생략
- 쉘 프로그램의 처음에 정의
  - ✓ 정의 후 호출
- 정의 되어 있는 모든 함수 목록으로 출력
  - ✓ typeset –f
- 함수의 본문 영역 표시인 중괄호를 함수 이름 다음에 이어 쓰려면 앞뒤에 공백이 반드시 필요

함수

- 매개변수
  - ✓ $1 – 첫 번째 매개변수
  - ✓ $2 – 두 번째 매개변수
  - ✓ $n – n 번째 매개변수

- 명령행 인자 $1, $2, … 등과 착오 조심

변수의 접근영역

- Scope of Variables
  - ✓ Basically, there is no scoping,
  - ✓ other than the parameters ($1, $2, $@, etc)

```
#!/bin/sh
myfunc()
{
  echo "I was called as : $@"
  x=2
}

### Main script starts here
### File name: scope.sh
echo "Script was called with $@“
x=1
echo "x is $x“
myfunc 1 2 3
echo "x is $x"
```

The script gives the following output:

```
$ scope.sh a b c
  Script was called with a b c
  x is 1
  I was called as : 1 2 3
  x is 2
```

A function will be called in a sub-shell if its output is piped somewhere else – that is, "**myfunc 1 2 3 | tee out.log**" will still say "x is 1" the second time around

함수 확인 및 호출

- 정의된 함수 확인

텔넷 cookook.co.kr

```
$ typeset -f
trash()
{
    mv   $*   ~/.TRASH
}
$
```

- 함수 호출

텔넷 cookook.co.kr

```
$ touch   a   b   c
$ ls
a   b   c
$ trash a   b   c
$ ls
$ ls   ~/.TRASH
a   b   c
$
```

재귀적 사용

```
### factorial.sh
#!/bin/sh
factorial()
{
   if [ "$1" -gt "1" ]; then
     i=`expr $1 - 1`
     j=`factorial $i`
     k=`expr $1 □  * $j`
     echo $k
   else
     echo 1
   fi
}

while :
do
   echo "Enter a number:“
   read x
   factorial $x
done
```

라이브러리 함수 호출

```
### common.lib
# Note no #!/bin/sh as this should not spawn
# an extra shell. It's not the end of the world
# to have one, but clearer not to.
#
STD_MSG="About to rename some files...“

rename()
{
  # expects to be called as: rename .txt .bak
  FROM=$1
  TO=$2

  for i in *$FROM
  do
    j=`basename $i $FROM`
    mv $i ${j}$TO
  done
}
```

```
#!/bin/sh
# function2.sh
. ./common.lib
echo $STD_MSG
rename txt bak
```

```
#!/bin/sh
# function3.sh
. ./common.lib
echo $STD_MSG
rename html html-bak
```

함수 호출

- 정의된 함수 확인

```
$ functions
$ function  trash
{
  mv  $*  ~/.TRASH
}
$
```

- 함수 호출

```
$ touch  a  b  c
$ ls
a  b  c
$ trash a  b  c
$ ls
$ ls  ~/.TRASH
a  b  c
$
```

함수의 종료 – return

- 함수 종료 조건
  - ✓ 함수 본문 안의 마지막 문장 실행
  - ✓ return 문 실행

```
return  [n]
```

- 지정한 값이 함수의 종료값으로 $?에 저장됨

함수의 종료 – return

- 예 : add

```
#!/bin/bash
#   함수 리턴값 테스트

function sum {
   typeset  sum

   (( sum= $1 + $2 ))
   return $sum
}

sum $1 $2
echo $1 + $2 = $?
```

```
$ add  3  4
3 + 4 = 7
$
```

함수의 삭제 – unset

```
unset  -f  함수명
```

- 정의된 함수를 삭제
- 사용법

```
$ unset  -f  trash
$ functions
$
```

실습:

- 쉘 스크립츠 함수 add, subtract, multiply, divide을 작성
- 다음의 방법 1, 2를 각각 수행
  1. 두 수와 연산자를 명령행 인자로 받아
  2. 표준 입력으로 두 수와 연산자를 입력
- 그 연산 결과를 화면에 출력

8.9 trap 명령

- signal and signal handler
- 시스템 내의 약속된 신호를 사용하여 자신이 실행하는 쉘 프로그램에게 특정 상황을 알리는 신호를 전달
- 신호 종류
  - ✓ 0 번: 쉘의 종료
  - ✓ 2번: 프로세스(쉘 프로그램)를 중단하는 인터럽트
  - ✓ 4번: 잘못된 명령어를 사용할 때 제어
  - ✓ 9번: 프로세스 강제 종료(kill)
- 명령의 문법 형식

  trap `명령어` 신호번호
  - ✓ 여러 개의 명령어인 경우 ;(semicolon)으로 분리
  - ✓ 명령어 생략하면 기본 신호(default)로 재 지정
  - ✓ 해당 신호 무시 – 명령어를 “” (NULL)로

trap 명령 사용 예문

- 특정 사용자가 로그인 하였는지 여부 검사 프로그램
- 10초 단위로 검사
- trap 명령을 사용하여 <C-c> 무시
- 프로그램 중단 방법
  - ✓ kill 프로세스번호

```
trap ` “” ` 2
read usrname ? “로그인 여부 검사 사용자명 입력: \c”
while true
do
   if who | grep $usrname > /dev/null
   then
      echo “ $usrname 가 지금 로그인 하였음!”
      break
   fi
   sleep 10
done
```

실습:

- 함수 작성
  - ✓ 앞의 trap 명령을 사용한 특정 로그인명 사용자 검사 프로그램
  - ✓ trap 명령문 생략
- 함수 실행
  - ✓ 함수 이름만 입력
- 함수 해제
  - ✓ $ unset 함수명

8.10 디버깅

- 스크립트 작성 도중 발생한 오류 수정 방법

- 구문 오류
  - ✓ 쉘이 실행도중 구문오류가 발생한 라인번호 출력

- 실행 오류
  - ✓ 오류 메시지 없이 실행이 안되거나 비정상 종료
  - ✓ 오류 수정 방법
    - ➢ bash -x, trap

- 버그 있는 스크립트의 증상을 요약해 보면,
  - ✓ syntax error – 메세지를 내면서 죽는다
  - ✓ 죽지는 않지만 생각했던 대로 동작하지 않는다(로직 에러, logic error).
  - ✓ 죽지도 않고 생각했던 대로 동작하지만, 처리하기 까다로운 부효과(side effect)가 있다(로직 폭탄, logic bomb).

스크립트 디버깅

1. 중요한 부분에 echo 문으로 변수값을 출력
2. 중요한 부분에 **tee** 필터를 걸어서 프로세스나 데이타 흐름을 확인
3. -n -v -x 옵션을 사용
   - ✓ **sh -n scriptname**는 스크립트를 돌리지는 않고 단순히 문법 에러(syntax error)만 확인
     - ➢ 스크립트 안에서 **set -n**이나 **set -o noexec**이라고 해도 같은 동작을 한다. 조심할 점은 이 옵션에 걸리지 않고 그냥 넘어가는 문법 에러도 있다는 것이다.
   - ✓ **sh -v scriptname**는 각 명령어를 실행하기 전에 그 명령어를 echo해 준다. 스크립트 안에서 **set -v**이나 **set -o verbose**라고 해도 같은 동작을 한다.
   - ✓ -n 과 -v 플래그를 같이 쓰면 좋다. **sh -nv scriptname** 이라고 하면 문법 체크를 아주 자세하게 해준다.
   - ✓ **sh -x scriptname**는 각 명령어의 결과를 간단한 형태로 echo시켜 준다. 스크립트 안에서 **set -x**나 **set -o xtrace**라고 해도 똑같다.
   - ✓ 스크립트에 **set -u**나 **set -o nounset**을 넣어두면, 선언 없이 쓰이는 변수들에 대해서 unbound variable 에러 메세지를 출력해 줄 것이다.

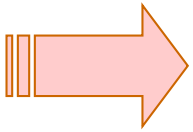
스크립트 디버깅

4. 스크립트의 아주 중요한 지점에서 변수나 조건을 테스트 하기 위해서 "assert" 함수 쓰기
5. exit 잡아채기(trapping at exit).
   - ✓ 스크립트에서 **exit**를 쓰면 프로세스 종료를 의미하는 0번 시그널을 발생하여 자기 자신을 종료시킨다. **exit**를 실행해서(trap) 강제로 변수값을 "출력"하도록 하는 등의 유용한 작업을 할 수 있다. 이렇게 하려면 **trap** 명령어가 스크립트의 첫 번째 명령어여야 한다.

디버깅 : trap 이용

```
trap    명령    시그널
```

- 지정한 시그널이 스크립트로 전달될 때마다 지정한 명령 실행
- 스크립트의 명령이 한 줄씩 실행될 때마다 DEBUG 시그널이 스크립트로 전달됨
- DEBUG 시그널을 받을 때마다 원하는 변수값 출력 가능



스크립트가 실행되는 도중 변수 값 확인

디버깅 : trap 이용

- 예 : test_trap

```
#!/bin/bash
# test_trap: trap 테스트 스크립트

trap ‘print “$LINENO : count=$count “ ’ DEBUG

count=1
sum=0

while (( count <= 10 ))
do
    (( sum += count ))
    let  count+=1
done

echo “1부터 10까지의 합 : $sum”
```

연습문제

1. 시스템 관리자들은 일반적인 작업들을 자동으로 하기 위해서 가끔씩 스크립트를 만들어 쓴다. 이런 스크립트로 하기에 좋은 상황들을 몇 개 나열해 보시오.

2. 시간과 날짜, 현재 로그인해 있는 모든 사용자들, 시스템 업타임(uptime)을 보여주는 스크립트를 만들어 보시오. 그 다음에는 로그 파일에 그 정보들을 저장하도록 해 보시오.

실습:

- 교재 459쪽
- 문제 3, 4, 7, 8

쉘 프로그래밍 참조 사이트

- 고급 Bash 스크립팅 가이드
  - http://wiki.kldp.org/HOWTO/html/Adv-Bash-Scr-HOWTO/why-shell.html
- http://www.arachnoid.com/linux/shell_programming.html
- http://steve-parker.org/sh/sh.shtml
- http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html
- Bash Reference Manual
  - http://www.gnu.org/software/bash/manual/bashref.html

기술동향

- 요즘 트렌드는 간단한 스크립팅은 Bash를 많이 쓰고
- 조금 복잡한 것들은 파이썬을

- Bash 추천 도서:
  - ✓ 예문이 많이 있대요!
  - ✓ https://www.ebooks-it.net/ebook/bash-cookbook

- 도움되는 사이트:
  - ✓ https://wiki.kldp.org/HOWTO/html/Adv-Bash-Scr-HOWTO/