Bourn Shell

변수

- 변수의 생성 및 할당 (Regular Expression)
  - {name=value}+

변수가 존재하면 새로운 값으로 바꾸고, 존재하지 않으면 자동적으로 만들어짐. 공백은 “ ” 로 .

(예)

```
$ Name = gtkim age=18
$ echo $Name is $age
gtkim is 18
$ name= gt kim          .... error   문자열은  따옴표(“ “)로 표시
$ name=“gt kim”
```

변수

- 변수로의 접근
  - $name : name의 값으로 대치
  - ${name} : name의 값으로 대치. 변수 이름의 일부로 번역되는 문자가 숫자가 표현식 다음에 올 때 유용.

(예)

```
$ verb=sing                          ...변수할당
$echo I like  $verbing               ... verbing 변수는 없다
I like
$echo I like ${verb}ing              ...성공
I like singing
```

변수

- 변수로의 접근

  ${name-word} : 설정 되었으면 name의 값으로 대치,
  그렇지 않으면 word 의 값으로 대치

  ${name+word} : name이 설정되었을 때 word만 대치

(예)

```
$ startData=${startdate-`date`}
$ echo $startDate                    ...설정된 값 보기
Tue Feb 4  06:51   CST 1998
$ flag =1                            ... 변수 할당
$ echo ${flag+flag is set}           ... 조건부 메시지 #1
flag  is  set
$ echo ${flag2+flag2  is  set}       ...조건부 메시지 #2
                                     ... 결과는 NULL
```

변수

◆ 변수로의 접근

${name=word} : name이 설정되지 않았으면 word를 할당하고 name의 값으로 대치

${name?word} : name이 설정되지 않았으면 word는 표준 에러 채널로 표시되고 그 셸은 종료.

(예)

```
$ echo X = ${X=10}                                  ...디폴트 값 할당
X=10
$ value=${X? 'X not set'}                           ...접근 성공
$ echo  $value                                      ...값 보기
10
$ value=${grandTotal? 'grand total not set '}       ...미설정
grandTotal : grand total not set
```

변수

- 표준 입력으로부터 변수 읽기
  - read {variable}+

(예)

```
$ cat script.sh
echo “enter your name, age : \c”
read name age
echo your name is $name
echo your age is $age
$ script.sh
enter your name, age : gtkim 18
your name is gtkim
your age is 18
```

변수

◆ 변수의 범위 전환

- export {variable}+

  환경변수로 범위 전환시키도록 명시된 변수들을 표시

  환경변수는 대문자 사용

(예)

```
$ export                          .... 어떤 변수도 명시되지 않으면
  export TERM                     .... 모든 변수들의 목록이 나타남.
$ DATABASE=/dbase/db                .... 지역변수 생성
$ export DATABASE
$ export                                .... export에 더해졌음
  export TERM
  export DATABASE
```

변수

- 변수의 범위 전환
  - env {variable = value}*[command]
    지정된 환경변수에 값을 할당

(예)

```
$ env                                  .... 환경을 표시
DATABASE=/dbase/db
HOME=/home/csservr2/glass
LOGNAME=glass
SHELL=/bin/sh
TERM=vt100
USER=glass
```

변수

◆ 읽기 전용 변수

- readonly {variable}+

  cf) 환경변수로 전환된 변수의 읽기 전용 상태는 상속되지 않음

  (예)

```
$ password=nicechej                    ...지역변수의 할당
$ echo $password
  nicechej
$ readonly password                    ...보호(수정방지)
$ readonly
  readonly password                    ... 모든 readonly 변수 나열
$ password=chej                        ...변수 수정시도
  password : is read only              ... error
```

변수

◆ 미리 정의된 지역 변수

이름	값
$@	모든 매개 변수 목록
$#	매개 변수의 수
$?	마지막 명령의 반환값
$!	마지막 후면 명령의 프로세스id
$_	현재 셸 옵션
$$	현재 쉘의 프로세스 ID
$0	쉘스크립터의 이름
$1 .. $9	$n 은 명령줄의 n번째 인수를 지칭
$*	모든 명령줄 인수들의 전체 목록

변수

(예)

```
$ cat script.sh
   echo there are $# command line arguments: $@
$ script.sh nofile tmpfile
   there are 2 command line arguments: nofile tmpfile
```

(예)

```
$ sleep 1000 &          …후면 프로세스 생성
29455                   …후면 프로세스의 프로세스 id
$ kill $!               …프로세스를 사멸시킴
29455 Terminated
$ echo $!               …프로세스 id는 아직도 기억됨
29455
```

변수

- 미리 정의된 환경변수

이름	값
$IFS	셸이 명령어를 실행하기 전 명령 줄을 토큰화 시키는 경우, 설정된 문자를 분리자로 사용. 공백, tab, newline문자를 포함
$PS1	명령 라인 프롬프트 형태를 제어하며 디폴트로 $를 갖음. 명령줄 프롬프트를 변경하려면 $PS1을 새로운 형태로 지정
$PS2	더 많은 입력이 셸에 의해 요구될 때 명령 줄의 보조 프롬프트 형태를 제어, 디폴트로 >을 갖음. 프롬프트를 변경시 새로운 형태로 지정해 준다
$SHENV	설정되어 있지 않으면, 새로운 셸이 생성될 때 셸은 사용자의 홈 디렉토리에서 “profile” 시작하기 화일을 찾음. 변수가 설정되어 있으면 셸은 $SHENV에 의해 명시된 디렉토리에서 찾음

변수

(예)

```
$ PS1=“sh ? ”                          ...새로운 주 프롬프트 설정
sh? oldIFS=$IFS                        …IFS의 이전 값을 기억
sh? IFS=“:”                            …단어 분리자를 콜론으로 변경
sh? ls:*.c                             …실행 성공!
  badguy.c    open.c    trunc.c    writer.c
sh? string=“a long\                    …긴 문자열을 할당
> string”                              …”>”는 보조 프롬프트
sh? echo $string
a long string
sh? PS2=“??? ”                         …보조 프롬프트 변경
sh? string=“a long\                    …긴 문자열을 할당
??? string”                            …”???”는 새로운 보조 프롬프트
sh? echo $string
a long string
sh?
```

산술 계산

◆ expr expression

cf) expression의 요소들은 공백으로 분리되어야 한다.

연 산 자	의미
\ * / %	곱셈, 나눗셈, 나머지
+ -	덧셈, 뺄셈
=\> \>= \< \<= !=	비교 연산자
\&	논리적 and
|	논리적 or

산술 계산

(예)

```
$ x=l                                          ... x의 최초값
$ x= 'expr $x +  l'                            ... x값 증가
$ echo $x
2
$ x='expr 2 + 3 \* 5'                          ... *  계산후 +계산
$ echo $x
17
$ echo 'expr \( 2 + 3 \) \* 5'                 ...\(    \) 먼저 계산
25
$ echo 'expr \( 4 \>5 \) '                     ….4 > 5 ?
0
$ echo 'expr \( 4 \> 5 \) \| \( 6\ <7 \) '     ….4 > 5 또는 6 > 7?
1
```

산술계산

◆ 문자열 연산자

(예)

$ echo 'expr **length** "cat" ' ….cat의 길이 구하기

3

$ echo 'expr **substr** "donkey" 4 3' ….부문자열 추출

key

$ echo 'expr **index** "donkey" "ke" ' ….부문자열 위치

4

$ echo 'expr **match** "smalltalk" '.*lk' ' ….일치하는 문자열 길이

9

조건식

- test expression

  expression이 참이면 0인 종료 코드를 돌려주고 그렇지 않으면 0이 아닌 종료 상태를 반환한다.

- test 명령의 옵션
  - Parentheses : \(\)
  - 논리 연산 : ! , -a , -o
  - 문자열 : -z , -n , = , !=
  - 숫자 : -eq , -ne , -gt , -ge , -lt , -le
  - 파일 : -r , -w , -x , -f , -d , -s

제어구조

- case-in-esac
  - 패턴에 대응하는 것을 찾아 그 패턴의 명령들을 실행하는 다중 선택형식.
  - expression은 문자열로 계산되는 식이고, pattern은 대표문자를 포함할 수 있음

```
case expression in
    pattern  { | pattern }* )
    list
    ;;
esac
```

```
#! /bin/sh
echo menu test program
stop=0      # 반복종료
while test $stop -eq 0    #완료시까지 반복
do
  cat << ENDOFMENU
   1  : print the date
   2:, 3 : print the current working direstory
   4   : exit
 ENDOFMENU
echo
echo
```

```
  echo -n `your choice?`  # prompt.
  read reply                    # read response.
  case $reply in              # process response.
       "l")
          date               # display date.
           ;;
       "2"|"3")
          pwd               # display working dictionnary
           ;;
        *)                    # default.
           echo illegal choice      # error.
            ;;
   esac
done
```

4번 옵션은 ?

제어구조

- for-do-done

  단어 리스트안의 각 맴버에 대해 명령의 집합을 한번씩 실행

```
for name [ in {  word  } * ]
do
      list
done
```

제어구조

(예)

```
$ vi for.sh                          ...스크립트 내용 표시
for color  in red yellow green blue
do
    echo one color is $color
done
$ for.sh                             ...스크립트 실행
one color is red
one color is yellow
one color is green
one color is blue
```

◆ if-then-fi

```
if  list1
then
    list2
elif   list3          ... elif는 여러번 반복가능함
then
    list4
else                  ...else는 1번 이하 발생함
    list5
fi
```

제어구조

```
$ vi  if.sh
echo -n ‘enter a nember : ’
read number
if [ $number -lt 0]   then   echo negative   # ..$number < 0 이면 음수
elif [$number -eq 0]  then  echo zero  # .. $number = 0 이면 0
else  echo positive
fi
$ if.sh                          ...스크립트 실행
enter a number : l
positive
$if.sh                           ...스크립트 다시 실행
enter a number : -l
negative
```

제어구조

- until-do-done
  - list1의 명령어를 실행하고 참이나 0이 아닌 값을 반환할 때까지 list2의 명령을 실행함

```
until list1
do
        list2
done
```

제어구조

(예)

```
$ cat until.sh
x=1
until [ $x -gt 2 ]                    #  ...$x가 2보다 크면 참
do
  echo x = $x
  x = ‘expr $x + 1 ’
done
$ until.sh
x = 1
x = 2
```

제어구조

- while-done

  list1의 의 조건이 만족되는 동안 list2의 명령을 반복 실행함

```
while  list1
do
        list2
done
```

제어구조

(예)

```
$ cat while.sh
x=1
while [ $x -le 2 ]                    ...$x가 3보다 작으면 참
do
  echo x = $x
  x = ‘expr $x + 1 ’
done
$ while.sh
x = 1
x = 2
```

그 밖의 내장명령어

- null 명령어

  어떤 연산도 수행하지 않음

- set 명령어

  변수나 명령 라인 매개 변수를 지정하거나 모든 변수를 표시

set -ekntuvx { arg }*

```
#! /bin/sh
x=1
while [ $x –le $1]
do
        y=1
        while [ $y –le $1 ]
        do
            echo –n `expr $x \* $y`  “ \c “
            y=`expr $y + 1`
        done
        echo
        x=`expr $x + 1`
done
```

```
$multi.sh 7
1       2       3       4       5       6       7
2       4       6       8       10      12      14
3       6       9       12      15      18      21
4       8       12      16      20      24      28
5       10      15      20      25      30      35
6       12      18      24      30      36      42
7       14      21      28      35      42      49
$
```

예제 : TRACK

script ; track[-ncount][-tpause] userId

track 은 지정된 사용자의 로그인과 로그아웃 세션을 감시한다. pause로 명시된 매초마다, track은 시스템을 검색하여 현재 시스템에 로그인 된 사용자의 목록을 만든다. 만일 명시된 사용자가 마지막 검색 후에 로그인 또는 로그아웃 하였다면, 이 정보는 표준 출력으로 출력된다. track 은 count 번의 검색이 완료될 때까지 수행한다. 디폴트로 pasue 20초 이고 count 는 10,000 번의 검색이다. track 은 표준 출력이 리다이렉트된 후면에서 실행된다

```
$ track -n3 -t200 kim ....kim 의 수행을 추적
track report for kim       ... 초기출력
login kim  ttyp3  Feb 5 06:53
track report for kim    ...  kim 로그아웃
logout  kim  ttp3  Feb 5 06:55
control C     ..  Control C 를 사용하여 프로그램 종료
stop tracking


$
```

```
# track.sed.→ diff 유틸리티의 출력을 편집 할 sed 스크립트
#숫자나 “–” 로 시작되는 모든 줄을 제거하고,
# ”<“을 #로그인으로, “>” 을 로그아웃으로 대치한다

/^[0–9].*/d
/^–––/d
s/^</login/
s/^>/logout/

#track,cleanup → 마지막 임시파일을 깨끗이 비우는 스크립트
echo stop tracking
rm –f .track.old.$1 .track.new.$1 .track.report.$1
```

```
# track bourn shell program
############################
pause=20
loopCount = 10000
rrror=0
for arg in $*
do
   case $arg in
     -t*)                #시간
     pause=‘expr substr arg 3 10’          #번호 추출
     ;;
      -n*)                                  #검색횟수
     loopCount=‘expr substr $arg 3 10’   #번호추출
     ;;
      *)
      user=$arg                                  #사용자이름
      ;;
   esac
done
```

```
If [ !"$user"]   #사용자 id가 발견되었는지 조사
then
error=1
fi
if[ $error -eq 1]   #오류발견시에 오류 표시
then
   cat << ENDOFERROR      #사용방법표시
   usage : track [-n#] [-t#} userId
   ENDOFERROR
exit 1  #쉘 종료
fi
#exit 트랩
trap 'track.cleanup $$; exit $exitCode' 0
trap 'exitCode=1; exit' 2 3   #INT/QUIT 트랩
> .track.old.$$          #이전트랙파일을 0으로
```

```
count=0                    #현재까지의 검색수
while [$count -lt $loopCount]
do
  who|grep $user | sort > .track.new.$$ #검색시스템
  diff .track.new.$$ .track.old.$$ | ₩
       sed -f track.sed > .track.report.$$
   if [ -s .track.report.$$ ]  #변화만 보고
   then                              #보고서 표시
     echo track report for ${user};
     cat .track.report.$$
  fi
mv .track.new.$$ .track.old.$$ #현재 상태 저장
sleep $pause                          #대기함
count='expr $count +1'           #검색횟수 수정
done
exitCode=0                          #종료코드 설정
```

실습 과제1(Bourn Shell)

1) 사용자의 이름을 키보드로부터 받아들여 이름을 입력하면 그 이름을 그대로 다시 출력해 주고 아무것도 입력하지 않으면 “무명씨”라고 출력하는 쉘 스크립트를 작성하시오.

2) 사용자로부터 패스워드(pass 변수 사용)를 입력받아 그것이 “Linux2"라는 값에 매칭이 이루어질 때까지 ”Sorry, try again."이라는 메시지를 보여주는 쉘 스크립트를 while 문을 이용하여 작성하시오.

3) 1 부터 10000 까지의 정수에서 소수를 구하는 프로그램을 본쉘로 작성하시오

4) 두개의 입력 변수들에 대한 공배수, 공약수 구하여 출력하시오

5) 구구단 프로그램을 셀로 작성하시오. 사용자로부터 구구단의 입력을 변수로 받아서 수행됩니다. 예, gugu 7 (7단 까지), gugu 9(9단까지)

(1) -le 를 gt 또는 ge로 변경

(2) while 을 for 로 변경하여 작성

6) TRACK 프로그램에서 입력된 사용자가 수행하는 명령어(프로그램)을 출력하는 옵션을 추가하시오

- $track -n100 -t200 -fgdkong

  출력 예

  login gdhong vi homework1

(7) ATM 기기 인터페이스 만들기

화면에 아래와 같이 출력하도록 하고, 하나의 항목을 선택하면 해당 기능이 10초 동안 수행되고 수행완료를 나타내는 문장을 출력한다. 하나의 거래가 종료된 후에는 화면에 출력된 내용이 모두 지워지고 첫번째 화면(메뉴 화면) 으로 이동하여 다른 입력을 처리할 준비를 한다

KNU ATM 입니다. 비밀번호를 입력하세요(메뉴화면)

비밀번호 : _ _ _ _

거래를 선택하세요. _ (처리 화면)

1) 송금
2) 출금
3) 대출
4) 입금
5) 중지

정상적으로 _ _ 거래가 수행되었습니다