

### 1. 개요

앞서 1장에서 셸의 기본적인 사항에 대하여 알아보았다. 이번 장에서는 셸 프로그래밍에 대하여 실습해보도록 하겠다.

앞 장에서는 분명히 셸은 명령어 처리기라고 배웠다. 하지만 난데없이 셸 프로그래밍에 대하여 공부하자고 하니 의아해할 수도 있을 것이다. 물론, 셸은 분명 DOS의 command.com과 같이 Unix 구조 내에서 명령어 처리기의 역할을 한다. 하지만 셸은 실제로 그 자체가 훨씬 더 강력하고 진정한 프로그래밍 언어이기도 하다.

셸은 인터프리터 언어이다. 즉, 컴파일을 필요로 하지 않는다. 따라서 간단한 프로그램에는 유용하나, 시간이 중요시되거나 프로세서를 과다하게 사용하는 작업에는 셸이 부적절하다.

그렇다면 왜 셸 프로그래밍을 하는 것일까? 이는 ‘재사용’이라는 Unix의 철학에 기인한다. Unix는 기본적으로 작고 간단한 역할을 수행하는 프로그램으로 이루어져 있다. 그리고 이 프로그램들을 Pipe나 redirection을 이용하여 연결하여 최종적으로 원하는 결과를 이뤄낸다. 간단한 예를 들면 다음과 같다.

```
$ ls -al | grep -v "^d" | more
```

이것은 ls, grep, more 프로그램을 사용하여 현재 디렉토리 상에서 하위 디렉토리를 제외한 파일들의 목록을 한 페이지 단위로 출력한다. 자세히 이야기하면 ls는 파일들의 목록을 출력하고, grep은 이 목록 중에서 디렉토리가 아닌 파일들을 추려낸다. 그리고 최종적으로 추려낸 파일 목록을 more 명령을 통해 한 페이지 단위로 출력하게 된다.

이와 같은 철학으로 인하여, Unix에서는 어떠한 내가 원하는 어떠한 작업을 수행하기 위해선 기존의 Unix 프로그램들을 구조적으로 연결하여 조립할 필요가 있다. 이러한 조립 과정이 물론 한 줄의 명령어로 표현될 수도 있지만, 프로그램을 제어하는 부분을 넣거나, 사용자와의 상호작용을 위해서는 한 줄의 명령어로는 턱없이 부족하다. 따라서 Unix에서는 이러한 명령어들을 스크립트로 작성하여 실행하도록 하고 있다. 그리고 이러한 셸 스크립트를 작성하는 것을 셸 프로그래밍이라고 한다.

이 장을 통해서 우리는 하나의 간단한 프로그래밍 실습을 통해, 셸 프로그래밍을 할 때의 유효한 형식, 구조, 명령들에 대하여 배울 것이다.

### 2. 문법

#### 2-1. 변수

##### 2-1-1. 변수 선언과 할당

변수는 변화하는 값을 의미하는 상징적인 이름이다. Bourne shell에서는 c언어와 달리 변수를 선언하지 않고 사용할 수 있다. 즉, 변수의 선언이 따로 있는 것이 아니라 어떠한 변수에 값을 대입하는 것이 변수의 선언을 대신한다. 보통 shell 명령어가 소문자로 이루어져 있기 때문에 변수의 이름은 암묵적으로 대문자를 많이 사용한다.

#### ☞ 예제

```
$ name=sogang
$ echo $name
sogang
$ name=computer
$ echo $name
computer
```

변수에 값을 할당할 때는 위의 예제에서처럼 '='을 사용한다. 이때 = 사이에는 공백이 있어서는 안 된다. 이렇게 할당된 변수는 \$name처럼 변수 이름 앞에 \$를 사용해서 값을 사용하게 된다.

#### ☞ 예제

```
$ name=sogang computer
bash: computer: command not found
$ name="sogang computer"
$ echo $name
sogang computer
$ ux=UNIX
$ echo ${ux}tm
UNIXtm
$ echo "$ux"tm
UNIXtm
$ echo $uxtm
$

$ set -u
$ echo $uxtm
bash: uxtm: unbound variable
$ set +u
$ echo $uxtm
$
```

공백이 들어간 변수 값을 사용할 때는 name="sogang computer"처럼 큰 따옴표로 묶어 주어야 한다. 또한, 위의 예처럼 변수값 UNIX에 문자열 tm을 합쳐서 출력시키는 경우{}나 큰따옴표(" ")를 사용해야 한다. 그렇지 않고 \$uxtm처럼 사용하면 uxtm이라는 변수를 의미하는 것이기 때문에 공백 문자로 출력된다. 기본적으로 선언되지 않은 변수에 대해서는 공백 문자가 출력되게 된다. 하지만 경우에 따라서는 선언되지 않은 변수를 사용하려 할 때 에러 메시지를 출력하는 것이 유용할 수도 있다. 이럴 때에는 "set -u" 명령을 사용하면 된다.

#### ☞ 예제

```
$ flower=tulip
$ readonly flower
$ flower=rose
bash: flower: readonly variable
```

readonly 명령을 사용하면 변수를 읽기 전용으로 바꿀 수 있다.

#### 2-1-2. Exporting shell variables

변수는 선언된 셸 내에서만 사용된다. 하지만 export 명령을 사용하면 변수를 다른 곳에서도 사용할 수 있게 된다. 기본적으로 변수가 적용되는 범위는 해당 셸로 제한되게 된다. script를 실행시키면, 새로운 셸이 실행되고 그 위에서 script가 동작하게 되므로 다른 셸에서는 script에서 정의한 셸 변수를 사용할 수 없다. 마찬가지로 다른 셸에서 정의한 변수를 script 내에서는 사용할 수 없다. 하지만 만약 변수를 모든 셸에서 사용하고 싶다면 export 명령을 사용하면 된다.

#### ☞ 예제

```
$ cat foodlike
echo $pn butter is yummy

$ pn=peanut
$ ./foodlike
butter is yummy
$ export pn
$ ./foodlike
peanut butter is yummy
```

#### 2-1-3. Automatic shell variables

Automatic shell 변수는 특수한 기능을 수행하는 변수를 나타낸다([표 1]).

[표 1] Automatic shell variable

변수	설명
\$?	직전에 실행한 명령에 대한 exit value를 나타낸다. 0값이 return 되면 직전의 명령이 정상적으로 수행되었음을 나타내고, 나머지 값이 return 되면 직전의 명령이 정상적으로 수행되지 못했음을 나타낸다.
\$\$	shell의 process id를 나타낸다.
#!	shell이 실행시킨 background process의 number를 나타낸다.
\$#	script의 argument의 개수를 나타낸다. \$1은 첫 번째, \$2는 두 번째... \$9는 9번째 argument를 나타낸다.
\$*	argument의 수를 나타낸다. \$#과 다른 점은 \$#은 9개까지 밖에 나타내지 못하지만, \$*는 입력된 모든 argument를 나타낼 수 있다.
\$@	\$*과 비슷한 기능을 수행한다. 단, "\$*"의 경우는 "\$1 \$2 ... \$n"을 나타내지만 "\$@"은 "\$1" "\$2" ... "\$n"을 의미한다.

#### ☞ 예제

```
$ cp abc cdf
cp: cannot stat `abc': No such file or directory
$ echo $?
1
$ echo $$
24085
$ ps
  PID TTY          TIME CMD
 24085 pts/0    00:00:00 bash
 24129 pts/0    00:00:00 ps
$ echo $?
0
$
```

`cp abc cdf` 와 같은 잘못된 명령어 뒤에 `echo $?`를 하면 0 이외의 값이 나오는 것을 볼 수 있다. 반면 마지막 `echo $?`의 결과가 0인 것은 직전의 `ps` 명령이 정상적으로 수행되었기 때문이다. `$$`를 통해서 확인한 셸의 pid가 `ps` 명령으로 출력시킨 pid와 동일한 것을 볼 수 있다.

#### ☞ 예제

```
$ cat tt
echo $# $2 $3 $4 $5 $6 $7 $8 $9 $10 $11 $12

$ ./tt a b c d e f g h i j k
11 a b c d e f g h i a0 a1 a2
$
```

위의 화면을 보면 \$10부터는 argument를 제대로 받지 못한다는 것을 알 수 있다. 10개가 넘는 argument를 받기 위해서는 아래와 같이 `$*`나 `$@`를 이용해야 한다.

#### ☞ 예제

```
$ cat tt1
echo $# $*

$ ./tt1 a b c d e f g h i j k
11 a b c d e f g h i j k
$ cat tt2
echo $# $@

$ ./tt2 a b c d e f g h i j k
11 a b c d e f g h i j k
$
```

#### 2-1-4. Standard shell variables

set 명령을 통해서 list 되는 변수들을 나타내며, 일반적으로 System Environment 변수로 사용된다. 로그인을 하면 bash의 경우 .bash\_profile이 수행되고, 다음으로 .bashrc가 실행되어 사용자의 환경을 설정하게 되는데 여기서 쓰이는 변수들이 standard shell variable이다.

[표 2] Standard shell variables

변수	설명
\$PATH	명령어 실행을 위해서 검색하는 디렉토리 경로를 나타낸다. PATH 상에 나타나 있는 디렉토리 순서대로 명령어를 검색하여 먼저 찾게 되는 디렉토리의 명령어를 실행한다. 가령 ls 명령어가 /bin과 /usr/bin에 각각 존재한다고 하자. \$PATH=/bin:/usr/bin:/usr/sbin으로 설정되어 있다면 사용자가 셸에서 ls 명령을 쳤을 때, /bin/ls 명령을 실행하게 된다.
\$CDPATH	cd 명령을 사용할 때의 기준점. 만약 \$CDPATH를 /home/abc로 설정한 경우 cd ttl을 하면, 현재 디렉토리가 /home/abc/ttl로 옮겨지게 된다.
\$LOGNAME	로그인 이름(계정명)
\$IFS	internal field seperator, 기본적으로 공백문자(스페이스, 탭)로 설정
\$SHELL	로그인하면 실행되는 기본 셸의 이름
\$PS1	로그인하였을 때 나타나는 프롬프트 모양
\$TERM	접속한 terminal
\$HOME	user의 홈 디렉토리

#### 2-1-5. Quoting special characters

유닉스에는 세 가지 인용(quoting) 방법이 있다.

- backslash(\)
  - 한 문자에 대하여 meta character에 관계없이 있는 그대로 하나의 문자로 인식한다.
- single quote(' ')
  - single quote 내에 있는 문자열을 있는 그대로 처리한다.
- double quote(" ")
  - double quote 내에 변수를 변수 값으로 치환하여 처리한다.

## 예제

```
$ echo $HOME
/home/venus
$ echo '$HOME'
$HOME
$ echo \ $HOME
$HOME
$ echo "$HOME"
/home/venus
$
```

## 2-2. 조건문

### 2-2-1. if

#### 문법

```
if condition
then
    statements
[elif condition
    then statements...]
[else
    statements]
fi
```

if 문(조건문)은 가장 기본적인 흐름 제어(flow control) 기법 중에 하나이다. 가장 간단한 형식은 *condition*의 조건이 참인 경우 *statements*를 수행하는 것이다. 여기에 **else** 구문이 추가가 되면 *condition*이 거짓인 경우 **else** 이하의 *statements*를 수행한다. **elif** (else if) 또한 추가 가능한데, **elif**의 *condition*이 참인 경우 **elif** 이하의 *statements*를 수행한다. **if** 구문의 끝은 항상 **fi**로 끝난다.

*condition*에 쓰일 수 있는 operator는 다음과 같다.

[표 3] string 비교 연산자

연산자	참인 조건
<i>str1</i> = <i>str2</i>	<i>str1</i> 과 <i>str2</i> 가 같은 경우
<i>str1</i> != <i>str2</i>	<i>str1</i> 과 <i>str2</i> 가 틀린 경우
<i>str1</i> < <i>str2</i>	<i>str1</i> 이 사전적 순서로 <i>str2</i> 보다 먼저인 경우
<i>str1</i> > <i>str2</i>	<i>str1</i> 이 사전적 순서로 <i>str2</i> 보다 나중인 경우
-n <i>str1</i>	<i>str1</i> 이 null이 아닌 경우 (예제) if [ -n "\$FILENAME" ] then ... fi
-z <i>str1</i>	<i>str1</i> 이 null인 경우

[표 4] 파일 속성 연산자

연산자	참인 조건
-d <i>file</i>	<i>file</i> 이 존재하고, 또한 그 종류가 디렉토리이다.
-e <i>file</i>	<i>file</i> 이 존재한다.
-f <i>file</i>	<i>file</i> 이 존재하고, 또한 그 종류가 일반 파일이다(디렉토리나 특별한 타입의 파일이 아님).
-r <i>file</i>	<i>file</i> 에 대한 읽기 소유권을 가지고 있다.
-s <i>file</i>	<i>file</i> 이 존재하고, 해당 파일이 비어 있는 파일이 아니다.
-w <i>file</i>	<i>file</i> 에 대한 쓰기 소유권을 가지고 있다.
-x <i>file</i>	<i>file</i> 에 대한 실행 소유권을 가지고 있거나, <i>file</i> 이 디렉토리라면 디렉토리에 대한 탐색 권한을 가지고 있다.
-O <i>file</i>	<i>file</i> 을 소유하고 있다.
-G <i>file</i>	<i>file</i> 의 group ID가 현재 셸을 실행시키는 사람이 속한 group 중에 하나다.
<i>file1</i> -nt <i>file2</i>	<i>file1</i> 이 <i>file2</i> 보다 더 최신의 것이다.
<i>file1</i> -ot <i>file2</i>	<i>file1</i> 이 <i>file2</i> 보다 더 오래된 것이다.

[표 5] 산술 비교 연산자

연산자	참인 조건
-lt	작다.
-le	작거나 같다.
-eq	같다.
-ge	크거나 같다.
-gt	크다.
-ne	같지 않다.

앞서 제시한 연산자는 test 연산자와 함께 사용하거나 [...] 안에서 사용해야 한다. 아래의 두 방법은 동일한 결과를 나타낸다.

#### ☞ 예제 1

```
if [ "$FILE1" = "$FILE2" ]
then
...
fi
```

## ☞ 예제 2

```
if test "$FILE1" = "$FILE2"
then
...
fi
```

이러한 연산자는 조건문뿐만이 아니고, 반복문, 선택문 등에서도 쓰일 수 있다.

## ☞ 예제 3

```
$ cat if1
if [ $# -eq 4 ]
then
    echo $4 $3 $2 $1
else
    echo Usage : $0 arg1 arg2 arg3 arg4
fi

$ ./if1
Usage: ./if1 arg1 arg2 arg3 arg4
$ ./if1 1 2 3 4
4 3 2 1
$
```

## 2-3. 반복문

### 2-3-1. for

#### ¶ 문법

```
for name [in list]
do
    statements that can use $name...
done
```

for 문을 사용하면 *list*에 있는 word들이 하나씩 *\$name* 변수에 할당된다. 만일 *in list*가 생략되면 *list*는 기본적으로 "\$@"로 설정이 된다. *list*는 하나의 변수로 볼 수 있는데, 변수의 스트링을 *\$IFS*로 구분하여 하나의 word씩 *\$name*에 할당하게 된다.



## 예제

```
$ cat for1
IFS=:

for dir in $PATH
do
    ls -ld $dir
done
$ echo $PATH
/bin:/usr/bin:/sbin:/usr/sbin:/usr/local/bin
$ ./for1
drwxr-xr-x  2 root    root      4096 Apr 11  2003 /bin
drwxr-xr-x  2 root    root     53248 Dec 16 21:56 /usr/bin
drwxr-xr-x  2 root    root      8192 Apr 15  2003 /sbin
drwxr-xr-x  2 root    root     12288 Apr 15  2003 /usr/sbin
drwxr-xr-x  2 root    root      4096 Feb  7  1996 /usr/local/bin
$
```

위의 셸 스크립트는 \$PATH에 속해 있는 디렉토리들을 화면에 출력한다. \$PATH는 구분자(separator)로 콜론(:)을 사용하기 때문에 하나의 디렉토리를 가져오기 위하여 IFS를 콜론으로 설정하였다.

### 2-3-2. while and until

#### 문법

```
while condition
do
    statements...
done
```

#### 문법

```
until condition
do
    statements...
done
```

while이나 until 문은 어떤 주어진 *condition* 동안에 *statements*를 반복 수행한다. for문은 주로 list내의 값들을 for문 안에서 이용하기 위해서 사용하나, while이나 until는 일정한 조건 동안에 반복하여 수행하는 것이 주된 목적이다. 모호하게 들릴지도 모르나 이는 다음의 예제를 보면 쉽게 이해가 갈 것이다.

## 예제

```
$ cat square
# squares - prints the square of integers in succession
int=1
while [ $int -lt 5 ]
do
    sq=`expr $int \* $int`
    echo $sq
    int=`expr $int + 1`
done
echo "job complete"
$ ./square
1
4
9
16
job complete
$
```

위의 예에서 `expr`는 `` (acute accent, 키보드 상 ~ 밑에 있는 것)으로 묶었다. 따라서 `sq`에는 `expr` 명령의 수행된 결과가 저장되게 된다. `expr`에 대한 자세한 정보는 뒤의 부록을 참조한다.

## 2-4. 선택문

### 2-4-1. case

#### 문법

```
case expression in
    pattern1 )
        statements ;;
    pattern2 )
        statements ;;
    ...
    * )
        statements ;;
esac
```

`case`문은 C언어나 Pascal에서 나오는 `case` 구문과 비슷하다. 하지만 여기에서는 정수 값이나 문자 값을 비교하는 것이 아니고 패턴에 의해서 흐름이 선택되어 수행된다.

## 예제

```
$ cat hi
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 recognized" ;;
esac
$ ./hi
Is it morning? Please answer yes or no
yes
Good Morning
$
```

위의 예제는 간단한 입력을 받아서 입력에 따라 적절한 출력을 해주는 셸 스크립트이다. read 명령은 stdin으로부터 입력받아서 이를 변수에 넣어준다. 이 스크립트에서는 세 가지 출력문 각각을 여러 가지 입력에 따라 출력해주는 기능을 한다. 이것은 스크립트를 더 짧게 만들고, 실용적이고 읽기 쉽게 해준다.

## 2-5. 함수

### 문법

```
name()
{
    statements
}
```

여타의 다른 프로그래밍 언어와 마찬가지로 셸 스크립트에서도 함수를 정의하여 사용할 수 있다. 어느 정도 규모의 셸 스크립트를 작성한다면 함수를 사용하는 것은 코드를 구조화 하는데 많은 도움을 준다.

## 예제

```
$ cat yesno
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
$ ./yesno lazy lune
Original parameters are lazy lune
Is your name lazy ?
Enter yes or no: yes
Hi lazy, nice name
$
```

스크립트가 실행될 때 함수 `yes_or_no`는 정의되지만, 아직까지 실행되지 않는다. 스크립트는 `if`문에서 `$1`을 원래 스크립트에 대한 첫 파라미터인 `lazy`로 대체한 후에 함수에 대한 파라미터로 문장의 나머지 내용을 전달하며, 함수 `yes_or_no`를 실행한다. 함수는 파라미터 `$1`, `$2` 등에 저장된 파라미터를 사용하고 나서 호출자에게 값을 반환한다. `if` 구조는 반환 값에 따라 적절한 문장을 실행한다.

### 3. 프로그래밍 문제

#### 온라인 전화/주소록

전화번호와 주소록이 기록되어 있는 데이터 파일이 주어져 있다. 사람의 이름이나 주소의 일부분, 전화번호의 일부분이 입력으로 주어지면, 데이터 파일에서 검색 조건에 맞는 데이터를 검색하여 포맷에 맞게 출력한다. 셸 스크립트의 파일명은 "phone"으로 한다.

#### 데이터 파일 형식

각 줄마다 하나의 레코드가 입력되며, 하나의 레코드는 "이름|주소|전화번호"로 이루어져 있다. 각 필드를 구분하기 위한 구분자로 '|' 문자가 쓰인다. 예를 들면 다음과 같다.

```
홍길동|서울시 마포구 신수동 서강대학교 AS관 301호|02-705-2665
박문수|서울시 서대문구 신촌동 연세대학교 연구관 102호|02-708-4678
Andrew|경기도 의정부시 호원동 23-12번지|031-827-7842
```

데이터 파일명은 "\$HOME/mydata"으로 한다.

#### 입력 형식

입력은 셸 프롬프트 상에서 인자(argument)로 주어진다. 이때 주어지는 인자는 데이터에 대한 부분적인 정보이다. 예를 들어, 이름이나 혹은 지역 등이 인자로 주어질 수 있다. 인자는 여러 개가 주어질 수 있으며, 이 경우에는 인자에 대한 모든 데이터를 출력하게 된다. 단 구분자인 "|"가 인자로 들어올 경우에는 무시한다.

#### 출력 형식

검색된 레코드를 다음과 같은 포맷으로 출력한다.

```
----->
이름 :
주소 :
전화번호 :
<-----
```

만약 검색된 레코드가 없다면 아무것도 출력하지 않는다.

#### 입출력 예

```
$ ./phone
Usage: phone searchfor [...searchfor]
(You didn't tell me what you want to search for.)
$ ./phone Andrew
----->
name : Andrew
address : 경기도 의정부시 호원동 23-12번지
phone : 031-827-7842
<-----
$ ./phone 서울시
----->
```

```

name : 홍길동
address : 서울시 마포구 신수동 서강대학교 AS관 301호
phone : 02-705-2665
<-----
----->
name : 박문수
address : 서울시 서대문구 신촌동 연세대학교 연구관 102호
phone : 02-708-4678
<-----
$ ./phone 홍길동 신수동
----->
name : 홍길동
address : 서울시 마포구 신수동 서강대학교 AS관 301호
phone : 02-705-2665
<-----
$

```

## 4. 실험 방법

### 4-1. 문제 해결

셸 프로그램이 해야 할 일은 다음과 같다.

#### ▪ 에러 처리

이 프로그램에서는 단 하나의 에러를 처리하도록 한다. 셸 프로그램을 실행시키기 위해서는 반드시 하나 이상의 인자(argument)를 필요로 한다. 이 인자는 위의 입출력 예에서 살펴볼 수 있듯이 전화번호부에서 찾고자 하는 문자열이다. 따라서 셸 프로그램은 해당 인자가 주어지지 않는다면 사용 예(usage)를 출력해준다. 사용 예의 출력 형식은 위의 예제에서 나온 형식을 따르기로 한다.

#### ▪ 입력 인자 변경

해당 문자열을 검색하기 위하여 본 셸 프로그램에서는 egrep을 사용한다. egrep은 grep의 확장된 형태로, 정규 표현식(regular expression)을 제공하는 grep이다. egrep을 통해 여러 단어를 검색할 때에 사용하는 명령은 다음과 같다.

```
egrep "(arg1|arg2|...|argn)" datafile
```

하지만 셸 프로그램에서 들어오는 인자의 형식은 다음과 같다.

```
$ ./phone arg1 arg2 ... argn
```

따라서 arg1 arg2 ... argn을 (arg1|arg2|...|argn)의 하나의 문자열로 고쳐주는 부분이 프로그램에 포함되어야 한다. egrep에 대한 자세한 설명은 man page를 참조하기 바란다.

#### ▪ 데이터 파일에서 해당 데이터 추출 및 출력

인자가 변경 되었으면 egrep을 통해 원하는 데이터를 추출한다. 추출된 데이터를 적절한 포맷에 맞게 출력한다. 포맷을 맞추는 것은 awk 프로그램을 이용할 수 있다.

#### 4-2. 프로그래밍 편집 및 실행

이 문제를 해결하기 위해 필요한 파일은 다음과 같다.

[표 6] 생성 파일

파일명	설명
phone	입력된 인자를 적절하게 바꾼 뒤 <code>egrep</code> 을 호출한다. <code>egrep</code> 에서 얻은 결과를 <code>display.awk</code> 를 호출하여 적절하게 출력한다. 이 이외에도 에러 처리를 수행하는 코드를 포함하고 있다.
display.awk	phone에서 넘어온 데이터 파일의 형식을 적절하게 포맷에 맞추어 출력한다. 이 프로그램에서는 <code>field</code> 구분자로 <code>" "</code> 를 사용해야 하는데 이를 설정하기 위해서는 <code>BEGIN { }</code> 내에 <code>FS=" ";</code> 를 넣어주면 된다.

`awk` 프로그래밍을 별도의 파일에 한 경우, 이를 호출하여 실행하는 방법은 다음과 같다.

```
awk -f display.awk
```

그 밖의 `awk` 관련 내용은 뒤의 부록을 참조하기 바란다.

각 프로그래밍의 편집은 `vi`를 사용 한다. 이를 위하여 부록의 `vi`를 읽어보기 바란다.

#### 4-3. 숙제 및 보고서 작성

첨부한 예비보고서의 내용을 작성하여 실험 당일 제출한다. 숙제 및 결과 보고서 제출은 강의 일정에 따라 제출한다.