Shell 프로그래밍의 기본

Katja and Guido Socher

katja (at) linuxfocus.org

도운이

윤상배

yundream (at) coconut.co.kr

이 문서는 Linux focus 의 2001 년 9 ~ 10 월 기사의 "point Shell Programming" 을 번역한것이다. 오타 및 탈자 용어의 사용에 있어서 그리 주의를 기울이지 않은 관계로 문서가 매끄럽지 않게 진행될수 있음을 미리 밝히며, 이러한 버그는 차후 업그레이드? 버젼에서 수정할것임을 약속? 한다. 그리고 원본에 포함되어 있지 않는 내용들도 상당수 들어가 있다.

리눅스를 위한 다양한 그래픽인터페이스를 지원하는 프로그램들이 있음에도 불구하고 쉘 스크립트는 여전히 많은 사랑?을 받으며 다양한 부분에서 필수적인 도구로 사용되고 있다. 언뜻 보기에 그래픽인터페이스의 다른 도구들에 비해서 사용하기 어려워 보이고 다분히 원시적으로 보이지만, 쉘 프로그래밍을 통하여, 많은 노동이 필요한 작업들을 단순화 시킬수 있으며, 훌륭한 시스템 관리 도구로 사용할수 있기 때문이다. 또한 대규모프로젝트를 시작하기 전에 간단한 프로토타입을 만들수 있도록 해주며, 전체적으로 어떤 일을 수행하는데 있어서 시간과 노력을 절약하도록 도와준다. 이렇게 절약된 시간을 좀더 생산적인일(디아블로를 하거나, 커피를 마시는) 에투자할수 있도록 도와준다. 이 문서를 통해서 우리는 이러한 쉘 스크립트의 사용법에 대해서 배우게 될것이다. 문서의 설명은 글을읽는 여러분이 아주 기본적인 프로그래밍을 해본적이 있고 Unix/Linux 환경을 한번이라도 접해봤을 거라는 가정하에 이루어질 것이다.

이 문서는 GNU Free Documentation License 를 가집니다.

1. 들어가기전에

1.1. 필요한 것들

리눅스가 깔려있는 컴터 한대만 있으면 된다. 또는 원격 리눅스 계정을 가지고 있으면 된다. bash 쉘이 필요하겠지만, 대부분의 리눅스 박스에 거의 100% 깔려있다고 보면 되니 걱정할 필요 없을 것이다. 리눅스박스에 로그인 한다음에 "bash" 라고 명령을 내려보자, 제대로 실행된다면 bash 쉘이 깔려 있는 것이다. 자신의 시스템이 아닐경우 기본쉘로 bash 외의 csh 나 tcsh 등의 쉘이 기본쉘로 지정되어 있을수도 있다. 이럴경우 관리자에게 기본쉘의 변경을 요청해야 한다 (물론 로그인다음에 프롬프트 상에서 "/bin/bash" 명령을 실행함으로써 베쉬 쉘 환경

으로 들어갈수도 있지만) 자신의 기본 쉘 환경을 알아보고 싶다면 "/etc/passwd" 파일을 보면 된다. 참고로 필자의 경우 "yundream:x:500:500::/home/yundream:/bin/bash" 로 되어 있다.

1.2. 쉘이란 무엇인가

쉘은 다른말로 명령어 해석기라고 불리운다. 인간의 명령어를 해석하여(주로 자판을 통하여 명령함), 시스템(커널)에 전달하는 일을 한다. 명령어를 해석하는 방법에 따라서 csh, tcsh, ksh, bash 등의 다양한 쉘이 존재한다. 이를테면 사용자와 커널간의 원할한 통신을 가능하도록 도와주는 통역사 정도로 생각하면 될것이다. 어떤 이름의 쉘이건 기본적으로 하는 일은 같다. 다만 변수의 사용방법, 사용되어지는 문법, 그밖의 추가적인 기능들에 있어서 차이가 있을 뿐이다. 그러한 이유로 하나의 쉘을 사용할수 있다면, 다른 쉘도 그리 어렵지 않게 사용가능하다.

2. 스크립트를 만들자

리눅스에는 매우다양한 여러 종류의 쉘이 존재하는데, 이 문서에서는 bash(bourne again shell)을 사용하도록 하겠다. 이유는 가장 널리 사용되며, 배우기 쉽고 또한 필자가 사용하는 쉘이기 때문이다. 때문에 이 문서에 있는 모든 쏘쓰역시 bash를 기준으로 작성되어 있다. 쉘 프로그래밍을 하기 위해서는 nedit, kedit, emasc, vi 등의 문자 편집기가 필요하다. 마음에 드는 아무 편집기나 사용하도록 하자. 쉘 프로그램의 첫번째 라인은 아래와 같은 라인을 포함해야 한다.

```
#!/bin/sh
```

#! 은 쉘에게 이 프로그램을 실행하기 위해서 #! 다음에 오는 아규먼트를 실행프로그램으로 사용한다는 것을 알려주기 위해서 사용된다. 위의 경우에는 /bin/sh 를 사용하라고 되어있다. 일단 위의 형식으로 프로그램을 만들고 이를 저장하였다면, 이를 실행가능한 파일로 만들어 줘야 한다. 리눅스(Unix 포함) 은 윈도와 달리 파일의 확장자명(.exe .com 등)으로 실행파일 유무를 결정하지 않으며, 파일권한 설정의 변경을 통하여 실행파일로 결정한다. 실행파일로 만들기 위해서는 아래와 같이 하면 되다.

```
chmod +x filename
```

위와 같이 하고나서 ./filename 명령을 내리면 작성한 쉘 프로그램이 실행된다.

3. 주석

주석은 "#" 을 이용하며 "#" 에서부터 라인의 마지막 까지가 주석으로 처리된다. 주석부분은 쉘이 실행될때 무시된다. 주석은 프로그램에 대한 이해를 쉽게 하기 위한 방안으로 사용된다.

```
#!/bin/sh
# 만든이 : yundream
```

```
# 하는일 : hello world 를 출력한다.

echo "hello world"   # 문자열을 출력한다.
```

4. 변수

변수는 모든 프로그래밍 언어의 가장 기본이 되는 요소이다. "변수" 라고부르고는 있지만, 어떤 데이타를 저장하기 위한 "임시공간" 이라고 말하는게 더욱 정확한 표현일듯 싶다. 인간 두뇌로 하자면, 대뇌피질의 한구역쯤 될까? 쉘에의 변수의 데이타 타입은 string(문자열) 만을 가지며, C 언어와 같은 변수 선언을 필요로 하지 않는다. 이는 perl, python, php 역시 마찬가지이다. 변수에 값을 넣기 위해서는 다음과 같이 한다.

```
varname=value
```

변수의 사용을 위해서는 변수명 앞에 $ 를 붙여주면 된다. 변수에 데이타를 저장하기 위해서는 대입연산자"=" 를 사용한다. 대입연산자와, 피연산자/연산자 사이에는 공백이 존재하면 안된다.

```
#!/bin/sh
# 변수에 값을 할당한다.
a="hello world"
# 이제 a라는 변수에 등록된 값을 화면에 출력한다.
echo "a is : $a"
```

위의 내용으로 파일을 만든다음 저장하고, 실행파일로 만들어서 실행하면 다음과 같은 결과물이 화면에 출력될 것이다.

```
[yundream@localhost /home]# ./test.sh
A is : hello world
```

아래와 같이 프로그램을 만들어 보자

```
num=2
echo "this is the $numnd"
```

우리가 원하는 값은 "this is the 2nd" 이다. 그러나 우리가 원하는 값대신 "this is the " 라는 문자열이 출력된다. 왜냐면 쉘은 "numnd"를 하나의 변수명으로 생각하고 있기 때문이다. 이럴 경우애는 아래와 같이 코딩하면 된다.

```
num=2
echo "this is the ${num}nd"
```

위의 프로그램을 실행시키면 우리가 원하는 값인 "this is the 2nd" 라는 출력물을 얻게 된다.

5. 대화모드

쉘스크립트를 제작하는데 있어서 위에서와 같이 파일로 저장한다음에 이를 실행권한을 주어서 실행하는 방법도 있지만, 간단하게 작업을 수행해야 할경우 파일을 만들지 않고, 쉘상에서 직접 실행시키는 방법도 있다.

```
[root@localhost /root]# test="hello world"
[root@localhost /root]# echo $test
hello world
```

아래는 좀더 복잡한 예제이다.

```
[root@localhost /root]# if [ "$SHELL" = "/bin/bash" ]
> then
> echo "your login shell is the bash (bourne again shell)"
> else
> echo "your login shell is not bash but $SHELL"
> fi
your login shell is the bash (bourne again shell)
```

6. 쉘명령과 제어구조

이번장에서는 쉘스크립트를 다루는데 필요한 3가지 부수적인 주제를 설명하게 될것이다.

6.1. 유닉스 명령어들

쉘스크립트는 유닉스 명령어들의 집합이므로, 유닉스 명령어에 대해서 어느정도의 숙지가 필요하다. 이러한 명령어들은 주로 파일과 문자열을 편집하기 위해서 쓰여진다. 이러한 명령어들중 자주 쓰이는 명령어들을 정리했다.

표 **1.** 자주 사용되는 유닉스 명령어들

echo "some text"	some text 를 화면에 출력한다
wc -l file	파일의 라인수
cp sourcefile destfile	sourcefile 을 destfile 로 복사
mv oldname newname	파일이름을 바꾸거나 파일의 이동
rm file	파일 지우기
grep 'pattern' file	파일에서 pattern의 문자열을 찾기
cub -b colnum file	파일에서 문자열을 컬럼단위로 잘라서 보여줌
cat file.txt	file.txt 를 표준출력(stdout) 시킴

file somefile	somefile 의 파일타입 알아내기
read var	입력값을 변수명var 에 대입
sort file.txt	file.txt 를 라인단위로 정렬
uniq	파일에서 중복되는 문자열을 제거
tee	표준출력되는 정보를 파일로 쓰기
basename file	디렉토리명을 제외한 파일의 실제이름을 돌려줌
dirname file	파일이름을 제외한 디렉토리의 이름을 돌려줌
head file	파일의 처음 몇라인을 출력함
tail file	파일의 마지막 몇라인을 출력함
sed	정규표현에 의한 문자열의 검색및 치환에 사용됨

6.2. pipes(파이프), redirection(재지향)

Pipes(|) 는 하나의 프로그램을 실행시켜서 발생된 표준출력 데이타를 다른 프로그램에 표준입력 시키고자 할때 사용된다. 즉 프로세스간 데이타 통신을 위한 하나의 방법으로 사용된다.

```
grep "hello" file.txt | wc -l
```

위의 스크립트는 file.txt 에서 hello문자열을 포함한 라인을 찾아서(grep), 몇개의 라인이 hello 를 포함하고 있는지의 라인수를 돌려준다.

redirection 은 "재지향" 이라고 불리워진다. 우리나라 말로 표현하자면 "다시 향하게 하다" 이며, 어떤 프로그램의 출력 정보를 다른곳으로 다시 향하게 할때 쓰인다. 여기에서 다른곳이란 주로 파일을 뜻한다. 재지향을 위해서는 ">" 과 ">>" 을 쓴다. ">"을 사용하게 되면 새로운 파일을 만들게 된다. 기존에 같은 이름의 파일이 있었다면, 그 파일은 지워지게 된다. ">>" 을 쓰게 되면 기존에 같은 이름의 파일이 있다면 그 파일의 마지막부분에 덧 붙여지게 된다. 같은 이름의 파일이 없다면 물론 새로운 파일을 만들게 된다.

```
[root@localhost /root]# cat address.txt | grep "seoul"  > seoul_add.txt
```

address.txt 에는 주소정보가 담겨 있다. 위의 스크립트는 이중 주소지가 "seoul" 인 정보만을 따로 뽑아서 seoul_add.txt 에 저장하는 일을 수행한다.

6.3. 제어구조

"if"는 참인지 거짓인지 판단할때 사용한다. 참이라면 then 부분을 실행하고 그렇지 않다면 else 부분을 실행한다.

```
if .....
then
....
else
....
fi
```

if 의 가장 유용한 사용처는 "상태" 를 테스트(test) 하는데 있다. 즉 문자비교, 파일이 존재하는지, 파일이 실행파일인지, 디렉토리인지, 읽을수 있는지 ... 등에 유용하게 사용할수 있으며, 이러한 작업의 제어를 위한 특수한 명령어 들을 제공한다. 이러한 "test" 조건들은 "[ ]" 사이에 쓰면된다. "[" 과 "]" 사이에는 반드시 공백문자가 들어가야 된다는 것을 주의하자.

```
[ -f "somefile" ]   : somefile 이 파일인지를 테스트 한다.
[ -x "/bin/ls" ]    : /bin/ls 가 실행파일인지를 검사한다.
[ -n "$var" ]       : $var 변수에 어떤 값이 대입되어 있는지를 검사한다.
[ "$a" = "$b" ]     : $a 와 $b 가 같은지 검사한다.
["$a" = "$b"]       : "[" 과 "]" 사이에 공백이 오지 않았음으로 잘못된 문장이다.
```

"man test" 를 이용해서 어떠한 test operator 이 있는지 확인 할수 있다.

```
#!/bin/sh
if [ "$SHELL" = "/bin/bash" ]
then
echo "your login shell is the bash (bourne again shell)"
else
echo "your login shell is not bash but $SHELL"
fi
```

$SHELL 은 환경변수로써 사용자의 로그인 쉘의 이름을 가지고 있다. 위의 스크립트는 $SHELL 의 값을 테스트 함으로써 사용자가 어떤 쉘을 사용하는지 알아내는 일을 한다.

6.4. 간단하게 표현하기

C 언어를 자주 사용해 본사람은 아래와 같은 문장에 익숙할 것이다.

```
[ -f "/etc/shadow" ] && echo "This computer uses shadow passwords"
```

위의 문장에서는 && 을 사용해서 if 문을 간단하게 표현하고 있다. 왼쪽 문장이 참이면 오른쪽을 실행하라는 것으로, /etc/shadow 라는 파일이 존재한다면 쉐도우 패스워드를 사용한다고 유저에게 알려주는 일을한다. || 를 사용하면 그반대의 경우이다. 간단한 사용예를 들어 보겠다.

```
#!/bin/sh
mailfolder=/var/spool/mail/james
```

```
[ -r "$mailfolder" ] || { echo "Can not read $mailfolder; exit 1;}
echo "$mailfolder has mail from : "
grep "^From " $mailfolder
```

위의 프로그램은 james 계정사용자의 메일파일을 검사해서 메일 파일을 읽을수 없으면 에러 메시지와 함께 종료 하고 그렇지 않으면 grep 을 써서 누구에게로 부터 메일이 왔는지를 계정 사용자에게 알려주는 일을한다.

case 는 if elif else 를 좀더 일반화 시킨 제어구조이다. if 문을 쓰더라도 여러번의 조건에 대해서 검사할수있지만 그럴경우 if elif 가 어지럽게 중첩 되는 결과를 보여줄것이다. 이럴때 case 를 사용하면 좀더 가독성과 유지가 용이한 코드를 만들어 낼수 있다. 즉 if elsif 를 간단하게 표현할수 있다. 이해를 쉽게 하기 위해서 특정파일이 어떠한 포멧의 압축파일인지를 알아내는 스크립트를 만들어 보도록 하자. 파일의 종류를 알기 위해서는 file 이란 명령을 쓰면 된다. 아래의 예제를 smartzip 이란 파일로 저장하도록 하자.

```
#!/bin/sh
ftype=`file "$1"`
case "$ftype" in
"$1: gzip compressed"*)
echo "gzip 압축";;
"$1: Zip archive"*)
echo "Zip 압축";;
*)
echo "FLE $1 can not be uncompressed with smartzip";;
easc
```

$1 변수는 프로그램의 첫번째 아규먼트를 저장하고 있는 변수다. ivmdemo.tar.gz 의 압축포맷을 알고 싶다면, smartzip ivmdemo.tar.gz 이라고 명령을 내리면 된다.

저 윗장에서 다룬적이 있는 덧셈 스크립트를 case 를 이용하여 사칙연산을 수행하도록 확장시켜보자. 물론 아래의경우 굳이 스크립트를 만들필요 없이 expr 만을 사용해도 동일한 작업이 가능하지만, 어디까지나 case 의 활용법 에 대한 이해를 위주로함이니 효율성, 가용성 기타 등등은 무시하고 넘어가기로 하자.

```
#!/bin/sh
add()
{
    result=`expr $1 + $2`
    echo "$1 + $2 = $result"
}
min()
{
    result=`expr $1 - $2`
    echo "$1 - $2 = $result"
}
div()
{
    result=`expr $1 / $2`
    echo "$1 / $2 = $result"
```

```
}
mul()
{
    result=`expr $1 \* $2`
    echo "$1 * $2 = $result"
}
#echo "$1, $2"
case $1 in
    "-") min $2 $3 ;;
    "+") add $2 $3 ;;
    "/") div $2 $3 ;;
    "*") mul $2 $3 ;;
esac
```

위의 스크립트는 첫번째 아규먼트로 연산자를 받아들이고 두번째 세번째 아규 먼트로 계산하고자 하는 숫자를 입력한다. "add - 1 3" 이런식으로 사용하면 된다. 주의할 점은 곱셈(*) 연산을 사용할 경우 "\" 등을 사용해서 "add * 1 3" 형식으로 써야한다는 점이다. 쉘상에서 * 는 와일드카드 확장을 실행하기 때문이다.

이번에는 select 제어문에 대해서 알아보자 select 는 interactive(대화형) 메뉴 프로그램을 짜는데 매우 간단한 방법을 제공해준다. 사용자가 어떤 OS를 가장 선호하는지 메뉴를 보고 그중 하나를 선택하는 프로그램을 만들어 보도록 하자.

```
#!/bin/sh
echo "What is your favourite OS ?"
select var in "Linux" "Free BSD" "Windows" "Solaris" "Other"
do
break
done
echo "You have selected $var"
```

위의 스크립트를 실행시키면 아래와 같은 메뉴가 뜨고 사용자의 입력을 요구하는 프롬프트가 대기 하게 될것이다.

```
What is your favourite OS ?
1) Linux
2) Free BSD
3) Windows
4) Solaris
```

원하는 운영체제의 번호(1 - 4) 를 선택하면 선택된 번호의 문자열이 var 변수에 저장된다. 1 을 입력하였다면 var 변수엔 Linux 가 저장 된다.

while 은 조건이 만족하는 동안 루프를 반복한다.

```
while ...
do
...
done
```

다음은 while 를 사용해서 1부터 10까지 출력하는 간단한 프로그램이다. #!/bin/sh

```
a=0
while [ $a -lt 10 ]
do
a=`expr $a + 1`
echo $a
done
```

bashsehll 에서의 for 문은 C의 for 문과는 사용에 있어서 차이가 난다. sehll 에있어서는 in 다음의 값들을 차례대로 변수에 입력하는 일을 한다.

```
#!/bin/sh
for var in A B C
do
echo "var is $var"
done
```

for 문을 이용한 좀더 유용한 스크립트를 만들어 보도록 하자. 아래의 스크립트는 배포판 CD에 해당 rpm이 있는지를 확인하고, 있다면 rpm 패키지의 정보를 보여주는 일을 한다. 아래의 내용을 showrpm 으로 저장하도록 하자.

```
#!/bin/sh
for rpmpackage in $*
do
if [ -r "$rpmpackage" ]
then
echo "================ $rpmpackage =============="
rpm -qi -p $rpmpackage
else
echo "ERROR: cannot read file $rpmpackage"
fi
done
```

위의 스크립트를 보면 $* 이라는 변수가 보일것이다. $*는 모든 아규먼트를 저장하는 변수이다.

6.5. Quoting

```
#!/bin/sh
echo $SHELL
echo "$SHELL"
echo '$SHELL'
```

위의 스크립트에서 1번째와 2번째의 경우 자신이 사용하는 쉘을 출력하지만 (아마도 /bin/bash) 3번째의 경우 $SHELL 자체를 출력하는걸 볼수 있을것이다. ' 를 사용하면 쉘이 사

용하는 특수문자(keyword)를 일반화 시켜서 사용할수 있다. 또한 백슬러쉬를 사용해서 와일드카드나 변수기호와 같은 특수한 문자를 일반화 시킬수도 있다.

```
echo \$SHELL
```

위에서 백슬러쉬를 사용함으로써 $ 의 특별한 의미를 제거시켜 버림으로써 $SHELL 이란 문자열을 출력하도록 만든다. 예를 들어서 $1,000 를 화면에 출력 시키려고 한다고 가정하자 이럴때 아래와 같이 써버리면

```
echo $1000
```

아무런 값도 출력되지 않음을 알수 있다. 왜냐면 쉘은 1000 앞에 $ 가 있음으로 이를 변수명으로 생각하고 이 변수명에 저장된 값을 echo 하려고 할것이기 때문이다. 우리가 원하는 값을 얻을려면 아래와 같이 코드를 수정해야 한다.

```
echo \$1000
```

6.6. 함수

여러분이 좀더 복잡한 프로그램을 만들다보면 함수의 필요성을 느끼게 될것이다. 함수를 사용함으로써, 좀더 이해하기 쉽고 단순한 프로그램을 만들수 있으며, 재사용을 용이하도록 만들수 잇다. 함수는 다음과 같이 선언한다.

```
functionname()
{
body
}
```

sehll 은 스크립트 언어이고 순차적으로 실행이 되므로 함수를 사용하기 전에 먼저 선언을 해주어야만 한다.

```
#!/bin/sh
help()
{
cat << HELP
xtitle bar -- change the name of an xterm, gnome-teminal or kde konsole
Usage: xtitlebar [-h] "string_for_titlebar"
OPTIONS: -h help text
EXAMPLE: xtitlebar "cvs"
HELP
exit 0
}
[ -z "$1" ] && help
[ "$1" = "-h" ] && help
```

함수를 이용한 간단한 덧셈 스크립트를 만들어 보자. 2개의 인자를 받아들이고 이를 더한후 출력하는 일을 한다.

```
#!/bin/sh
add()
{
result=`expr $1 + $2`
echo "$1 + $2 = $result"
}
add $1 $2
```

6.7. 명령행 인자(argument)

각 명령행 인자는 $* 과 $1, $2, $3, ... 등의 변수를 통해서 가져올수 있다. 그러나 이러한 명령행 인자들을 단순히 읽어들이는 것만으로는 -h 와 같은 명령행 옵션에 대한 내용은 다룰수 없다. 왜냐면 shell 에서는 -h 를 옵션이 아닌 인자로 취급하기 때문이다. 이를 처리하기 위해서는 약같의 기술이 필요하다. 보통 C에서는 getopt()와 같은 함수를 이용해서 옵션을 처리한다. 아래는 명령행 인자를 분석하는 쉘프로그램이다. 프로그램의 이름은 cmdparser 로 하자

```
#!/bin/sh
help()
{
cat << HELP
This is a generic command line parser demo.
Usage Example : cmdparser -l hello -f somefile1 somefile2
HELP
exit 0
}

while [ -n "$1" ]
do
case $1 in
-h) help; shift1;;
-f) opt_f=1;shift 1;;
-l) opt_l=$2;shift 2;;
--) shift;break;;
-*) echo "error : no such option $1. -h for help"; exit 1;;
*) break;
esac
done

echo "opt_f is $opt_f"
echo "opt_l is $opt_1"
echo "first arg is $1"
echo "2nd arg is $2"
```

shift 란 새로운 쉘명령이 나왔는데, 아규먼트를 하나씩 이동시키는 일을한다. 자세한 내용은 man bash 를 참조하기 바란다. 위의 스크립트를 cmdparser -l hello -f -- -somefile1 somefiel2 로 실행시켜보면 아래와 같은 결과가 나올것이다.

```
opt_f is 1
opt_l is hello
first arg is -somefile1
2nd arg is somefile2
```

위의 프로그램이 어떻게 동작하는지 알아보자, 위의 루프는 아규먼트가 검색될때까지 계속해서 순환하도록 되어 있으며 case 를 이용해서 아규먼트와 대응되는 값을 매칭시킨다. 만약에 매칭된 값을 찾았다면, 해당 명령어나 함수를 실행하고 shift 를 이용해서 필요한 만큼 아규먼트를 이동시킨다.

7. 예제

7.1. 일반적인 쉘프로그램의 구조

이번장에서는 그동안 배웠던 기본적인 내용들을 토대로, 실질적인 쉘프로그램을 만들도록 해보자. 모든 종류의 훌륭한 쉘스크립트는 도움말을 가지고 있으며, 아규먼트옵션을 파싱하는 일반적인 루틴을 가지고 있다. 이번에 새로만들 스크립트는 이러한 좋은 스크립 트가 가지는 루틴들을 포함하게 될것이다.

7.2. 바이너리를 10진수로 바꿔주는

이 스크립트는 바이너리를 숫자로 바꿔주는 일을한다. 간단한 산수 계산을 위해서 expr 을 사용하도록 한다.

```
#!/bin/sh
help()
{
cat << HELP
b2h -- convert binary to decimal
USAGE: b2h [-h] binarynum
OPTIONS: -h help text

EXAMPLE: b2h 111010
whill return 58
HELP
exit 0
}

error()
{
# print an error and exit
echo "$1"
exit 1
```

```
}

lastchar()
{
#return the last character of a string in $rval
if [ -z "$1" ]
then
rval=""
return
fi
numofchar=`echo -n "$1" | wc -c | sed 's/ //g'`
rval=`echo -n "$1" | cut -b $numofchar`
}

chop()
{
# remove the last character in string and return it in $rval
if [ -z "$1" ]
then
# empty string
rval=""
return
fi
# wc puts some space behind the output this is why we need sed:
numofchar=`echo -n "$1" | wc -c | sed 's/ //g' `
if [ "$numofchar" = "1" ]
then
# only one char in string
rval=""
return
fi
numofcharminus1=`expr $numofchar "-" 1`
# now cut all but the last char:
rval=`echo -n "$1" | cut -b 0-${numofcharminus1}`
}

while [ -n "$1" ]
do
case $1 in
-h) help;shift 1;; # function help is called
--) shift;break;; # end of options
-*) error "error: no such option $1. -h for help";;
*)  break;;
esac
done

# The main program
sum=0
weight=1
```

```
# one arg must be given:
[ -z "$1" ] && help
binnum="$1"
binnumorig="$1"

while [ -n "$binnum" ]; do
lastchar "$binnum"
if [ "$rval" = "1" ]
then
sum=`expr "$weight" "+" "$sum"`
fi
# remove the last position in $binnum
chop "$binnum"
binnum="$rval"
weight=`expr "$weight" "*" 2`
done

echo "binary $binnumorig is decimal $sum"
```

위의 프로그램은 이른바 2진수를 10진수로 바꾸어주는 프로그램이다. 만약에 위의 스크립트로 주어진 아규먼트가 1101 이라면, 2진수를 10진수로 바꾸는 계산법에 의해서 아래와 같은 결과가 나오게 될것이다.

```
1*2^3 + 1*2^2 + 0*2 + 1 = 6
```

위 프로그램의 분석은 여러분 각자에게 맡기겠다.

7.3. 디버깅

디버깅을 위한 가장간단한 방법은 echo 명령을 이용해서 결과값을 출력해보는 방법이다. - 이방법은 다른 모든 언어에도 공통적으로 적용된다. C는 printf, Perl은 print 등이 될것이다 - 당신은 echo 를 이용해서 변수값을 알아봄으로써, 어느부분에서 실수를 했는지를 알수 있다. 아마도 대부분의 쉘 프로그래머는 이러한 실수를 찾는데 전체 프로그램을 짜는 시간의 80% 정도를 보낼것이다. 쉘스크립트의 잇점은 이러한 테스트를 위해서 echo 문을 사용하는데, 다시 컴파일할 필요가 없어서, 시간을 절약할수 있다라는 것이다.

쉘스크립트를 디버깅 모드로 돌리고 싶다면 아래와 같이 하면 된다. 디버깅 하고자 하는 쉘스크립트의 이름은 strangescript 라고 하자.

```
sh -x strangescript
```

위와 같이 스크립트를 실행하면, 쉘이 실행되는 동안에 사용된 변수의 모든 값을 화면상에 출력시켜준다. 그럼으로 어느 곳에서 실수를 했는지 쉽게 찾을수 있다.

```
sh -n strangescript
```

위와 같이 "-n" 옵션을 이용하면 syntax 오류를 찾아낼수 있다. 위의 옵션에서는 syntax 오류 외의 다른 값들은 보여주지 않는다.

8. 효율적인 작업을 위해서

다른 여러가지 훌륭한 언어(C, Perl, Python)가 있음에도 불구하고 여전히 쉘을 사용하는 이유는 효율적인 작업을 위해서 일것이다. bash 는 사용자가 중복시간을 줄이고, 작업을 빠르고 효율적으로 할수 있도록 하기 위해서 history, 명령어치환 과 같은 여러가지 기능들을 제공한다.

8.1. 환경변수

bash(다른 쉘도 마찬가지)은 환경변수라는걸 사용한다. 환경변수는 쉘이 운용되는데 필요한 여러가지 값들을 가지고 있다. 대표적인 예로 SHELL 변수가 있다. 이변수에는 현재 로그인 유저가 사용하고 있는 쉘이 어떤건지에 대한 정보를 가지고 있다. 여러분이 현재 사용하는 쉘이 어떤건지 알고 싶다면 다음과 같이 하면된다.

```
[yundream@localhost yundream]$ echo $SHELL
/bin/bash
```

이밖에도 PATH, USER, TERM, PWD 등과 같은 많은 환경변수를 가지고 있다.

환경변수를 세팅하기 위해서는 export 라는 쉘 명령어를 사용한다.

```
[yundream@localhost yundream]$ export MY_NAME="yundream"
[yundream@localhost yundream]$ echo $MY_NAME
yundream
```

현재 설정되어 있는 모든 환경변수를 보기 위해서는 set 명령을 이용하면 된다. 이렇게 해서 설정된 환경변수는 현재 쉘에서 실행되는 모든 자식 프로세스에 상속 되게 된다.

8.2. history

history 는 이전에 내렸던 명령을 기억해서 다시 써먹으려고 할때 유용하다. 터미널에서 history 명령을 내림으로써 이전에 실행했던 명령어의 목록을 확인할수 있다.

```
[yundream@localhost yundream]$ history
    1   micq
2   exit
3   make clean
...
454 vi /home/httpd/cont/httpd.conf
455 history
```

히스토리가 저장되는 크기는 환경변수인 HISTSIZE 에 저장된다. 보통은 1000 라인정도로 세팅되어 있다. 히스토리에서 원하는 명령을 사용하고 싶을때는 "!n" 을 사용하면 된다. "n" 은 히스토리 번호이다. 즉 make clean 을 다시 실행하고 싶다면 "!3" 하면 된다.

"!" 는 히스토리 기능을 사용하기 위한 쉘 예약어이다. 이걸 사용하면 작업을 편하게 할수 있다. "!(문자열)" 하게 되면, 히스토리 목록중에서 (문자열)로 시작되는 가장 최근에 실행된 히스토리의 명령어를 불러와서 실행하게 된다. 위에서 454 번의 명령을 다시 실행하고 싶다면, !vi 하면 된다. 물론 !v 해도 관계없다.

히스토리는 grep 과 조합되어서 자주 사용한다.

```
[root@localhost /root]# history | grep configure
    8  ./configure --with-apache=../apache-1.3.22
   13  ./configure
   14  ./configure i686-linux
   15  ./configure --help
[root@localhost /root]# !8
./configure --with-apache=../apache-1.3.22
```

8.3. 최근명령어 치환

아래와 같은 상황을 가정해보자

```
[root@localhost /]# vp file1 file2
bash: vp: command not found
```

사실은 file1 을 file2 로 복사하는 작업을 하길 원했으나, cp 를 vp 로 치는 오류를 범했다. 이를 정정하기 위해서는 처음부터 다시치거나, 자판을 "v" 까지 움직 여서 이를 "c"로 교체 해야 할것이다. 이럴경우 "^"를 사용하면 작업을 좀더 빨리 할수 있다.

```
[root@localhost /]# ^v^c
cp file1 file2
```

"^[원본문자열]^[바뀔문자열]" 의 형식으로 사용하면, 가장 최근의 명령에서 원본문자열을 바뀔문자열로 치환하여 명령을 다시 실행하게 된다.

8.4. Prompt

Prompt 란 쉘이 사용자의 입력을 기다리고 있음을 유저에게 알려주기 위해서 사용한다. Prompt 를 잘 사용하게 되면 여러분이 어떤 호스트 에서 작업을 하는지, 혹은 몇시인지, 어떤 디렉토리에서 작업하고 있는지 등의 작업정보등을 얻을수 있다.

프롬프트 정보는 환경변수 PS1 에 저장된다. echo $PS1 해 보면 현재 설정된 프롬프트 정보를 볼수 있다. bash 는 효율적인 프롬프트 정보를 보여주기 위해서 몇개의 특수 문자 들을 제공하고 있다.

표 **2. Prompt** 예약어

\t	현재 시간을 HH:MM:SS 형식으로 보여준다.
\d	날자를 "요일 월 일" 형식으로 보여준다.
\s	쉘의 이름을 보여준다.
\w	현재 작업디렉토리를 완전경로로 보여준다.
\W	현재 작업디렉토리의 이름을 보여준다.
\u	현재 사용자의 사용자명
\h	호스트이름
\!	이 명령의 히스토리 번호
\nnn	8진수 nnn에 해당하는 문자
\[	비출력문자의 시퀀스를 시작한다.
\]	비출력문자의 시퀀스를 마친다.

간단한 예를 들어보자. 우리는 Prompt 상에 현재 시간과, 작업디렉토리 정보를 보여주길 원한다. 이럴때는 아래와 같이 하면 된다.

```
[root@localhost httpd]# export PS1="[\t \W]# "
[17:40:37 html]# echo "성공적으로 바꼈군요"
```

8.5. Prompt 전에 명령 실행시키기

이렇게 해서 프롬프트를 변경시키는방법을 알았다. 그런데, 어떤 한팀 의 경우 한팀의 타이틀바 제목이 시시각각 바뀌는것을 본적이 있을 것이다. 이건 어떻게 하는것일까. ?

이건 bash 쉘의 환경변수인 PROMPT_COMMAND 를 사용하여 가능하다. PROMPT_COMMAND 에는 특정 명령어(스트립트 혹은 실행파일)가 값으로 들어가 있는데, 사용자에게 프롬프트가 떨어지기 전에 변수에 있는 명령이 실행된다.

```
[root@localhost /root]# export PROMPT_COMMAND="date"
월 11월 19 17:51:03 KST 2001
[root@localhost /root]# echo "hello world"
hello world
월 11월 19 17:51:19 KST 2001
[root@localhost /root]#
```

위의 예에서 보다시피 사용자 프롬프트가 떨어질때 마다, 그전에 "date" 명령이 실행됨을 알 수 있다. 이 명령을 약간 변경하면 타이틀바가 그때그때 바뀌도록 설정 할수 있다.

```
[root@localhost /home]# export PROMPT_COMMAND='echo -
ne "\033]0;${USER}@${HOSTNAME}: ${PWD}\007"'
```

위의 경우 디렉토리가 변경될때 마다 작업디렉토리가 타이틀바에 표시 가된다. 위의 내용을 /etc/bashrc 정도에 입력해 놓으면 창을 띄울때 마다 환경변수의 내용이 적용될것이다.

하지만 단순히 사용자에게 시각적으로 그럴듯하게 보여주는것 외에도 PROMPT_COMMAND 를 응용하면 좀더 다양한 일들을 할수 있다. 다음장에서 이에 대해서 간단하지만 유용한 한가지 예를 들어 보도록 하겠다.

8.6. PROMPT_COMMAND 를 이용한 사용자 명령어 추적

요즘 컴터업계 최대의 화두는 "보안"이다. 이런 대세에 맞추어서 사용자가 현재 무슨 명령을 사용중에 있는지 실시간으로 확인할수 있는 간단한 보안 프로그램? 을 작성해보도록 하겠다.

보통 이러한 사용자 명령을 실시간으로 추적하는 프로그램을 보안쉘 이라고 한다. - 물론 실지 보안쉘에는 이외에도, 파일보안, 디렉토리 보안, 프로세스보안 등과 같은 다양한 기능이 들어가지만 여기서는 사용자의 작업내용을 실시간으로 확인할수 있는 기능만 구현하겠다 - 보통 보안쉘을 만드는데는 많은 시간과 인내심이 필요하지만, 쉘의 특징을 조금만 알고 있다면, 시간과 노력을 단축시킬수가 있다.

PROMPT_COMMAND 를 이용하는 방법은 매우 간단하다. 사용자의 프롬프트가 떨어졌을 때, 사용자의 이름, 사용자의 가장 최근의 명령어등을 가져와서 이를 파일로 만들거나 소켓을 통해서, 관리자에게 통보하도록 하면 된다. 쉘에서는 물론 history라는걸 제공하긴 하지만, history 는 실시간이 아니다. 일단은 쉘의 메모리에서 관리하고 있다가, 로그아웃 하면 그때 파일로 저장이 된다. 즉 로그아웃 하기 전까지는 무슨일을 하는지 통 알수가 없게 된다. 그리고 history 명령을 이용해서 편집 가능하다. 다음과 같은이름으로 moniter.sh 란 간단한 스크립트 파일을 작성한다.

```
LAST_COMMAND=`tail -1 /root/dump`
CURRENT_COMMAND=`history 1`

if [ "$LAST_COMMAND" != "$CURRENT_COMMAND" ]
then
echo "$CURRENT_COMMAND" >> /root/dump
fi
```

위의 스크립트를 적당한 디렉토리에 복사한다음에, 각 사용자가 로그인할때 환경변수로 초기화 시키면 된다.

```
export PROMPT_COMMAND="/usr/bin/moniter.sh"
```

물론 위의 프로그램에는 여러가지 부족한점이 많다. 마지막 명령이 화일로 저장되며, 또한 사용자가 쉘의 특성을 잘알고 있고, 여기에 주의를 기울인다면 환경변수를 분석해서 이를 조작할수 있다. 이는 여러가지 방법을 통해서 해결할수 있지만, 이는 쉘스크립팅의 범위를 벗어나는 것임으로, 간단히 이러한 일을 할수도 있다라는 것만 이해해 두기로 하자. 이 글을 읽는 여러분이 좋은 방법을 생각해 보기 바란다. (물론 가장 좋은 방법은 쉘 쏘쓰를 수정하는 거다. 생각만큼 복잡하지 않으니 연구해 보시길..)