

교과목 : 디지털정보처리

4장. Shell 제대로 사용하기

2021학년도 1학기
옥수열



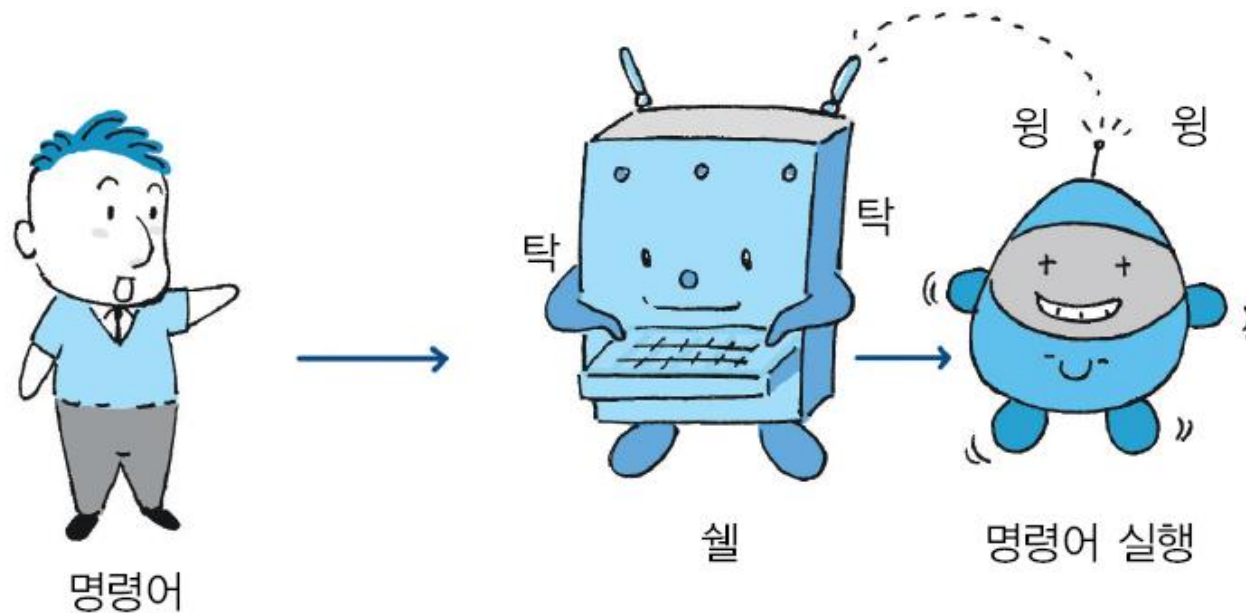
Computer & Ai

Department of AI Engineering

셸(Shell)이란 무엇인가?

셸의 역할

- 셸은 사용자와 운영체제 사이에 창구 역할을 하는 소프트웨어
- 명령어 처리기(command processor)
- 사용자로부터 명령어를 입력받아 이를 처리한다



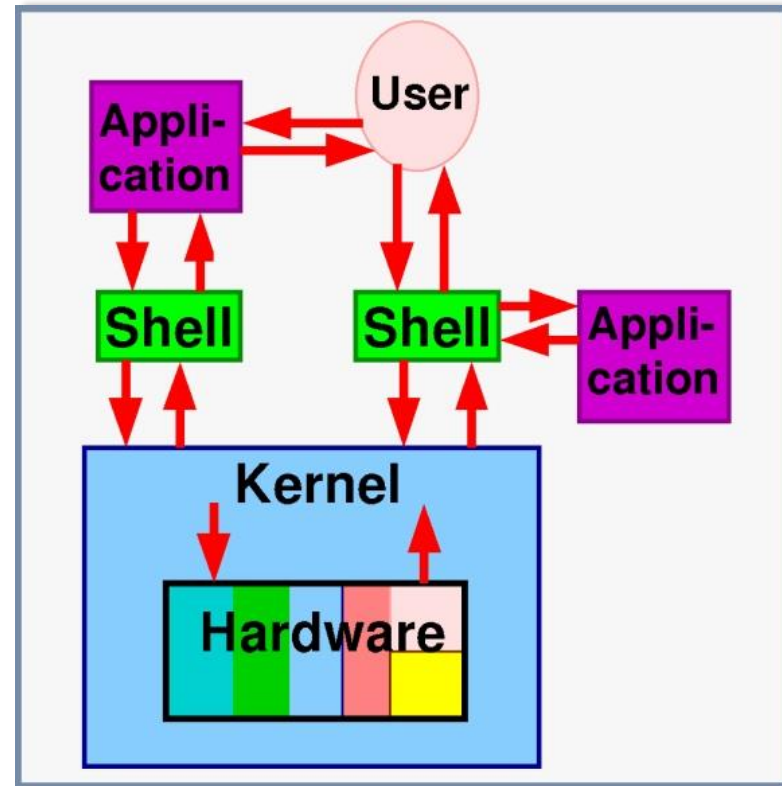
셸

□ 사용자가 입력한 명령을 해석하고 실행하는 명령 해석기 (Command Interpreter)

- 사용자가 처음 수행할 수 있는 특수 프로그램

□ UNIX 셸 종류

- sh: Stephen Bourne, 기본 표준 셸, \$ prompt
- csh: Bill Joy, C와 닮은 꼴, % prompt
- ksh: David Korn, sh과 호환, \$ prompt
- zsh: Paul Falstad, ksh의 업그레이드
- bash: Brian Fox, GNU free software, sh과 호환, csh과 ksh의 장점 수용, \$ prompt
- 그 외에도 많음.



셸(Shell)의 기능과 종류

셸의 종류

- 유닉스/리눅스에서 사용 가능한 셸의 종류

셸의 종류	셸 실행 파일
Bourne 셸	/bin/sh
Korn 셸	/bin/ksh
C 셸	/bin/csh
Bash 셸	/bin/bash
tcsh	/bin/tcsh



셸(Shell)의 기능과 종류

■ 본 셸(Bourne shell)

- 유닉스 V7에 처음 등장한 최초의 셸 (유닉스에서 기본 셸로 사용됨)
- 개발자의 이름인 스티븐 본(Stephen Bourne)의 이름을 따서 본 셸이라고 함
- 본 셸의 명령 이름은 sh임
- 초기에 본 셸은 단순하고 처리 속도가 빨라서 많이 사용되었고, 지금도 시스템 관리 작업을 수행하는 많은 셸 스크립트는 본 셸을 기반으로 하고 있음
- 히스토리, 에일리어스, 작업 제어 등 사용자의 편의를 위한 기능을 제공하지 못해 이후에 다른 셸들이 등장

■ C 셸(C shell)

- 캘리포니아대학교(버클리)에서 빌 조이(Bill Joy)가 개발,
- BSD 계열의 유닉스에 많이 사용됨
- 셸의 핵심 기능 위에 C언어의 특징을 많이 포함함
- 셸 스크립트 작성을 위한 구문 형식이 C 언어와 같아 C 셸이라는 이름을 가지게 되었음
- C 셸의 명령 이름은 csh
- 본 셸에는 없던 에일리어스나 히스토리 같은 사용자 편의 기능을 포함
- 최근 이를 개선한 tcsh이 개발되어 사용됨

셸(Shell)의 기능과 종류

■ 콘 셸(Korn shell)

- 1980년대 중반 AT&T 벨연구소의 데이비드 콘(David Korn)이 콘 셸을 개발
- 유닉스 SVR 4에 포함되어 발표
- C 셸과 달리 본 셸과의 호환성을 유지하고 히스토리, 에일리어스 기능 등 C 셸의 특징도 모두 제공하면서 처리 속도도 빠름
- 콘 셸의 명령 이름은 ksh

■ 배시 셸(bash shell) (Bash: Bourne again shell)

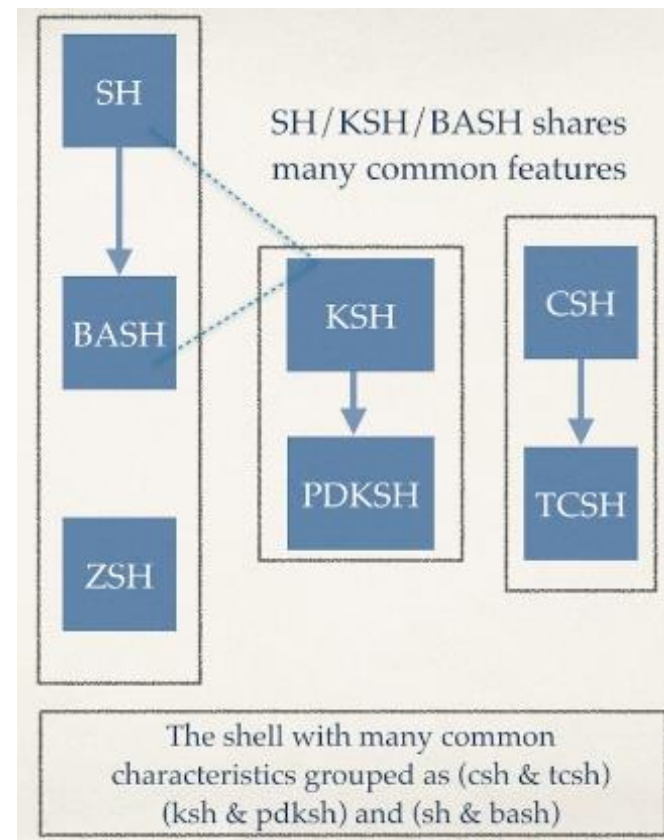
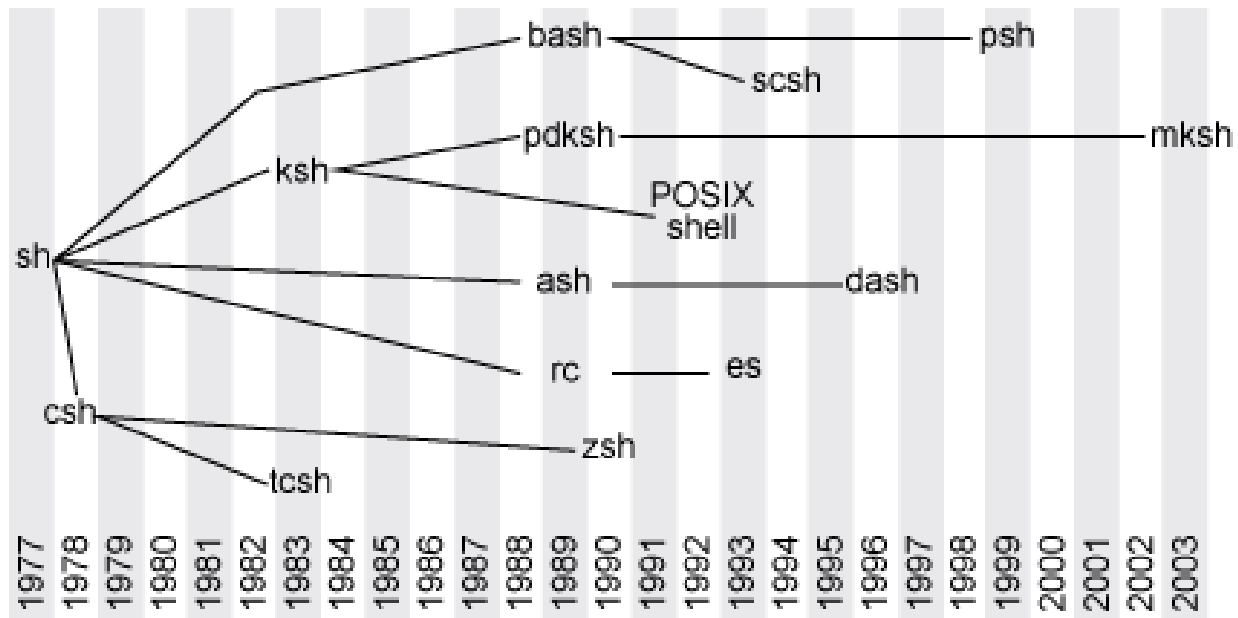
- 본 셸을 기반으로 개발된 셸로서 1988년 브레인 폭스(Brain Fox)가 개발
- 본 셸과 호환성을 유지하면서 C 셸, 콘 셸의 편리한 기능도 포함
- 배시 셸의 명령 이름은 bash, 명령어의 구문은 본 셸 명령어 구문을 확장함
- 배시 셸의 모든 버전은 GPL 라이선스에 의거하여 자유롭게 사용 가능
- 리눅스의 기본 셸로 제공되고 있어 리눅스 셸로도 많이 알려짐

■ 대시 셸(dash shell)

- 본 셸을 기반으로 개발된 셸로 포직스(POSIX) 표준을 준수하면서 보다 작은 크기로 개발
- 암키스트 셸(ash, Almquist Shell)의 NetBSD 버전으로 1997년 초에 허버트 슈가 리눅스에 이식
- 우분투 6.10부터 본 셸 대신 대시 셸을 사용

셸(Shell)의 기능과 종류

셸 History



셸의 기능

■ 명령어 해석기 기능

- 사용자와 커널 사이에서 명령을 해석하여 전달하는 해석기(interpreter)와 번역기(translator) 기능
- 사용자가 로그인하면 셸이 자동으로 실행되어 사용자가 명령을 입력하기를 기다림 → 로그인 셸
- 로그인 셸은 /etc/passwd 파일에 사용자별로 지정
- 프롬프트: 셸이 사용자의 명령을 기다리고 있음을 나타내는 표시

■ 프로그래밍 기능

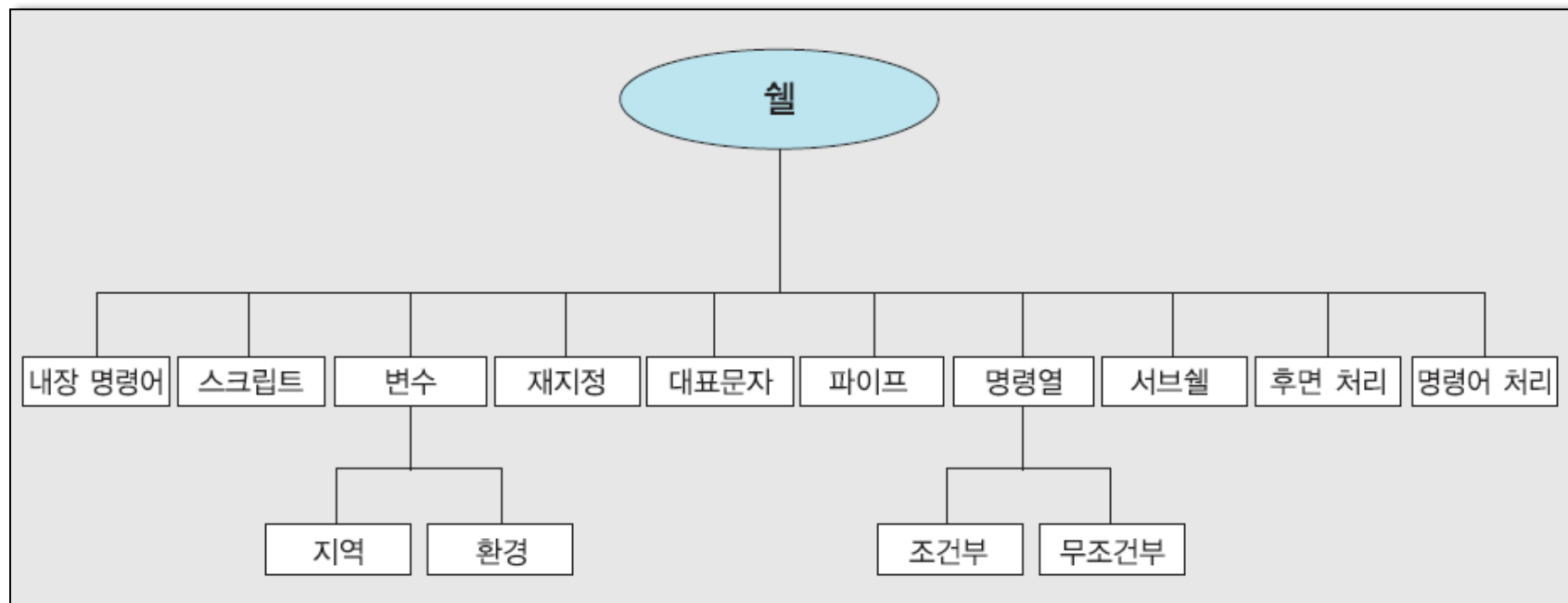
- 셸은 자체 내에 프로그래밍 기능이 있어 반복적으로 수행하는 작업을 하나의 프로그램으로 작성 가능
- 셸 프로그램을 셸 스크립트

■ 사용자 환경 설정 기능

- 사용자 환경을 설정할 수 있도록 초기화 파일 기능을 제공
- 초기화 파일에는 명령을 찾아오는 경로를 설정하거나, 파일과 디렉터리를 새로 생성할 때 기본 권한을 설정하거나, 다양한 환경 변수 등을 설정

❖ 셸의 주요 기능

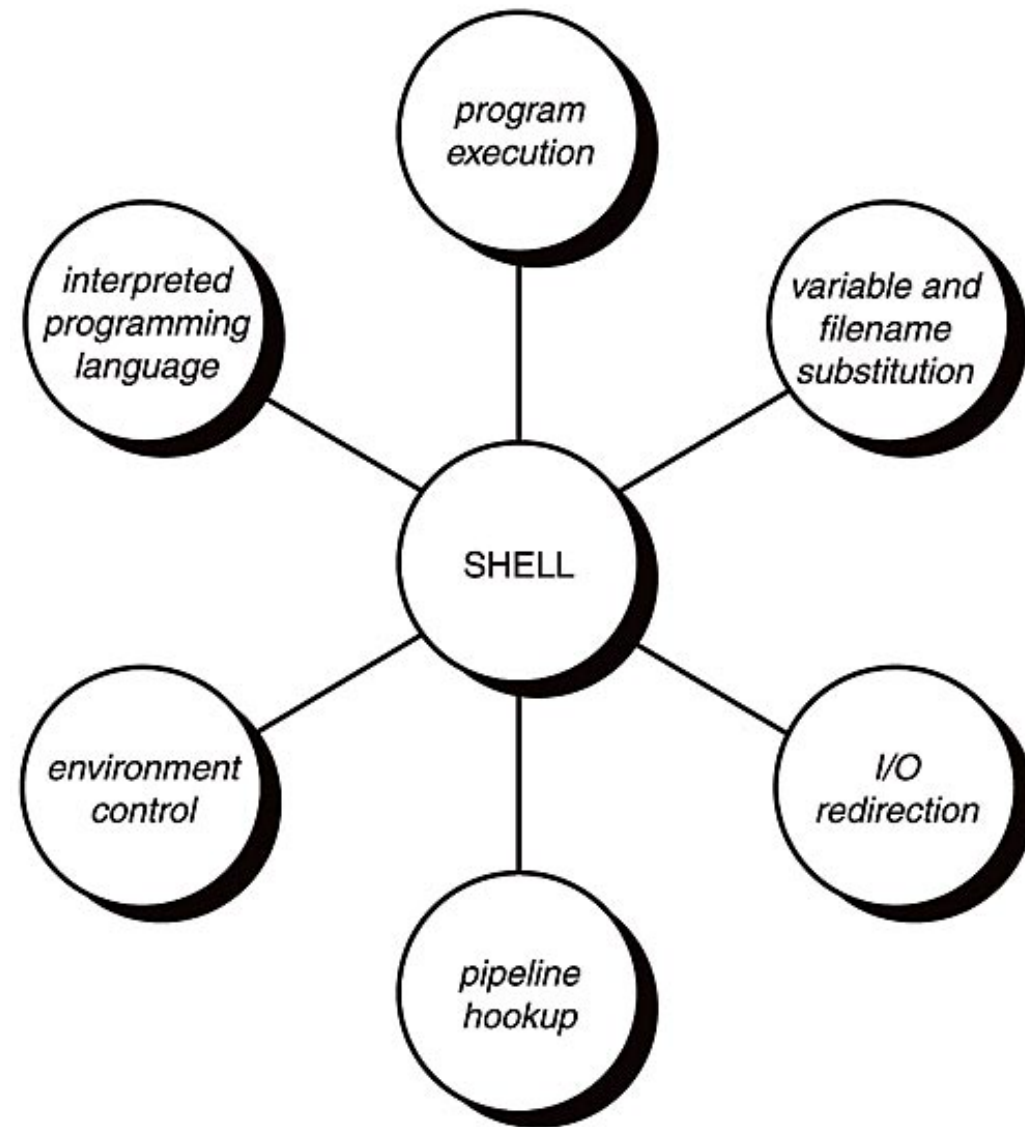
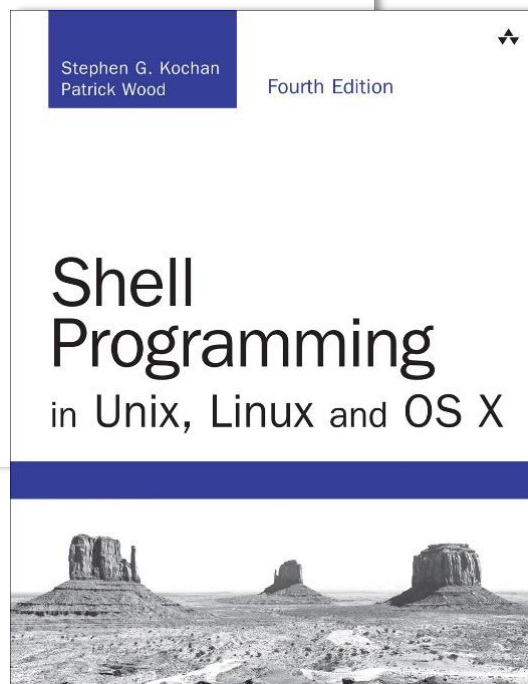
- 명령어 처리 (명령어 해석기)
 - 사용자가 입력한 명령을 해석하고 적절한 프로그램을 실행
- 시작 파일 (사용자 환경 설정)
 - 로그인할 때 실행되어 사용자별로 맞춤형 사용 환경 설정
- 스크립트 (프로그래밍)
 - 셸 자체 내의 프로그래밍 기능



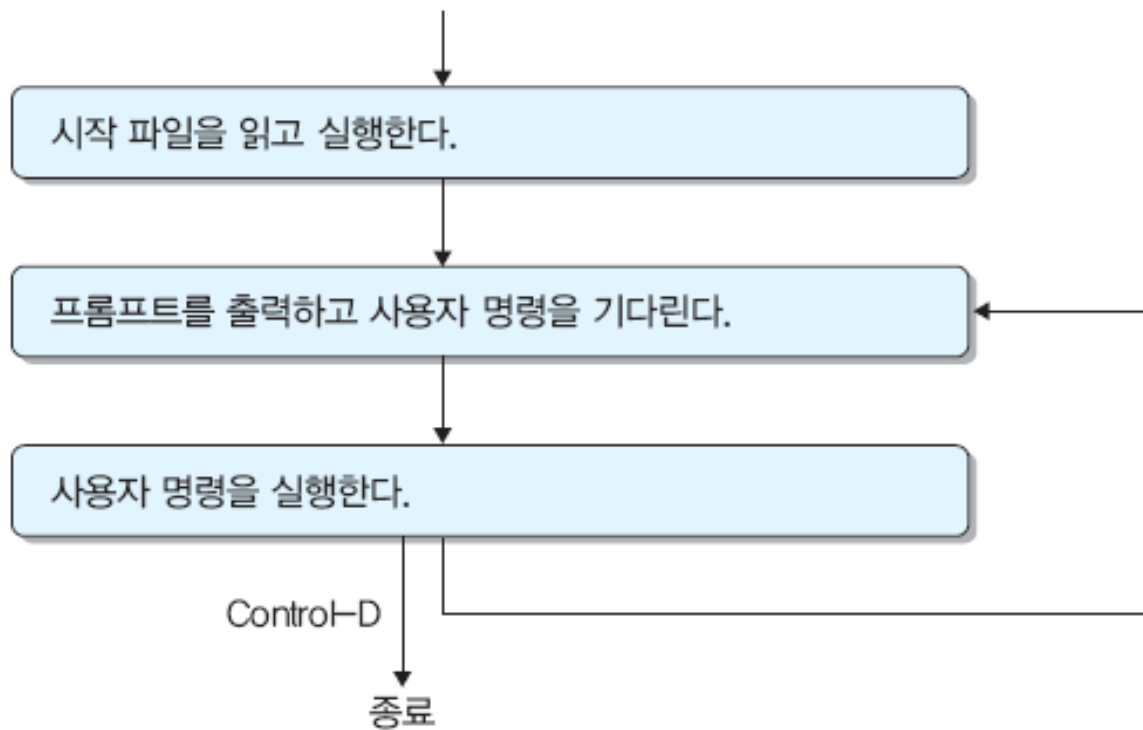
셸(Shell)의 기능과 종류

□ 셸의 공통 기능

- 다수의 내장 명령어 제공
- 메타문자
- 입력 / 출력 / 오류의 재지정 (redirection)
- 파이프라인
- 명령어 대치 / 완성 / 편집
- 환경 변수와 지역변수
- 후면 처리
- 하위셸 (subshell) 생성
- 셸 프로그래밍
- 작업 제어



셸의 실행 절차



셸의 환경변수

- 환경변수 설정법

\$ 환경변수명=문자열
환경변수의 값을 문자열로 설정한다.

- 예
\$ TERM=xterm
\$ echo \$TERM
xterm

- 환경변수 보기

```
$ env
TERM=xterm
SHELL=/bin/sh
GROUP=cs
USER=chang
HOME=/home/chang
PATH=/usr/local/bin:/usr/bin: ...
...
```

- 사용자 정의 환경 변수

```
$ MESSAGE=hello
$ export MESSAGE
```

셸의 시작파일 (start-up file)

- 시작 파일
 - 셸마다 시작될 때 자동으로 실행되는 고유의 시작 파일
 - 주로 사용자 환경을 설정하는 역할을 하며
 - 환경설정을 위해서 환경변수에 적절한 값을 설정한다.
- 시스템 시작 파일
 - 시스템의 모든 사용자에게 적용되는 공통적인 설정
 - 환경변수 설정, 명령어 경로 설정, 환영 메시지 출력, ...
- 사용자 시작 파일
 - 사용자 홈 디렉터리에 있으며 각 사용자에게 적용되는 설정
 - 환경변수 설정, 프롬프트 설정, 명령어 경로 설정, 명령어 이명 설정, ...

셸 시작

□ 로그인

- 계정 생성 시 기본 셸이 지정된다.
 - 로그인 셸 확인: `echo $SHELL`
 - 현재 수행 중인 셸 확인: `ps`

□ 셸 변경

- 해당 셸의 이름을 입력
 - `sh`, `csh`, `ksh`, `tcsh`, `zsh`, ...
 - 셸 프로그램이 설치되어 있지 않을 경우 실행되지 않는다.
- 빠져 나올 때는 `exit`

셸(Shell)의 기능과 종류

셸 시작 파일 (start-up file)

셸의 종류	시작파일 종류	시작파일 이름	실행 시기
본 셸	시스템 시작파일	/etc/profile	로그인
	사용자 시작파일	~/.profile	로그인
Bash 셸	시스템 시작파일	/etc/profile	로그인
	사용자 시작파일	~/.bash_profile	로그인
	사용자 시작파일	~/.bashrc	로그인, 서버셸
	시스템 시작파일	/etc/bashrc	로그인
C 셸	시스템 시작파일	/etc/.login	로그인
	사용자 시작파일	~/.login	로그인
	사용자 시작파일	~/.cshrc	로그인, 서버셸
	사용자 시작파일	~/.logout	로그아웃

셸 시작 파일 예시

- **.profile**

```
PATH=$PATH:/usr/local/bin:/etc
```

```
TERM=vt100
```

```
export PATH TERM
```

```
stty erase ^
```

- **시작 파일 바로 적용**

```
$ . .profile
```


셸(Shell)의 기능과 종류

로그인 셸(login shell)

- 로그인 하면 자동으로 실행되는 셸
- 보통 시스템관리자가 계정을 만들 때 로그인 셸 지정

```
/etc/passwd _____  
root:x:0:1:Super-User:/:/bin/bash  
...  
chang:x:109:101:Byeong-Mo Chang:/user/faculty/chang:/bin/bash
```

로그인 셸 변경

- 셸 변경
\$ csh
%
...
% exit
\$
- 로그인 셸 변경
\$ chsh
Changing login shell for chang
Old shell : /bin/sh
New shell : /bin/csh
\$ logout

login : chang
passwd:
%

Bash

□ Bash (Bourne Again Shell)

- GNU 표준 셸 → 리눅스 표준 셸
- 1988년 처음 배포
- 현재 버전 3.2

□ Bash 정보

- 다운로드: <http://www.gnu.org/software/bash>
 - 리눅스에 bash가 설치되어 있지 않은 경우 또는 upgrade
- 매뉴얼
 - 온라인: <http://www.gnu.org/software/bash/manual>
 - 명령행에서 help 명령으로 도움말 기능 제공

셸(Shell)의 기능과 종류

Bash

□ More meta-characters

메타문자	의미	예
?	문자 하나	a? - ab, ac, a3, ...
*	문자 여러 개	c*t - cat, chat, come at, ...
[set]	set에 있는 하나의 문자	[abc] - abc 중 하나 [a-z] - 모든 소문자 중 하나 [-a-z] - -와 모든 소문자 중 하나
[!set]	set에 없는 하나의 문자	[!0-9] - 숫자가 아닌 문자 [0-9!] - 모든 숫자와 !
{ s1,s2,... }	s1 and s2 and ...	b{ed,olt,ar}s - beds, bolts, bars ls *. {c,h,o} - 확장자가 .c, .h, .o인 모든 파일 리스트

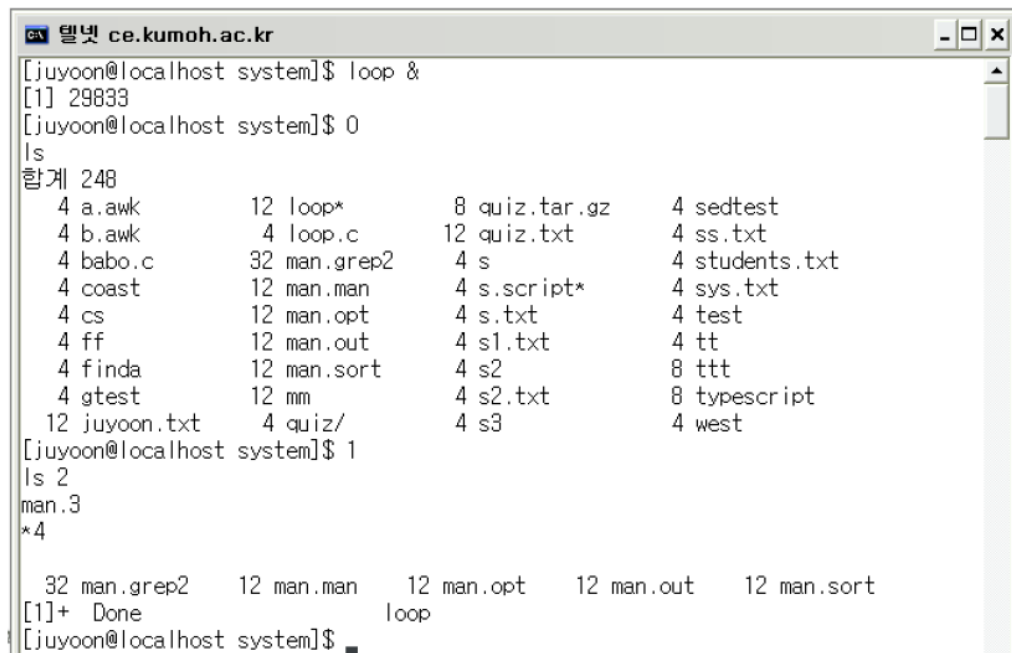
□ 셸에서 의미를 가지는 특수 문자

메타문자	의미	메타문자	의미
~	홈 디렉터리	\$	변수
`	명령 대체	&	백그라운드 작업
#	Comment	*?	와일드카드
()	하위 셸 시작/종료	\	문자 그대로
	파이프	[]	문자 집합
{ }	명령 블록	;	셸 명령 분리
'	강한 인용부호	"	약한 인용부호
<	입력 재지정	>	출력 재지정
/	경로명 분리	!	논리 NOT

Bash

□ 백그라운드 작업 &

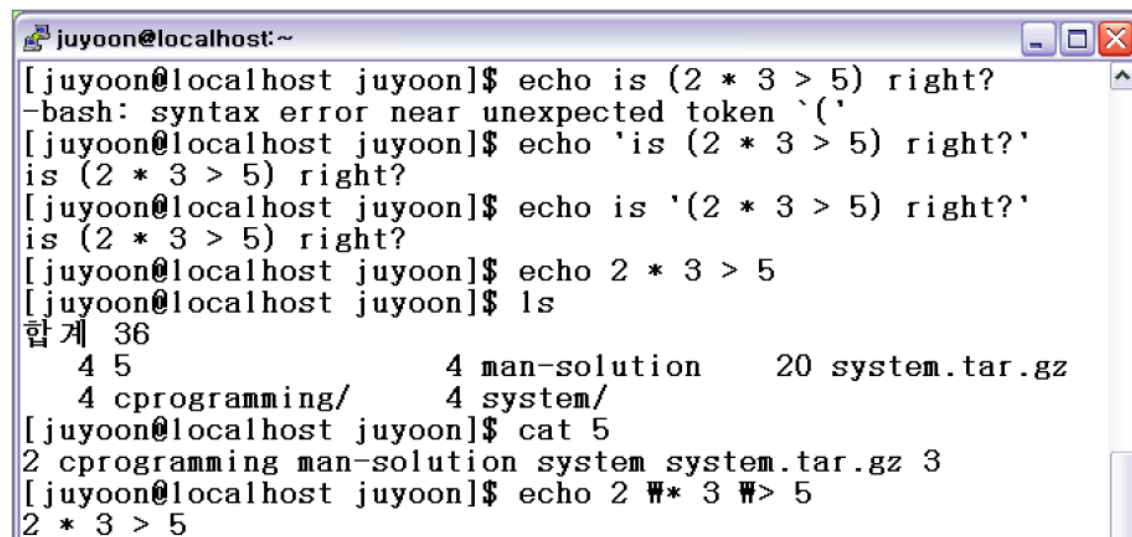
- 여러 작업을 동시에 수행
- 입력 없이 시간이 많이 걸리는 작업에 편리



```
ce.kumoh.ac.kr
[juhyon@localhost system]$ loop &
[1] 29833
[juhyon@localhost system]$ 0
ls
합계 248
 4 a.awk      12 loop*      8 quiz.tar.gz  4 sedtest
 4 b.awk      4 loop.c      12 quiz.txt    4 ss.txt
 4 babo.c     32 man.grep2   4 s            4 students.txt
 4 coast      12 man.man     4 s.script*    4 sys.txt
 4 cs         12 man.opt     4 s.txt        4 test
 4 ff         12 man.out     4 s1.txt       4 tt
 4 finda      12 man.sort    4 s2           8 ttt
 4 gtest      12 mm         4 s2.txt       8 typescript
 12 juhyon.txt 4 quiz/       4 s3           4 west
[juhyon@localhost system]$ 1
ls 2
man.3
*4
32 man.grep2 12 man.man 12 man.opt 12 man.out 12 man.sort
[1]+ Done loop
[juhyon@localhost system]$
```

□ 특수 문자를 일반 문자로 취급하려면?

- 인용부호 사용
- \ 사용



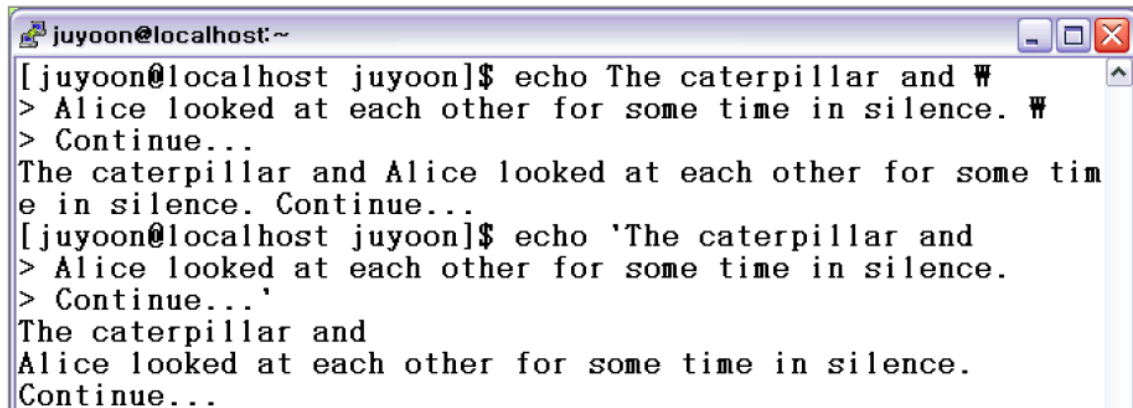
```
juhyon@localhost:~
[juhyon@localhost juhyon]$ echo is (2 * 3 > 5) right?
-bash: syntax error near unexpected token '('
[juhyon@localhost juhyon]$ echo 'is (2 * 3 > 5) right?'
is (2 * 3 > 5) right?
[juhyon@localhost juhyon]$ echo is `(2 * 3 > 5) right?'
is (2 * 3 > 5) right?
[juhyon@localhost juhyon]$ echo 2 * 3 > 5
2 * 3 > 5
[juhyon@localhost juhyon]$ ls
합계 36
 4 5 4 man-solution 20 system.tar.gz
 4 cprogramming/ 4 system/
[juhyon@localhost juhyon]$ cat 5
2 cprogramming man-solution system system.tar.gz 3
[juhyon@localhost juhyon]$ echo 2 \* 3 \> 5
2 * 3 > 5
```

- 약한 인용부호 " : \$, ?, \ 제외하고 일반 문자로 해석

Bash

□ 명령행의 계속

- '\n'을 일반 문자로 취급하기
- 행의 끝에 \ 사용
 - \n을 완전히 무시하고 한 줄로 연결
- 인용부호(')로 연결
 - \n을 명령의 끝이 아닌 하나의 문자로 취급



```
jyoon@localhost:~  
[jyoon@localhost jyoon]$ echo The caterpillar and \  
> Alice looked at each other for some time in silence. \  
> Continue...  
The caterpillar and Alice looked at each other for some tim  
e in silence. Continue...  
[jyoon@localhost jyoon]$ echo 'The caterpillar and  
> Alice looked at each other for some time in silence.  
> Continue...'  
The caterpillar and  
Alice looked at each other for some time in silence.  
Continue...
```

□ 컨트롤 키

컨트롤키	stty 명	기능
^C	intr	현재 명령 중지
^D	eof	입력의 끝
^\	quit	^C가 동작하지 않을 경우 현재 명령 중지
^S	stop	화면 출력 정지
^Q	start	화면 출력 다시 시작
DEL, ^?	erase	마지막 문자 삭제
^U	kill	전체 명령행 삭제
^Z	susp	현재 명령 일시 중단

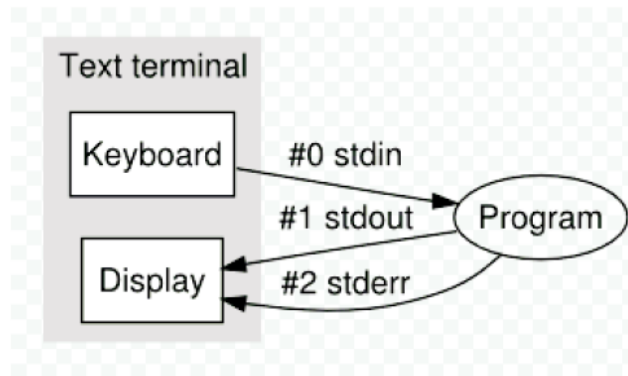
Redirection

□ 표준 입출력

- 데이터가 유닉스 시스템에 저장되거나 전송되는 표준 방식
- 다양한 입출력 기기를 추상화하고 공통된 처리 방식을 사용하도록 최초로 시도

□ 세 가지 표준 파일

- stdin (0) : 표준 입력
- stdout (1) : 표준 출력
- stderr (2) : 표준 오류 출력



Redirection

□ Redirection

- 셸에서 제공하는 편의 기능
- 표준 입력 또는 출력을 파일 등 다른 입출력 장치로 보냄.
- 메타문자를 사용해서 redirection을 지시한다.

기 호	의 미
>	출력 redirection
>!	출력 redirection, csh의 noclobber 옵션을 중복 정의
>>	기존 파일에 출력을 추가
>>!	기존의 파일에 출력을 추가, csh의 noclobber 옵션을 중복 정의하고 파일이 존재하지 않으면 파일 생성
	다른 명령으로 파이프 출력
<	입력 redirection
<<word	word로 시작하는 줄의 앞줄까지 표준 입력으로 받아들임
>&	표준 출력과 표준 에러를 파일로 redirection한다.
>>&	표준 출력과 표준 에러를 파일에 추가한다.

Redirection

□ Redirection을 이용한 텍스트 파일 편집

```
jyoon@linda: ~/test/system
jyoon@linda:~/test/system$ cat > myfile
This is a test of redirection.
읽을 수 있어?
jyoon@linda:~/test/system$ cat myfile
This is a test of redirection.
읽을 수 있어?
jyoon@linda:~/test/system$
```

- 입력 끝 (EOF) 표시는 ^d
- >>로 redirection: 기존 파일에 추가

```
jyoon@linda: ~/test/system
jyoon@linda:~/test/system$ cat >> myfile
이건 덧붙이는 거야. 끝낼 때는 ^D
jyoon@linda:~/test/system$ cat myfile
This is a test of redirection.
읽을 수 있어?

이건 덧붙이는 거야. 끝낼 때는
jyoon@linda:~/test/system$
```

□ 입력

```
jyoon@linda: ~/test/cprogramming
jyoon@linda:~/test/cprogramming$ getchar < getchar.c
#include <stdio.h>

int main(void)
{
    char ch=0;
    while (ch != EOF) {
        ch = getchar();
        putchar(ch);
    }
    return 0;
}
jyoon@linda:~/test/cprogramming$
```

□ 입출력 혼용

```
jyoon@linda: ~/test/cprogramming
jyoon@linda:~/test/cprogramming$ getchar < getchar.c > get.c
jyoon@linda:~/test/cprogramming$ cat get.c
#include <stdio.h>

int main(void)
{
    char ch=0;
    while (ch != EOF) {
        ch = getchar();
        putchar(ch);
    }
    return 0;
}
jyoon@linda:~/test/cprogramming$
```


셸(Shell)의 기본 사용법

후면 처리 예

- `$ (sleep 100; echo done) &`
[1] 8320
- `$ find . -name test.c -print &`
[2] 8325

후면 처리 확인

- 사용법

`$ jobs [%작업번호]`

후면에서 실행되고 있는 작업들을 리스트 한다. 작업 번호를 명시하면 해당 작업만 리스트 한다.

- 예

`$ jobs`

[1] + Running [sleep 100; echo done]

[2] - Running find . -name test.c -print

`$ jobs %1`

[1] + Running [sleep 100; echo done]

셸(Shell)의 기본 사용법

후면 작업을 전면 작업으로 전환

- 사용법

\$ fg %작업번호

작업번호에 해당하는 후면 작업을 전면 작업으로 전환

- 예

```
$ (sleep 100; echo DONE) &
```

```
[1] 10067
```

```
$ fg %1
```

```
[ sleep 100; echo DONE ]
```

셸(Shell)의 기본 사용법

출력 재지정(output redirection)

- 사용법

\$ 명령어 > 파일

명령어의 표준출력을 모니터 대신에 파일에 저장한다.

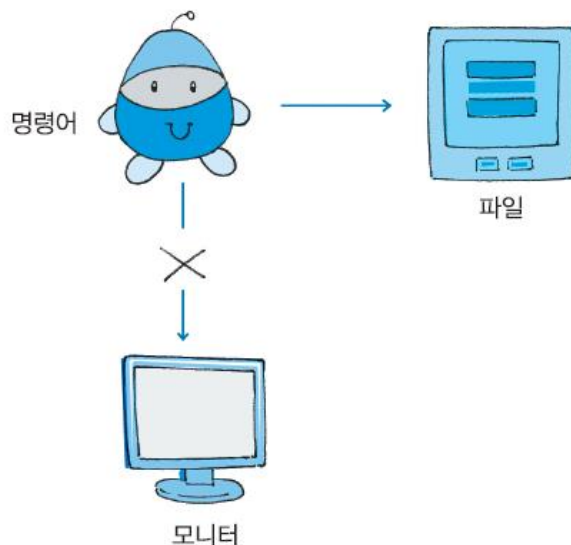
- 예

```
$ who > names.txt
```

```
$ cat names.txt
```

```
$ ls / > list.txt
```

```
$ cat list.txt
```



출력 재지정 이용: 간단한 파일 만들기

- 사용법

\$ cat > 파일

표준입력 내용을 모두 파일에 저장한다. 파일이 없으면 새로 만든다.

- 예

```
$ cat > list1.txt
```

```
Hi !
```

```
This is the first list.
```

```
^D
```

```
$ cat > list2.txt
```

```
Hello !
```

```
This is the second list.
```

```
^D
```

셸(Shell)의 기본 사용법

두 개의 파일을 붙여서 새로운 파일 만들기

- 사용법

```
$ cat 파일1 파일2 > 파일3
```

파일1과 파일2의 내용을 붙여서 새로운 파일3을 만들어 준다.

- 예

```
$ cat list1.txt list2.txt > list3.txt
```

```
$ cat list3.txt
```

```
Hi !
```

```
This is the first list.
```

```
Hello !
```

```
This is the second list.
```

셸(Shell)의 기본 사용법

입력 재지정(input redirection)

- 사용법

\$ 명령어 < 파일

명령어의 표준입력을 키보드 대신에 파일에서 받는다.

- 예

```
$ wc < list1.txt
3 13 58 list1.txt
```

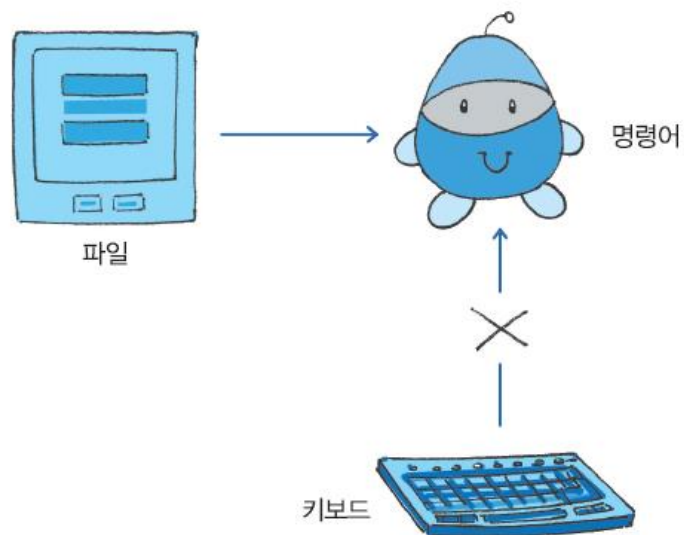
- 참고

```
$ wc
```

```
...
```

```
^D
```

```
$ wc list1.txt
```



문서 내 입력(here document)

- 사용법

\$ 명령어 << 단어

...

단어

명령어의 표준입력을 키보드 대신에 단어와 단어 사이의 입력 내용으로 받는다.

- 예

```
$ wc << END
hello !
word count
END
2      4      20
```

셸(Shell)의 기본 사용법

오류 재지정

- 사용법

\$ 명령어 2> 파일

명령어의 표준오류를 모니터 대신에 파일에 저장한다.

- 명령어의 실행결과

- 표준출력(standard output): 정상적인 실행의 출력
- 표준오류(standard error): 오류 메시지 출력

- 사용법

\$ ls -l /bin/usr 2> err.txt

\$ cat err.txt

ls: cannot access /bin/usr: No such file or directory

셸(Shell)의 기본 사용법

파이프

- 로그인 된 사용자들을 정렬해서 보여주기

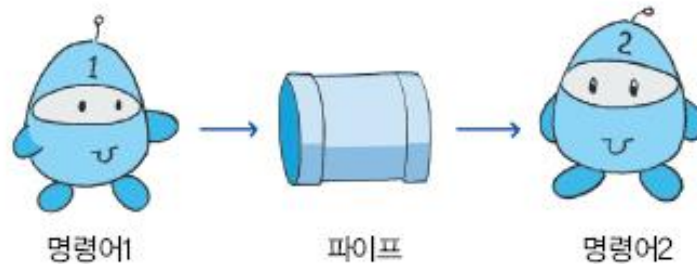
```
$ who > names.txt
```

```
$ sort < names.txt
```

- 사용법

\$ 명령어1 | 명령어2

명령어1의 표준출력이 파이프를 통해 명령어2의 표준입력이 된다.



- 예

```
$ who | sort
```

```
agape pts/5 2월 20일 13:23 [203.252.201.55]
```

```
chang pts/3 2월 20일 13:28 [221.139.179.42]
```

```
hong pts/4 2월 20일 13:35 [203.252.201.51]
```

셸(Shell)의 기본 사용법

파이프 사용 예

- 예: 로그인 된 사용자 이름 정렬

```
$ who | sort
```

```
agape pts/5 2월 20일 13:23 (203.252.201.55)
```

```
chang pts/3 2월 20일 13:28 (221.139.179.42)
```

```
hong pts/4 2월 20일 13:35 (203.252.201.51)
```

- 예: 로그인 된 사용자 수 출력

```
$ who | wc -l
```

```
3
```

- 예: 특정 디렉터리 내의 파일의 개수 출력

```
$ ls 디렉터리 | wc -w
```


셸(Shell)의 기본 사용법

입출력 재지정 관련 명령어 요약

명령어 사용법	의미
명령어 > 파일	명령어의 표준출력을 모니터 대신에 파일에 추가한다.
명령어 >> 파일	명령어의 표준출력을 모니터 대신에 파일에 추가한다.
명령어 < 파일	명령어의 표준입력을 키보드 대신에 파일에서 받는다.
명령어 << 단어 ... 단어	표준입력을 키보드 대신에 단어와 단어 사이의 입력 내용으로 받는다.
명령어 2> 파일	명령어의 표준오류를 모니터 대신에 파일에 저장한다.
명령어1 명령어2	명령어1의 표준출력이 파이프를 통해 명령어2의 표준입력이 된다.
cat 파일1 파일2 > 파일3	파일1과 파일2의 내용을 붙여서 새로운 파일3을 만들어준다.

셸(Shell)의 기본 사용법

명령어 열(command sequence)

- **명령어 열**
 - 나열된 명령어들을 순차적으로 실행한다.
- **사용법**

```
$ 명령어1; ... ; 명령어n  
나열된 명령어들을 순차적으로 실행한다.
```

- **예**
\$ date; pwd; ls
Fri Sep 2 18:08:25 KST 2016
/home/chang/linux/test
list1.txt list2.txt list3.txt

명령어 그룹(command group)

- **명령어 그룹**
 - 나열된 명령어들을 하나의 그룹으로 묶어 순차적으로 실행한다.
- **사용법**

```
$ [명령어1; ... ; 명령어n]  
나열된 명령어들을 하나의 그룹으로 묶어 순차적으로  
실행한다.
```

- **예**
\$ date; pwd; ls > out1.txt
Fri Sep 2 18:08:25 KST 2016
/home/chang/linux/test
\$ [date; pwd; ls] > out2.txt
\$ cat out2.txt
Fri Sep 2 18:08:25 KST 2016
/home/chang/linux/test
...

셸(Shell)의 기본 사용법

조건 명령어 열(conditional command sequence)

- 조건 명령어 열

- 첫 번째 명령어 실행 결과에 따라 다음 명령어 실행을 결정할 수 있다.

- 사용법

\$ 명령어1 && 명령어2

명령어1이 성공적으로 실행되면 명령어2가 실행되고, 그렇지 않으면 명령어2가 실행되지 않는다.

- 예

\$ gcc myprog.c && a.out

- 사용법

\$ 명령어1 || 명령어2

명령어1이 실패하면 명령어2가 실행되고, 그렇지 않으면 명령어2가 실행되지 않는다.

- 예

\$ gcc myprog.c || echo 컴파일 실패

셸(Shell)의 기본 사용법

여러 개 명령어 사용: 요약

명령어 사용법	의미
명령어1; ... ; 명령어n	나열된 명령어들을 순차적으로 실행한다.
[명령어1; ... ; 명령어n]	나열된 명령어들을 하나의 그룹으로 묶어 순차적으로 실행한다.
명령어1 && 명령어2	명령어1이 성공적으로 실행되면 명령어2가 실행되고, 그렇지 않으면 명령어2가 실행되지 않는다.
명령어1 명령어2	명령어1이 실패하면 명령어2가 실행되고, 그렇지 않으면 명령어2가 실행되지 않는다.

셸(Shell)의 기본 사용법

파일 이름 대치

- 대표문자를 이용한 파일 이름 대치
 - 대표문자를 이용하여 한 번에 여러 파일들을 나타냄
 - 명령어 실행 전에 대표문자가 나타내는 파일 이름들로 먼저 대치하고 실행

대표문자	의미
*	빈 스트링을 포함하여 임의의 스트링을 나타냄
?	임의의 한 문자를 나타냄
[..]	대괄호 사이의 문자 중 하나를 나타내며 부분범위 사용 가능함

```
$ gcc *.c
```

```
$ gcc a.c b.c test.c
```

```
$ ls *.txt
```

```
$ ls [ac]*
```

셸(Shell)의 기본 사용법

명령어 대치(command substitution)

- 명령어를 실행할 때 다른 명령어의 실행 결과를 이용
 - **명령어** 부분은 그 명령어의 실행 결과로 대치된 후에 실행
- 예

```
$ echo 현재 시간은 date
```

```
$ echo 현재 디렉터리 내의 파일의 개수 : ls | wc -w
```

```
현재 디렉터리 내의 파일의 개수 : 32
```

셸(Shell)의 기본 사용법

따옴표 사용

- 따옴표를 이용하여 대치 기능을 제한

```
$ echo 3 * 4 = 12
```

```
3 cat.csh count.csh grade.csh invite.csh menu.csh test.sh = 12
```

```
$ echo "3 * 4 = 12"
```

```
3 * 4 = 12
```

```
$ echo '3 * 4 = 12'
```

```
3 * 4 = 12
```

```
$ name=나가수
```

```
$ echo '내 이름은 $name 현재 시간은 date '
```

```
내 이름은 $name 현재 시간은 date
```

```
$ echo "내 이름은 $name 현재 시간은 date "
```

```
내 이름은 나가수 현재 시간은 2016. 11. 11. (금) 10:27:48 KST
```

- 정리

- 작은따옴표(')는 파일이름 대치, 변수 대치, 명령어 대치를 모두 제한한다.
- 큰따옴표(")는 파일이름 대치만 제한한다.
- 따옴표가 중첩되면 밖에 따옴표가 효력을 갖는다.

셸(Shell)의 기본 사용법

cut 명령어

- 기능

파일에서 필드를 뽑아낸다. 필드는 구분자로 구분할 수 있다.

- 사용법

cut [option] [file]

- 옵션

-c 문자위치 : 잘라낼 곳의 글자 위치를 지정한다. 콤마나 하이픈을 사용하여 범위를 정할 수도 있으며, 이런 표현들을 혼합하여 사용할 수도 있다.

-f 필드 : 잘라낼 필드를 정한다.

-d 구분자 : 필드를 구분하는 문자를 지정한다. 디폴트는 탭 문자다.

-s : 필드 구분자를 포함할 수 없다면 그 행은 하지 않는다.

```
vagrant@localhost:~$ cut -b 2 cutdata.txt
1
2
3
4
5
vagrant@localhost:~$
```

```
root@localhost
vagrant@localhost:~$ cut -b 2-4 cutdata.txt
1:o
2:a
3:a
4:g
5:o
vagrant@localhost:~$
```

```
root@localhost
vagrant@localhost:~$ cut -b 2-4,7-10 cutdata.txt
1:onge:
2:ale:2
3:ale:1
4:gpe:4
5:onge:
vagrant@localhost:~$
```

```
root@localhost
vagrant@localhost:~$ cut -b 2,4- cutdata.txt
1orange:250:Tokyo
2apple:230:Nagoya
3apple:130:Nagoya
4grape:450:Tokyo
5orange:150:Osaka
vagrant@localhost:~$
```

```
root@localhost
vagrant@localhost:~$ cut -f 2 -d ":" cutdata.txt
orange
apple
apple
grape
orange
vagrant@localhost:~$
```


셸(Shell)의 기본 사용법

paste 명령어

ex) exam 1, exam2 2개의 파일이 있을 경우

```
[root@chefclient02 ~]$cat exam 1
```

red

blue

white

```
[root@chefclient02 ~]$cat exam2
```

yellow

green

gray

black

```
[root@chefclient02 ~]$paste exam 1 exam2
```

red yellow

blue green

white gray

black

```
[root@chefclient02 ~]$paste -d : exam 1 exam2
```

red:yellow

blue:green

white:gray

:black

```
[root@chefclient02 ~]$paste -d : exam 1 exam2
```

red:yellow

blue:green

white:gray

:black

```
[root@chefclient02 ~]$paste -s exam 1 exam2
```

red blue white

yellow greengray black

```
[root@chefclient02 ~]$paste -s -d '|' exam 1 exam2
```

red|blue|white

yellow|green|gray|black

셸(Shell)의 기본 사용법

paste -d - f1.txt f2.txt f3.txt f4.txt

```
vagrant@localhost: ~  
vagrant@localhost: ~$ paste -d - f1.txt f2.txt f3.txt f4.txt  
1- Apple- 100- 235  
2- Orange- 150-  
3- Grape- 300- 54  
4- - -  
vagrant@localhost: ~$
```

```
vagrant@localhost: ~  
vagrant@localhost: ~$ paste -s f1.txt f2.txt f3.txt f4.txt  
1      2      3      4  
Apple  Orange Grape  
100    150    300  
235    54  
vagrant@localhost: ~$
```

```
vagrant@localhost: ~  
vagrant@localhost: ~$ cat f1.txt f2.txt f3.txt f4.txt  
1  
2  
3  
4  
Apple  
Orange  
Grape  
100  
150  
300  
235  
  
54  
vagrant@localhost: ~$
```

핵심 개념

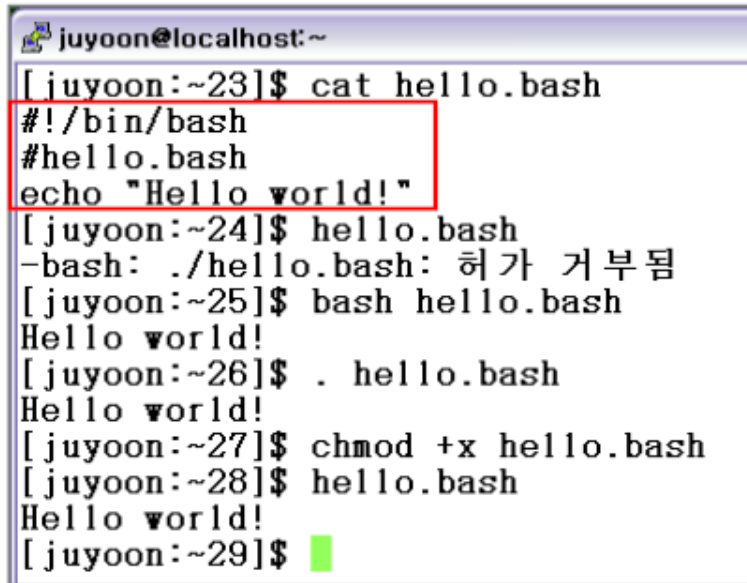
- 셸은 사용자와 운영체제 사이에 창구 역할을 하는 소프트웨어로 사용자로부터 명령어를 입력받아 이를 처리하는 명령어 처리기 역할을 한다.
- 출력 재지정은 표준출력 내용을 파일에 저장하고 입력 재지정은 표준입력을 파일에서 받는다.
- 파이프를 이용하면 한 명령어의 표준출력을 다른 명령어의 표준입력으로 바로 받을 수 있다.

□ 셸 프로그램

- 복잡한 명령어나 반복적인 명령어를 처리할 때는 셸 명령어들을 나열한 **스크립트**를 작성

□ Simple Example

- 첫 행은 항상 **#!**로 시작
 - 시스템에 해당 스크립트가 직접 실행 가능하다는 것을 알림
 - /bin/bash: Bash 셸로 명령어를 해석해야 함
- 나머지 행의 **#**는 주석(comments)
 - 주석은 실행에서 제외되며 스크립트를 설명하기 위해 사용

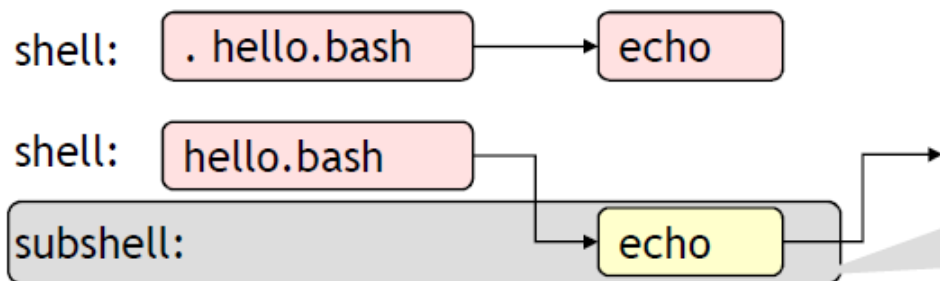


```
juyoon@localhost:~  
[juyoon:~23]$ cat hello.bash  
#!/bin/bash  
#hello.bash  
echo "Hello world!"  
[juyoon:~24]$ hello.bash  
-bash: ./hello.bash: 허가 거부됨  
[juyoon:~25]$ bash hello.bash  
Hello world!  
[juyoon:~26]$ . hello.bash  
Hello world!  
[juyoon:~27]$ chmod +x hello.bash  
[juyoon:~28]$ hello.bash  
Hello world!  
[juyoon:~29]$
```

□ 실행 방법

- . 명령으로 실행
 - \$. hello.bash
- 실행 권한을 부여한 후 직접 호출
 - \$ chmod +x hello.bash
 - \$ hello.bash

□ 직접 호출 시에는 하위 셸 (subshell)을 생성하여 실행



전역환경변수를
부모로부터 복사
하고 지역변수는
새로 초기화해서
사용

□ 셸 스크립트 기본 요소

- 변수 (variable)
- 함수 (function)
- 제어문 (control statement)
 - 조건문 (conditional branch)
 - 반복문 (iteration)
- 명령행 옵션 (command line option)
- 인터럽트 처리 (interrupt handling)

□ 표준 출력

`echo` 문자열

□ 표준 입력

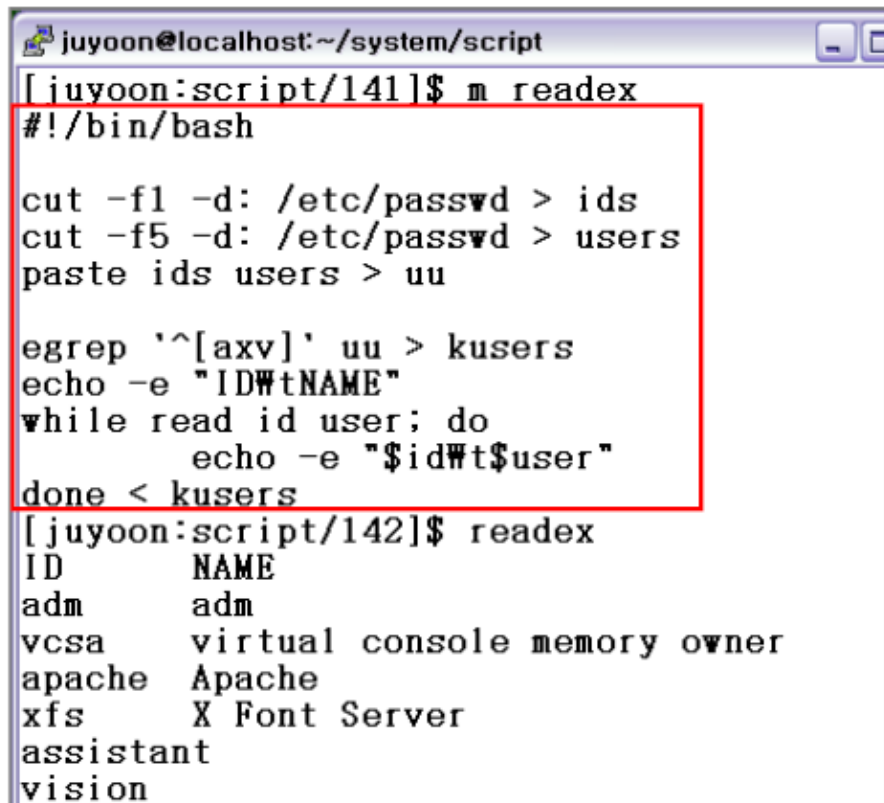
`read varname ...`

- 변수의 개수보다 입력 단어가 많으면 마지막 변수에 모두 할당
- 변수 이름 없이 사용하면 **REPLY** 변수에 입력 할당
- 기본적으로 **행 단위** 처리

```
juyoon@localhost:~/system/script
[juyoon:script/20]$ read a b
baboo computer merong!
[juyoon:script/21]$ echo $a
baboo
[juyoon:script/22]$ echo $b
computer merong!
[juyoon:script/23]$ read
baboo computer merong!
[juyoon:script/24]$ echo $REPLY
baboo computer merong!
[juyoon:script/25]$
```

□ 파일에서 입력

- redirection을 활용한다.



```
juyoon@localhost:~/system/script
[juyoon:script/141]$ m readex
#!/bin/bash

cut -f1 -d: /etc/passwd > ids
cut -f5 -d: /etc/passwd > users
paste ids users > uu

egrep '^[axv]' uu > kusers
echo -e "ID\tNAME"
while read id user; do
    echo -e "$id\t$user"
done < kusers
[juyoon:script/142]$ readex
ID      NAME
adm      adm
vcsa     virtual console memory owner
apache   Apache
xfs      X Font Server
assistant
vision
```

The image shows a terminal window titled 'juyoon@localhost:~/system/script'. The user runs a command 'm readex' which executes a shell script. The script's content is displayed within a red rectangular box. The script uses input redirection (<) to read from a file 'kusers' and output redirection (>) to write to files 'ids', 'users', 'uu', and 'kusers'. The script processes user data from /etc/passwd and formats it into a table. The final output of the script is shown below the red box.

□ 셸 변수: bash 제공 내장 변수 + 사용자 지정 변수

□ 변수명

- 예약어(reserved word)가 아니어야 한다.

```
if      then  else  elif  fi      case  esac  
for     while until do    done  function  
in      select          !    {      }      time
```

- 영문자로 시작하며 '_'와 영숫자로만 이루어진다.

□ 변수값 설정

```
varname=value
```

- value는 문자열, 공백이 포함되면 '나 "로 쓴다.

□ 변수값 참조

- \$varname 또는 \${varname}

□ 변수 사용 범위

- 기본적으로 **지역 변수**, 즉 현재 실행 중인 셸 또는 스크립트에서만 효력을 가진다.
- **export** 명령으로 **환경 변수**가 된다. 즉, 하위 셸 또는 로그아웃 후 재실행하는 경우에도 사용할 수 있다.

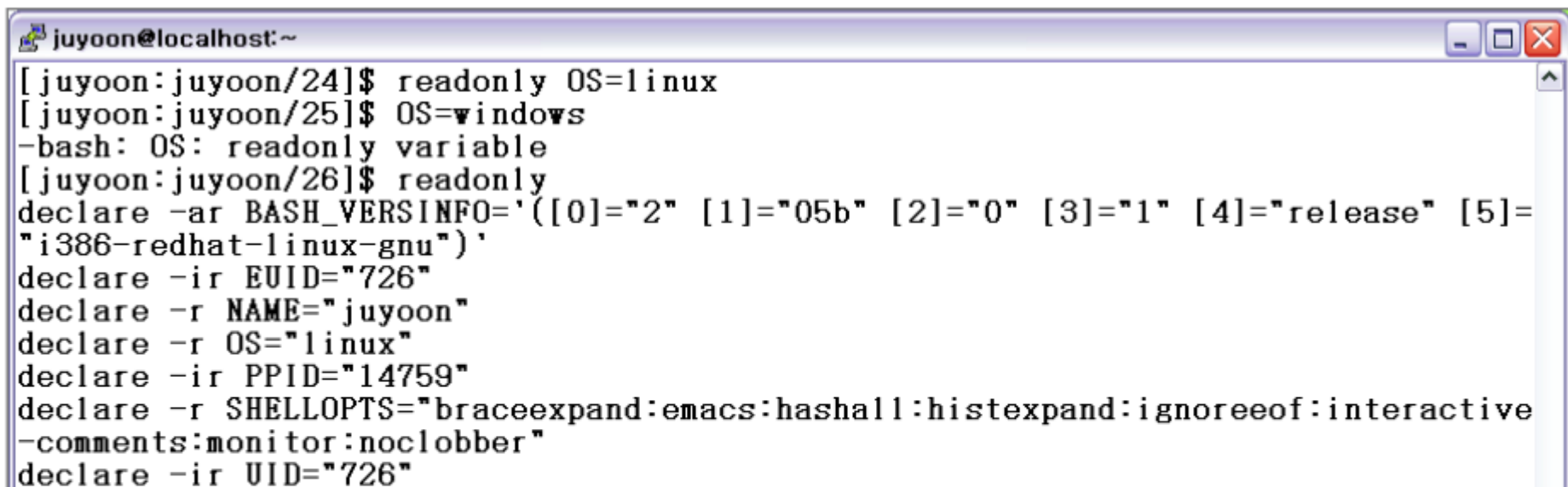
□ 읽기 전용 변수: readonly

- 변수의 값을 변경하지 못하게 만드는 셸 내장 명령
- 인자가 없으면 현재 설정된 읽기 전용 변수 확인

```
readonly [-af] [name[=value] ...]
```

- 새로운 셸 또는 하위 셸이 실행될 때 readonly 속성은 상속되지 않는다.

셸 변수 예



```
juyoon@localhost:~  
[juyoon:juyoon/24]$ readonly OS=linux  
[juyoon:juyoon/25]$ OS=windows  
-bash: OS: readonly variable  
[juyoon:juyoon/26]$ readonly  
declare -ar BASH_VERSINFO='([0]="2" [1]="05b" [2]="0" [3]="1" [4]="release" [5]=  
"i386-redhat-linux-gnu")'  
declare -ir EUID="726"  
declare -r NAME="juyoon"  
declare -r OS="linux"  
declare -ir PPID="14759"  
declare -r SHELLOPTS="braceexpand:emacs:hashall:histexpand:ignoreeof:interactive  
-comments:monitor:noclobber"  
declare -ir UID="726"
```

셸 명령어 사용법을 잘 모르겠다? 게다가 man 해도 정보가 없다?
→ 셸 내장 명령어는 'help'로 찾아 본다!

□ 위치 변수

셸 변수	의미
\$\$	명령을 실행하는 프로세스 번호
\$#	명령어 줄에서의 변수의 수
\$?	마지막으로 실행된 명령어의 exit 상태. 일반적으로 명령어 실행 후에는 0의 값을 반환한다.
\$1, ..., \$9	명령어 줄 상에서의 위치에 따른 각 인수
\$0	스크립트 이름
\$*	명령어 줄 상에서의 모든 인수를 포함하는 하나의 문자열
\$@	명령어 줄 상에서의 모든 인수를 나타내는 각 문자열, 즉 "\$1" "\$2" "\$3" ... (공백으로 구분)

□ 위치 변수 예제

```
juyoon@localhost:~/system/script
[juyoon:script/53]$ m ex1
#!/bin/bash

echo "PID: $$"
echo "parameters:$@"
echo "$0: $1 $2 $3 $4"
echo "$# arguments"
echo $*
[juyoon:script/54]$ ex1
PID: 21054
parameters:
./ex1:
0 arguments

[juyoon:script/55]$ ex1 a b c
PID: 21055
parameters:a b c
./ex1: a b c
3 arguments
a b c
[juyoon:script/56]$
```

□ 문자열 연산

연산자	의미	목적
<code>\${var:-word}</code>	var이 존재하고 값이 널이 아니면 값을 반환, 아니면 word 반환	정의하지 않은 변수 사용 시 오류 대신 기본값 사용
<code>\${var:=word}</code>	위와 같으나 위치변수와 특수변수 제외	
<code>\${var:?mesg}</code>	var이 존재하고 값이 널이 아니면 값을 반환, 아니면 mesg 출력하고 현재 명령 무시	정의하지 않은 변수 사용 시 오류 메시지 출력
<code>\${var:+word}</code>	var이 존재하고 값이 널이 아니면 word 반환, 아니면 널 반환	변수의 존재 유무 검사
<code>\${var:offset}</code> <code>\${var:offset:length}</code>	var의 값(문자열)에서 offset부터 length까지 부분문자열 반환. offset은 0부터 센다.	문자열의 일부분 사용

□ 문자열 연산 예제

```
juyoon@linda: ~/test/system/shell_scripts
juyoon@linda:~/test/system/shell_scripts$ m ex2
#!/bin/bash

# 명령행에 입력된 파일명을 이용해 새 파일 이름을 만든다.
# 이름이 입력되지 않았으면 오류 메시지를 출력하고 종료한다.

names=${1:?''명단 파일 이름이 입력되지 않았습니다.
Usage: ex2 namefile scorefile''}

# 점수 파일 이름이 입력되지 않았으면 score를 사용한다.
score=${2:-score}
newfile=$names.$score

# 학생 명단과 점수 파일을 붙여 새 파일을 만든다.

paste $names $score > $newfile

# 새로운 파일을 점수 순서대로 정렬한다.

sortfile="$newfile.sort"
sort -nr -k5 $newfile > $sortfile

juyoon@linda:~/test/system/shell_scripts$
```

□ 패턴 비교

연산자	의미
<code>\${var#pattern}</code>	pattern과 var 값을 앞에서 비교하여 일치하는 가장 짧은 부분을 삭제하고 나머지 반환
<code>\${var##pattern}</code>	pattern과 var 값을 앞에서 비교하여 일치하는 가장 긴 부분을 삭제하고 나머지 반환
<code>\${var%pattern}</code>	pattern과 var 값을 끝부터 비교하여 일치하는 가장 짧은 부분을 삭제하고 나머지 반환
<code>\${var%%pattern}</code>	pattern과 var 값을 끝부터 비교하여 일치하는 가장 긴 부분을 삭제하고 나머지 반환
<code>\${var/pattern/string}</code>	pattern과 var 값을 비교하여 일치하는 가장 긴 부분을 string으로 대체. 처음 하나만 바꾼다.
<code>\${var//pattern/string}</code>	pattern과 var 값을 비교하여 일치하는 가장 긴 부분을 string으로 대체. 모든 일치하는 부분을 바꾼다.

□ 명령 대체 (command substitution)

- 명령의 표준 출력을 변수 값처럼 사용

`$ (command)`

`-`

```
juyoon@localhost:~/system/script
[juyoon:script/109]$ m subs
#!/bin/bash
# pwd 명령 수행 결과를 mypwd 변수에 저장
mypwd=$(pwd)
echo $mypwd
# newline 출력
echo -e -n "WWW"
# pattern 연산 결과 출력
echo ${mypwd##*/}
echo ${mypwd#*/}
echo ${mypwd%*/}
echo ${mypwd%%*/}
echo ${mypwd/home/myhome}
# 또 다른 명령 수행 방법과 결과
echo -e -n "Wn"
echo `ls $HOME`
[juyoon:script/110]$ subs
/home/juyoon/system/script

script
juyoon/system/script
/home/juyoon/system/script
/home/juyoon/system/script
/myhome/juyoon/system/script

5 cprogramming hello.bash man-solution sys
tem system.tar.gz
```

□ 함수 - script within a script

- 함수 이름을 명령어처럼 사용
 - 셸 메모리에 미리 저장 - 속도가 빠르다.
 - 내장 명령이나 스크립트보다 우선 순위가 높다.

- 함수 정의

```
function func_name {  
    shell commands  
}
```

```
func_name() {  
    shell commands  
}
```

- 중괄호 '{' 다음엔 반드시 공백
- 함수 정의의 삭제

```
unset -f func_name
```

□ 변수의 참조 범위

■ 지역 변수

- 함수 내에서 정의된 변수
- 함수 내에서만 의미가 있고 함수를 벗어나면 의미가 없다.

■ 전역 변수

- 전체 스크립트에서 의미가 있는 변수

```
#!/bin/bash
```

```
function afunc {  
    echo in function: $0 $1 $2  
    var1="in function"  
    echo var1: $var1  
}
```

지역 변수

전역 변수

```
var1="outside function"  
echo var1: $var1  
echo $0: $1 $2  
afunc funcarg1 funcarg2  
echo var1: $var1  
echo $0: $1 $2
```

위치 변수만
지역적!

□ 변수의 참조 범위

- 일반 변수를 지역 변수로 만들고 싶으면? → **local**

```
#!/bin/bash

function afunc {
    local var1
    echo in function: $0 $1 $2
    var1="in function"
    echo var1: $var1
}

var1="outside function"
echo var1: $var1
echo $0: $1 $2
afunc funcarg1 funcarg2
echo var1: $var1
echo $0: $1 $2
```

□ 명령행에서 직접 함수를 사용하고 싶으면?

- .bashrc에 넣거나 명령행에서 함수 정의

□ 프로그램 실행 우선 순위

- 같은 이름의 명령, 함수 등이 있을 때
 - 1) alias
 - 2) function, if, for 등의 키워드
 - 3) 함수
 - 4) 내장 명령 (cd, type, pwd, ...)
 - 5) 스크립트와 프로그램 (PATH 순서대로)

□ 순서 바꾸기

- alias → unalias로 삭제
- command: 내장 명령과 일반 명령만 사용
- builtin: 내장 명령만 사용
- enable -n: 내장 명령 금지

□ 이름이 어떻게 정의되어 있는가? - type 명령

```
type [-all | -path | -type] name
```

```
juyoon@localhost:~/system/script
[juyoon:script/14]$ function ls {
> /bin/ls -la "$@"
> }
[juyoon:script/15]$ type -all ls
ls is aliased to `ls -sFC'
ls is a function
ls ()
{
    /bin/ls -la "$@"
}
ls is /bin/ls
[juyoon:script/16]$ type -path ls
[juyoon:script/17]$ type -type ls
alias
[juyoon:script/18]$ type -all -path ls
/bin/ls
[juyoon:script/19]$ type -all -type ls
alias
function
file
```

□ pushd, popd

- 셸 내장 명령
- 작업 디렉터리를 스택에 저장해 두었다가 되돌아갈 때 사용
- **dirs**: 디렉터리 스택을 보여 준다.
- **pushd *dir***
 - 스택에 현재 디렉터를 push
→ *dir*로 이동하고 *dir*을 스택에 push
 - 연속 실행 시에는 *dir*만 push
- **popd**: 스택 top에 있는 디렉터를 pop해서 삭제하고 다음 디렉터리가 top이 되며 그 곳으로 이동
- 관련 내장 변수: DIRSTACK

□ 실행 예시

명령	스택의 내용	결과 디렉터리
pushd system	~/system ~	~/system
pushd /etc	/etc ~/system ~	/etc
popd	~/system ~	~/system
popd	~	~
popd	<empty>	(error)

□ 옵션

- +N, -N: N개 만큼 rotate

□ pushd, popd를 직접 구현해 보자!

- 두 함수에 공통인 스택을 사용해야 하므로 환경변수를 설정한다.

```
$ DIR_STACK=""  
$ export DIR_STACK
```

DIR_STACK은 디렉터리 이름들을 저장할 문자열

```
pushd() {  
    dirname=$1  
    DIR_STACK="$dirname ${DIR_STACK:-$PWD' '}"  
    cd ${dirname:? "missing directory name"}  
    echo "$DIR_STACK"  
}
```

기존 DIR_STACK에 현재 입력 디렉터리를 덧붙인다.

```
popd() {  
    DIR_STACK=${DIR_STACK%* }  
    cd ${DIR_STACK%% *}  
    echo "$PWD"  
}
```

DIR_STACK 맨 앞 단어(공백 구분)를 잘라낸다.

DIR_STACK 맨 앞 단어로 이동한다.

- 어떤 문제가 있는가?

□ Bash에서 사용되는 제어구조

- if/then/else
- for
- while
- until
- case
- select
- break/continue/exit

□ if/then/else

- if 다음의 조건식이 참(true)이면 then 뒤의 명령어들을 실행하라는 의미
- 조건이 비교될 때 **결과값이 0이면 참으로 인식**
 - 명령어가 오류 없이 실행되면 0을 반환 (exit code)

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

위치를 지켜!

<사용 예>

```
pushd() {
    dirname=$1
    if cd ${dirname:?*"missing directory name"}
    then
        DIR_STACK="$dirname ${DIR_STACK:-$PWD' '}"
        echo $DIR_STACK
    else
        echo still in $PWD
    fi
}
```

□ 조건 검사

- [conditions] 또는 test 문 사용

□ 문자열 비교

연산자	참인 경우
str1 = str2	str1과 str2가 같다.
str1 != str2	str1과 str2가 같지 않다.
str1 < str2	str1이 str2보다 작다.
str1 > str2	str1이 str2보다 크다.
-n str1	str1이 널이 아니다.
-z str1	str1이 널이다. (길이가 0)

■ 사용 예

```
popd() {  
    if [ -n "$DIR_STACK" ]; then  
        DIR_STACK=${DIR_STACK#* }  
        cd "${DIR_STACK%% *}"  
        echo "$PWD"  
    else  
        echo "stack empty, still in $PWD"  
    fi  
}
```

<https://linuxconfig.org/bash-scripting-tutorial#h1-hello-world-bash-shell-script>

1. Hello World Bash Shell Script
2. Simple Backup bash shell script
3. Global vs. Local variables
4. Declare simple bash array
5. Read file into bash array
6. Simple Bash if/else statement
7. Nested if/else
8. Arithmetic Comparisons
9. String Comparisons
10. Bash for loop
11. Bash while loop
12. Bash until loop
13. Control bash loop with
14. Escaping Meta characters
15. Single quotes

16. Double Quotes
17. Bash quoting with ANSI-C style
18. Bash Addition Calculator Example
19. Bash Arithmetics
20. Round floating point number
21. Bash floating point calculations
22. STDOUT from bash script to
STDERR
23. STDERR from bash script to
STDOUT
24. stdout to screen
25. stdout to file
26. stderr to file
27. stdout to stderr
28. stderr to stdout
29. stderr and stdout to file

Hello World Bash Shell Script

Bash
위치
확인

\$ which bash

<https://linuxconfig.org/bash-scripting-tutorial#h1-hello-world-bash-shell-script>

소스
작성

```
#!/bin/bash  
# declare STRING variable  
STRING="Hello World"  
#print variable on a screen  
echo $STRING
```

실행
파일
화

\$ chmod +x hello_world.sh

실행

./hello_world.sh

Simple Backup bash shell script

소스
명

Backup.sh

소스
작성

```
#!/bin/bash  
tar -czf myhome_directory.tar.gz /home/linuxconfig
```

실행
파일
화

\$ chmod +x **Backup.sh**

실행

./Backup.sh

Your backup script and variables:

소스
명

Backup2.sh

소스
작성

```
#!/bin/bash  
OF=myhome_directory_$(date +%Y%m%d).tar.gz  
tar -czf $OF /home/linuxconfig
```

실행
파일
화

\$ chmod +x **Backup2.sh**

실행

./Backup2.sh

Global vs. Local variables

소스
작성

```
#!/bin/bash
#Define bash global variable
#This variable is global and can be used anywhere in this bash script
VAR="global variable"
function bash {
#Define bash local variable
#This variable is local to bash function only
local VAR="local variable"
echo $VAR
}
echo $VAR
bash
# Note the bash global variable did not change
# "local" is bash reserved word
echo $VAR
```

Passing arguments to the bash script

소스 작성

```
#!/bin/bash
# use predefined variables to access passed arguments
#echo arguments to the shell
echo $1 $2 $3 ' -> echo $1 $2 $3'

# We can also store arguments from bash command line in special array
args=("$@")
#echo arguments to the shell
echo ${args[0]} ${args[1]} ${args[2]} ' -> args=("$@"); echo ${args[0]} ${args[1]} ${args[2]}'

#use $@ to print out all arguments at once
echo $@ ' -> echo $@'

# use $# variable to print out
# number of arguments passed to the bash script
echo Number of arguments passed: $# ' -> echo Number of arguments passed: $#'
```

/arguments.sh Bash Scripting Tutorial

```
linuxconfig.org$ ./arguments.sh Bash Scripting Tutorial
Bash Scripting Tutorial -> echo $1 $2 $3
Bash Scripting Tutorial -> args=("$@"); echo ${args[0]} ${args[1]} ${args[2]}
Bash Scripting Tutorial -> echo $@
Number of arguments passed: 3 -> echo Number of arguments passed: $#
linuxconfig.org$
```

Reading User Input

소스
작성

```
#!/bin/bash
```

```
echo -e "Hi, please type the word: \c "
```

```
read word
```

```
echo "The word you entered is: $word"
```

```
echo -e "Can you please enter two words? "
```

```
read word1 word2
```

```
echo "Here is your input: \"$word1\" \"$word2\""
```

```
echo -e "How do you feel about bash scripting? "
```

```
# read command now stores a reply into the default build-in variable $REPLY
```

```
read
```

```
echo "You said $REPLY, I'm glad to hear that! "
```

```
echo -e "What are your favorite colours ? "
```

```
# -a makes read command to read into an array
```

```
read -a colours
```

```
linuxconfig.org$ ./read.sh
```

```
Hi, please type the word: linuxconfig.org
```

```
The word you entered is: linuxconfig.org
```

```
Can you please enter two words?
```

```
Debian Linux
```

```
Here is your input: "Debian" "Linux"
```

```
How do you feel about bash scripting?
```

```
good
```

```
You said good, I'm glad to hear that!
```

```
What are your favorite colours ?
```

```
blue green black
```

```
My favorite colours are also blue, green and black;-)
```

```
linuxconfig.org$
```

Reading User Input

소스
작성

```
#!/bin/bash
# bash trap command
trap bashtrap INT
# bash clear screen command
clear;
# bash trap function is executed when CTRL-C is pressed:
# bash prints message => Executing bash trap subroutine !
bashtrap()
{
echo "CTRL+C Detected !...executing bash trap !"
}
# for loop from 1/10 to 10/10
for a in `seq 1 10`; do
echo "$a/10 to Exit."
sleep 1;
done
echo "Exit Bash Trap Example!!!"
```

```
1/10 to Exit.
2/10 to Exit.
3/10 to Exit.
4/10 to Exit.
5/10 to Exit.
6/10 to Exit.
CTRL+C Detected !...executing bash trap !
7/10 to Exit.
8/10 to Exit.
9/10 to Exit.
CTRL+C Detected !...executing bash trap !
10/10 to Exit.
Exit Bash Trap Example!!!
linuxconfig.org ~$
```

Arrays Declare simple bash array

소스
작성

```
#!/bin/bash
#Declare array with 4 elements
ARRAY=( 'Debian Linux' 'Redhat Linux' Ubuntu Linux )
# get number of elements in the array
ELEMENTS=${#ARRAY[@]}

# echo each element in array
# for loop
for (( i=0;i<$ELEMENTS;i++)); do
echo ${ARRAY[$i]}
done
```

```
linuxconfig.org$ ./arrays
Debian Linux
Redhat Linux
Ubuntu
Linux
linuxconfig.org$ █
```