8. 셸 프로그래밍

8.1 셸 프로그램에 유용한 명령어

- expr – 산술 연산
  - Bourne shell은 산술 연산을 지원하지 않으며 산술 연산을 위해 expr 명령어를 사용함

```
$ expr 1  +  2
3
```

  - 모든 연산자와 피연산자는 공백으로 구분된 인수로 제공되어야 함
  - 메타문자와 같은 연산자는 \를 함께 사용하거나 인용부호 사용

```
$ expr 1+2     … 틀렸음
$ expr 3 \* 4
```


expr – 산술연산/문자열 연산 수행

- 산술 연산 – (예) 교과서 참조

연산자	의미
* / %	곱셈, 나눗셈, 나머지
+ -	덧셈, 뺄셈
< <= >= > = == !=	크기 비교, 참이면 1, 거짓이면 0 출력
&	(AND) 첫째 인수가 0(널)이 아니면 첫째 인수를, 0이면 0을 출력
|	(OR) 첫째 인수가 0(널)이 아니면 첫째 인수를, 0이면 둘째 인수를 출력

- 문자열 연산 – (예) 교과서 참조

연산자	의미
`match str RE` `str : RE`	문자열 str의 앞부분이 정규표현식 RE와 일치하면 일치한 문자열 길이를, 그렇지 않으면 0을 출력 (두 형식은 같은 의미)
`substr str start len`	문자열 str의 start 위치부터 len 개 만큼의 부분문자열 출력 위치는 1부터 시작함
`index str charList`	문자열 str에서 charList에 포함된 문자가 처음으로 나타난 위치를 출력
`length str`	문자열 str의 길이를 출력

test – 조건 검사

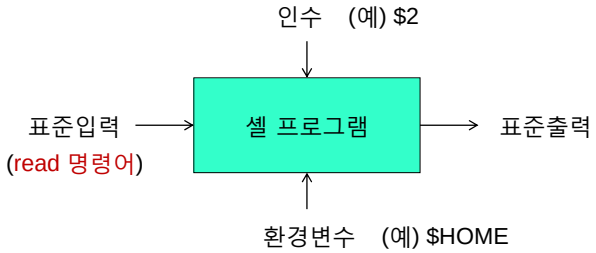
- test
  - 파일의 유형 검사, 문자열/정수 비교 → 결과를 종료코드로 반환
    - 참 : 종료코드 0
    - 거짓 : 종료코드 1
  - 화면에 결과를 출력하지 않음
  - 셸 프로그램의 제어문에서 조건 검사용 명령어로 주로 사용
  - 사용 형식:

파일 유형 검사	문자열/정수 비교
test –option filename	test v1 –op v2
[-option filename]	[v1 –op v2]

- 파일 속성 검사 옵션
  - -d (directory) -f (보통 file) -e (exist) -s (exist, 크기 0이 아님)
  - -r (read) –w (write) –x (execute) -L (symbolic link) 등
- 문자열/정수/파일수정날짜 비교
  - 정수 비교: -eq –ne –gt –ge –lt –le
  - 문자열 비교: = != -z -n
  - 파일수정날짜 비교: -nt -ot
- 검사/비교 결과들의 결합
  - ! -a -o \(.. \)

셸 프로그램 입력

- 셸 프로그램의 입출력 경로



수정된 환경에서 명령어 실행

- 환경변수
  - 셸 프로그램이나 명령어에서 접근 가능
  - 프로그램에 전달할 목적으로 셸에서 환경변수 값을 변경한 경우 프로그램이 종료한 이후에도 변경 값이 그대로 유지됨
- env
  - 수정된 환경변수 값을 사용하여 명령어를 실행하도록 함
  - 환경변수 값을 지정한 후 명령어 실행

    $ env *var1=value1 var2=value2 ... command*
  - env에서의 환경변수 값의 변경은 command에만 전달되며 현재 셸의 환경변수 값에는 영향이 없음
  - 인수 없이 사용하면 현재 설정된 환경변수 이름과 값을 모두 출력

    $ env

read – 표준 입력

- read
  - 한 줄을 입력, 인수로 지정된 변수들에 단어 단위로 저장
  - 인수 개수 < 단어 개수 → 나머지 단어들은 마지막 인수에 저장

    read x y
- read –p 입력 안내문을 함께 출력

```
echo -n "입력안내문"
read x y
```

```
read –p "입력안내문"  x  y
```

특별한 내장 변수

변수	의미
$$	현재 셸의 프로세스 번호(pid)
$!	가장 최근에 백그라운드로 실행한 프로세스의 번호(pid)
$?	직전 명령어의 종료코드
$n	n번째 인수, n이 10이상이면 { }를 함께 사용함
$0	명령어 이름 (상대경로 또는 절대경로로 확장됨)
$*	모든 인수
$@	모든 인수를 나타내는 데 "$@"와 같이 사용할 때에 '$1' '$2' ... 와 같이 인수들을 개별적으로 구분이 된다.

- $*와 $@의 차이점
  - 모든 인수를 나타냄
  - "$*"와 "$@"가 다름 → 뒤의 for 반복문에서 차이점을 예시

shift, readonly, unset

- shift – 인수 이동

```
shift        ... 인수를 왼쪽으로 이동 ($2가 $1로)
shift  2     ... 인수를 2번 왼쪽으로 이동 ($3이 $1로)
```

  - shift할 인수가 없으면 종료코드 1
- readonly – 읽기 전용 변수

```
readonly var     ... var을 읽기 전용변수로 지정
var=123          ... 수정되지 않음
readonly         ... 모든 읽기전용변수 출력
```

  - 읽기전용 변수 속성은 서브셸로 상속되지 않음
- unset – 설정된(정의된) 변수 제거

  $ unset var
  - 정의되지 않은(제거된) 변수 값은 null 문자열로 출력됨

eval

- eval
  - 모든 인수를 하나의 명령어를 결합하여 셸에서 실행함

```
$ foo=10   x=foo
$ echo ${$x}          ... x에 저장된 이름의 변수 접근을 시도 ($foo)
-bash: ${$x}: bad substitution ... 이중 변수 대치가 되지 않음
$ eval echo \${$x}    ... 실행할 명령어를 갖는 인수 생성, eval로 실행
10                    ... echo ${foo} 를 실행
```


wait

- wait – 자식 프로세스 종료 대기

```
wait pid      ... 지정된 프로세스 종료 대기
wait          ... 모든 자식 프로세스 종료 대기
```

```
#!/bin/sh
date +%s              ... 1970년1월1일 이후 경과된 초
sleep 20 &
sleep 10 &
wait                  ... 모든 자식프로세스 종료 대기
date +%s
```

```
#!/bin/sh
date +%s              ... 1970년1월1일 이후 경과된 초
sleep 20 &
sleep 10 &
pid=$!                ... 최근의 백그라운드 프로세스 번호
wait $pid             ... pid번호 프로세스 종료 대기
date +%s
```

파일입출력 방향전환

- 셸은 일반적인 파일입출력의 방향전환 기능 제공 (참고)

변수	의미
2>*filename*	표준에러 출력을 filename으로 방향전환
2>>*filename*	표준에러 출력을 filename으로 방향전환, 추가출력
2>&1	표준에러 출력으로 표준 출력 장치를 같이 사용
1>&2	표준 출력으로 표준에러 출력 장치를 같이 사용
&>*filename*	표준출력과 표준에러출력을 filename으로 동시에 전환
&>>*filename*	표준출력과 표준에러출력을 filename으로 동시에 전환, 추가 출력
n<*filename*	filename을 입력용으로 열고 n번 파일기술자를 할당 (n이 없으면 0)
n>*filenamee*	filename을 출력용으로 열고 n번 파일기술자를 할당 (n이 없으면 1)
n>>*filename*	filename을 추가 출력용으로 열고 n번 파일기술자를 할당
n<&m	n번 파일기술자에 m번 파일기술자를 복제. m번과 같은 입력 장치 사용
n>&m	n번 파일기술자에 m번 파일기술자를 복제. m번과 같은 출력 장치 사용
n<&-	n번 파일기술자 입력 장치를 닫음 (n이 없으면 0)
n>&-	n번 파일기술자 출력 장치를 닫음 (n이 없으면 1)

8.2 제어 구문

- if .. then .. fi 구문

```
if 조건명령어
then
  명령어리스트
fi
```

if [$# -eq 0]; then 로 표기 가능

- 예

```
#!/bin/sh
if [ $# -eq 0 ]                          ... 인수개수($#)가 0이면
then
        echo "usage: $0 argument" 1>&2  ... 사용법을 표준에러출력으로 출력
        exit 1                            ... 종료코드 1 (실패)
fi
echo Hi! $*                               ... 모든 인수 출력
```

```
$ ./if1.sh                                ... 실행 - 실패
usage: ./if1.sh argument
$ ./if1.sh a b c d                        ... 실행 - 성공
Hi! a b c d
```


선택문: if – then – elif – else – fi

- if .. then .. elif .. else .. fi 구문

```
if 조건명령어
then
  명령어 리스트
else
  명령어 리스트
fi
```

```
if 조건명령어
then
  명령어 리스트
elif 조건명령어
then
  명령어 리스트
...
else
  명령어 리스트
fi
```

  - elif : else if 동작

- 예:

```
#!/bin/sh
if [ $# -eq 0 ]
then                                      ... 인수가 없으면
        echo No argument                  ... No argument 출력
else                                      ... 인수가 있으면
        echo Hi! $*                       ... Hi!와 함께 인수 출력
fi
```

```
#!/bin/sh
if [  "$1" == ""  ]; then        # no argument
        ls -l . | wc -l
elif [ ! -e "$1" ]; then         # not exist
        echo $1 does not exist.
elif [ -d "$1" ]; then           # directory
        ls -l "$1" | wc -l
else                             # not directory
        echo $1 is not a directory.
fi
```

선택문 : case .. in .. easc

- case .. in .. easc 구문

```
case 단어 in
  패턴|패턴..) 명령어리스트 ;;     ... 여러 패턴이 같은 명령어 실행
  패턴) 명령어리스트 ;;
  ...
  *) 명령어리스트 ;;              ... default 경우
esac
```


- 예:

```
#!/bin/sh
cat <<MENU                                  ... 메뉴 안내문 출력
  1:    date
  2,4:  print working directory
  3:    print user name
  5-12: print host name
MENU
read -p "Select a menu : " number           ... 메뉴선택 값 입력
case $number in                             ... 입력 값에 따라서 처리
 1)     date ;;
 2|4)   pwd ;;                              ... 2 또는 4
 3)     whoami ;;
 [5-9] | 1[0-2])                            ... 5부터 12까지
        hostname ;;
 *)                                         ... default 경우
        echo illegal menu ;;
esac
```


반복문: for .. do .. done

- for .. do .. done 구문

```
for 변수이름 [in 단어리스트]  ⟶ 생략되면 모든 인수를 리스트로 사용 "$@"
do
  명령어리스트
done
```

- 예:

```
#!/bin/sh
i=1
for color in red gree blue white black
do
        echo color $i : $color
        i=`expr $i + 1`
done
```

```
#!/bin/sh
i=1
for arg
do
        echo argument $i : $arg
        i=`expr $i + 1`
done
```

- $* $@ "$*" "$@"의 차이점 ?

반복문 : while .. do .. done

- while .. do .. done 구문

```
while 조건명령어
do
  명령어리스트
done
```

- 예:

```
#!/bin/sh
i=1
while [ $i -le 10 ]
do
        echo -n "$i "
        i=`expr $i + 1`
done
echo
```

조건명령어로 test가 아닌 명령어 사용

```
#!/bin/sh
i=1
while read line; do
        echo $i : $line
        i=`expr $i + 1`
done
```

반복문 : until .. do .. done

- until .. do .. done 구문

```
until 조건명령어
do
  명령어리스트
done
```

- 예:

```
#!/bin/sh
i=1
until [ $i -gt 10 ]
do
        echo -n "$i "
        i=`expr $i + 1`
done
echo
```


break와 continue

- break, continue – C언어와 비슷한 동작
  - break : 현재 반복 종료, done 다음으로
  - continue : 현재 반복의 done까지의 나머지 부분 생략, 다음 반복으로
- 예: 파일 끝까지 입력을 읽어서 출력, 빈 줄은 출력하지 않음

```
#!/bin/sh
while [ 1 ]                          ... 항상 참 - 무한루프
do
        if ! read line; then         ... 읽을 입력이 없으면(파일 끝)
                break                ... 반복문 종료
        elif [ "$line" == "" ]; then ... 빈 줄이 입력되었으면
                continue             ... 다음 반복으로(echo문 생략)
        fi
        echo $line                   ... 입력된 줄을 출력
done
echo bye
```

while true 사용가능

- break n, continue n – 중첩된 반복문에서 n개 바깥 반복문에 적용

trap 명령어

- trap – signal 처리 명령어 리스트 지정

```
trap '명령어리스트' 시그널번호 ...
```

- 예:

```
#!/bin/sh
trap 'echo interrupted by Control-C 1>&2; exit 1' 2
trap 'echo terminted by kill 1>&2; exit 1' 15
trap 'echo bye' 0

i=1
while [ $i -le 10 ]; do
        echo loop $i
        sleep 2
        i=`expr $i + 1`
done
```

^C

kill에 의한 종료

다양한 변수 접근

형식	동작
${var-default}	변수가 설정되어 있으면 변수 값으로, 그렇지 않으면 default 값으로 대치
${var+alt}	변수가 설정되어 있으면 alt 값으로, 그렇지 않으면 널 문자열로 대치
${var=default}	변수가 설정되어 있으면 변수 값으로, 그렇지 않으면 default 값을 변수에 지정하고 이 값으로 대치
${var?errmsg}	변수가 설정되어 있으면 변수 값으로 대치. 그렇지 않으면 에러 메시지 errmsg를 출력하고, 종료코드 1을 갖고 셸을 종료