고급 Bash 스크립팅 가이드

Bash를 이용한 쉘 스크립팅 완전 가이드

Mendel Cooper
Brindlesoft

thegrendel (at) theriver.com

도운이
차현진

terminus (at) kldp.org

고급 Bash 스크립팅 가이드: Bash를 이용한 쉘 스크립팅 완전 가이드

지은이 Mendel Cooper

도운이 차현진

편냄 2001년 12월 16일

본 튜토리얼은 여러분이 Bash에 대해서 어떠한 사전 지식도 없다고 가정을 합니다만, 금방 중/고급 수준의 명령어들을 소개합니다(**...**유닉스의 슬기롭고 교훈적인 부분들을 배워 나가면서). 이 문서는 교과서나, 혼자 공부할 때 볼 수 있는 메뉴얼, 쉘 스크립팅 기술에 대한 레퍼런스및 소스로 쓰일 수 있습니다. 스크립트를 배우는 유일한 방법은 스크립트를 직접 짜 보는 것이다라는 전제하에, 연습문제와 아주 자세하게 주석 처리된 예제들로 능동적인 독자들의 참여를 유도할 것입니다.

이 문서의 최신 버전은 저자의 홈페이지 (http://personal.riverusers.com/~thegrendel/abs-guide-1.0.tar.gz)에서 SGML 소스와 HTML을 “타르볼”형태로 얻을 수 있습니다. 고침 변경 사항은 change log (http://personal.riverusers.com/~thegrendel/Change.log)를 참고하세요.

고친 과정
고침 0.1 2000년 6월 14일 고친이 mc
초기 릴리스.
고침 0.2 2000년 10월 30일고친이 mc
버그 수정, 내용및 예제 스크립트 추가.
고침 0.3 2001년 2월 12일 고친이 mc
메이저 업데이트.
고침 0.4 2001년 7월 8일 고친이 mc
버그 수정, 더 많은 내용및 예제 추가 - 완전한 책 형태의 개정판.
고침 0.5 2001년 9월 3일 고친이 mc
메이저 업데이트. 버그 수정, 내용 추가, 장과 절을 재편성.
고침 1.0.112001년 12월 16일고친이 mc
버그 수정, 재편성, 내용 추가. Stable release.

바치는 글

모든 마법의 근원인 Anita에게 이 책을 바칩니다.

차례

표 목록

예 목록

Part 1. 소개

쉘은 명령어 해석기(command interpreter)로서, 단지 커널과 사용자 중간에 놓여 있는것 이상으로 꽤 강력한 프로그래밍 언어입니다. 보통 스크립트(script)라고 부르는 쉘 프로그램은 시스템 콜, 여러 프로그래밍 도구들, 유틸리티, 실행파일등을 “묶어서” 어떤 어플리케이션을 쉽게 만들어 줍니다. 사실 모든 종류의 유닉스 명령어, 유틸리티, 도구들이 쉘에서 쓰일 수 있습니다. 만약에 이런 것들로 부족하다면 테스트문이나 루프문등의 쉘 내부 명령어를 써서 추가적인 강력함과 유연함을 얻을 수 있습니다. 쉘 스크립트는 완전히 구조적인 프로그래밍 언어의 편리한 부가 기능들(bells and whistles)이 필요없는 작업들, 특별히 시스템 관리자의 시스템 관련 작업이나 반복적인 일들에 아주 잘 맞습니다.

1장. 왜 쉘 프로그래밍을 해야 하죠?

쉘 스크립트가 어떻게 동작하는지를 이해하는 것은 실력있는 시스템 관리자가 되고 싶어 하는 이들에게는 필수적입니다. 비록 그들이 실제로 스크립트를 작성하지 않는다고 해도 말이죠. 여러분의 리눅스 머신이 부팅될 때를 생각해 봅시다. 부팅이 되면 시스템 설정 정보들을 읽어 들이고 서비스를 구동하기 위해서 `/etc/rc.d`에 있는 쉘 스크립트를 돌립니다. 이 스크립트들을 자세히 이해하는 것은 시스템의 동작을 분석하기 위해서 매우 중요하기도 하지만 나중에 고칠 필요가 있을지도 모르는 일입니다.

쉘 스크립트를 만드는 것은 배우기가 어렵지 않습니다. 왜냐하면, 몇 개의 쉘용 연산자와 옵션들[1] 만으로 아주 작게 만들 수 있기 때문입니다. 쉘 문법은 간단하고 명확합니다. 명령어줄 상에서 명령어를 실행시키거나 유틸리티들을 연결해서 실행시키는 것과 거의 비슷하지만 단지 몇 개의 "규칙"만 배우면 됩니다. 거의 대부분의 스크립트가 한 번에 잘 동작하지만 덩치가 큰 스크립트라도 디버깅하기는 쉽습니다.

쉘 스크립트는 아주 복잡한 어플리케이션을 작성하기 전에 "빠르고 간단한" 프로토타입으로 쓰일 수 있습니다. 스크립트가 원래 하려고 하던 기능보다 제한된 기능만 제공하고 속도가 느리더라도 이는 프로젝트 개발의 첫 단계에 있어 아주 유용합니다. 이렇게 하면 실제로 C, C++, 자바, 펄등으로 마지막 코딩에 들어가기에 앞서 전체 동작 상태를 점검해 볼 수 있기 때문에 전체 구조상의 중요한 결함을 발견할 수도 있습니다.

쉘 스크립팅은 복잡한 일들을 작은 단위로 나누어 처리하거나 여러 요소들과 유틸리티를 묶어 처리하는 고전적인 유닉스 철학을 따릅니다. 많은 사람들은, 펄처럼 모든 이에게 모든 것을 제공하면서 모든 기능을 갖고 있는 신세대 언어를 써서 문제를 푸는데 쓸 시간을 그 도구를 익히는데 쓰게 하는 이런 방법보다는 유닉스식을 더 낫다고 생각하고, 적어도 미적으로는 유닉스식이 더 유쾌한 해결법이라고 생각합니다.

쉘 스크립트를 쓰면 안 될 때

- 리소스에 민감한 작업들, 특히 속도가 중요한 요소일 때(정렬, 해쉬 등등)
- 강력한 산술 연산 작업들, 특히 임의의 정밀도 연산(arbitrary precision)이나 복소수를 써야 할 때(C++이나 포트란을 쓰세요)
- 플랫폼간 이식성이 필요할 때(C를 쓰세요)
- 구조적 프로그래밍이 필요한 복잡한 어플리케이션(변수의 타입체크나 함수 프로토타입등이 필요할 때)
- 업무에 아주 중요하거나 회사의 미래가 걸렸다는 확신이 드는 어플리케이션
- 보안상 중요해서, 여러분 시스템의 무결성을 보장하기 위해 외부의 침입이나 크래킹, 파괴 등을 막아야 할 필요가 있을 때
- 서로 의존적인 관계에 있는 여러 콤포넌트로 이루어진 프로젝트
- 과도한 파일 연산이 필요할 때(Bash는 제한적인 직렬적 파일 접근을 하고 , 특히나 불편하고 불충분한 줄단위 접근만 가능)
- 다차원 배열이 필요할 때

- 링크드 리스트나 트리같은 데이타 구조가 필요할 때
- 그래픽이나 GUI를 만들고 변경하는 등의 일이 필요할 때
- 시스템 하드웨어에 직접 접근해야 할 때
- 포트나 소켓 I/O가 필요할 때
- 예전에 쓰던 코드를 사용하는 라이브러리나 인터페이스를 써야 할 필요가 있을 때
- 독점적이고 소스 공개를 안 하는 어플리케이션을 짜야 할 때(쉘 스크립트는 필연적으로 오픈 소스입니다.)

위에서 얘기한 것중 하나라도 맞는 상황이라면 펄이나 Tcl, 파이썬 같은 다른 스크립팅 언어를 쓰거나 C, C++, 자바 같은 고수준 언어를 고려해 보는게 낫습니다. 어쨌든, 어플리케이션의 프로토타입으로 쉘 스크립트를 쓰는 것은 유용한 개발 단계가 될 것입니다.

우리는 Bash를 사용할 것인데 Bash란 “Bourne-Again Shell”의 앞 글자를 딴 것입니다. 이제는 고전인 된 Stephen Bourne의 Bourne Shell에 대한 말장난 같은 겁니다. Bash는 이제 모든 종류의 유닉스에서 쉘 스크립트에 관한 실질적인 표준(*de facto*)입니다. 이 문서에서 다루고 있는 거의 대부분의 원리들은 Bash가 몇몇 특징을 이어 받은 Korn 쉘 [2] 이나, C 쉘과 그 변형들에도 동일하게 적용됩니다(C 쉘 프로그래밍은 Tom Christiansen이 1993년 10월에 뉴스 그룹 포스팅(http://www.etext.org/Quartz/computer/unix/csh.harmful.gz)을 통해 지적했듯이 타고난 문제점을 갖고 있어서 추천하지 않습니다).

다음부터는 쉘 스크립팅에 대한 튜토리얼입니다. 쉘의 특징들을 설명하기 위해서 최대한 예제들을 통해 접근 했습니다. 예제들은 가능한한 모두 테스트해 보았고, 몇몇은 실제로 쓸 만합니다. 독자 여러분은 이 문서의 소스 아카이브에서 실제 예제를 사용할 수가 있습니다(`something-or-other.sh`). [3] 실행 권한을 주고(**`chmod u+rx scriptname`**), 실행을 시킨 다음 어떤 일들이 일어나는지 살펴보십시오. 소스를 구할 수 없다면 여러분이 보고 있는 HTML이나 pdf, text 버전에서 복사-붙여넣기를 하면 됩니다. 몇몇 예제들은 스크립트들은 설명하기 전에 그 특징을 소개할텐데, 이는 여러분들에게 링크를 따라 이곳 저곳을 왔다 갔다 하게 할 지도 모릅니다.

특별한 언급이 없다면 이 문서에서 쓰인 예제들은 모두 저자가 작성한 것입니다.

주석

1. 이것들은 내부 명령(builtin)이라고 하는 쉘이 갖고 있는 특징입니다.
2. **ksh88**의 많은 특징들과, 업데이트된 **ksh93**의 일부분이 Bash로 통합되었습니다.
3. 관습적으로, 사용자가 작성한 본쉘 호환 스크립트는 보통 `.sh` 확장자를 갖습니다. 반면에 `/etc/rc.d`에서 볼 수 있는 시스템 스크립트는 이런 지침을 따르지 않습니다.

2장. #! 으로 시작하기

쉘 스크립트의 가장 간단한 예는 스크립트 파일에 시스템 명령어들을 단순히 나열해 놓는 것입니다. 이렇게 하면 적어도, 특정한 순서로 명령어를 실행시켜야 할 때 다시 치는 수고를 덜어 줍니다.

예 **2-1. cleanup: /var/log** 에 있는 로그 파일들을 청소하는 스크립트

```
# cleanup
# 루트로 실행시키세요.

cd /var/log
cat /dev/null > messages
cat /dev/null > wtmp
echo "로그를 정리했습니다."
```

별 다른게 없죠? 단순히 콘솔이나 한텀에서 쉽게 실행 시킬 수 있는 명령어들의 조합입니다. 명령어들을 스크립트 상에서 실행시키는 것은 이들을 다시 치지 않아도 된다는 것 이상도 이하도 아닙니다. 스크립트는 특정한 응용에 대해 쉽게 고치고 입맛에 맞게 수정하고 일반화 시킬 수 있습니다.

예 **2-2. cleanup:** 위 스크립트의 향상되고 일반화된 버전**.**

```
#!/bin/bash
# cleanup, version 2
# 루트로 실행시키세요.

LOG_DIR=/var/log
ROOT_UID=0     # $UID가 0인 유저만이 루트 권한을 갖습니다.
LINES=50       # 기본적으로 저장할 줄 수.
E_XCD=66       # 디렉토리를 바꿀 수 없다?
E_NOTROOT=67   # 루트가 아닐 경우의 종료 에러.


if [ "$UID" -ne "$ROOT_UID" ]
then
  echo "이 스크립트는 루트로 실행시켜야 됩니다."
  exit $E_NOTROOT
fi

if [ -n "$1" ]
# 명령어줄 인자가 존재하는지 테스트(non-empty).
then
  lines=$1
else
  lines=$LINES # 명령어줄에서 주어지지 않았다면 디폴트값을 씀.
fi
```

```
#  Stephane Chazelas 가 명령어줄 인자를 확인하는 더 좋은 방법을
#+ 제안해 주었는데 지금 단계에서는 약간 어려운 주젭니다.
#
#    E_WRONGARGS=65  # 숫자가 아닌 인자.(틀린 인자 포맷)
#
#    case "$1" in
#    ""       ) lines=50;;
#    *[!0-9]*) echo "사용법: `basename $0` 정리할파일"; exit $E_WRONGARGS;;
#    *        ) lines=$1;;
#    esac
#
#* 이것을 이해하려면 "루프" 절을 참고하세요.


cd $LOG_DIR

if [ `pwd` != "$LOG_DIR" ]  # 혹은    if [ "$PWD" != "LOG_DIR" ]
                            # /var/log 에 있지 않다?
then
  echo "$LOG_DIR 로 옮겨갈 수 없습니다."
  exit $E_XCD
fi  # 로그파일이 뒤죽박죽되기 전에 올바른 디렉토리에 있는지 두번 확인함.

# 더 좋은 방법은:
# ---
# cd /var/log || {
#   echo "필요한 디렉토리로 옮겨갈 수 없습니다." >&2
#   exit $E_XCD;
# }



tail -$lines messages > mesg.temp # message 로그 파일의 마지막 부분을 저장.
mv mesg.temp messages             # 새 로그 파일이 됨.


# cat /dev/null > messages
#* 위의 방법이 더 안전하니까 필요 없음.

cat /dev/null > wtmp  # > wtemp   라고 해도 같은 결과.
echo "로그가 정리됐습니다."

exit 0
#  스크립트 종료시에 0을 리턴하면
#+ 쉘에게 성공했다고 알려줌.
```

시스템 로그 전체를 날려 버릴 생각이 없을 테니까 여기서는 message 로그의 마지막 부분을 그대로 남겨 놓습니다. 앞으로는 이렇게 앞서 썼던 스크립트를 가공해서 다시 쓰는 식의 좀 더 효과적인 방법을 계속 보게 될 것입니다.

The *#!* 은 스크립트의 제일 앞에서 이 파일이 어떤 명령어 해석기의 명령어 집합인지를 시스템에게 알려주는 역할을 합니다. #! 은 두 바이트 [1] 의 "매직 넘버"(magic number)로서, 실행 가능한 쉘 스크립트라는 것을 나타내는 특별한 표시자입니다(**`man magic`**을 하면 재미있는 주제의 이야기들을 볼 수 있습니다). **#!** 바로 뒤에 나오는 것은 경로명으로, 스크립트에 들어있는 명령어들을 해석할 프로그램의 위치를 나타내는데 그 프로그램이 쉘인지, 프로그램 언어인지, 유틸리티인지를 나타냅니다. 이 명령어 해석기가 주석은 무시하면서 스크립트의 첫 번째 줄부터 명령어들을 실행시킵니다. [2]

```
#!/bin/sh
#!/bin/bash
#!/usr/bin/perl
#!/usr/bin/tcl
#!/bin/sed -f
#!/usr/awk -f
```

각각의 줄은 기본 쉘인 `/bin/sh`이나 기본쉘(리눅스에서는 **bash**), 혹은 다른 명령어 해석기를 부르고 있습니다. [3] 거의 대부분의 상업용 유닉스 변종에서 기본 본쉘인 **`#!/bin/sh`**을 쓰면 비록 Bash 만 가지고 있는 몇몇 기능들을 못 쓰게 되겠지만 리눅스가 아닌 다른 머신에 쉽게 이식(port)할 수 있게 해 줍니다(이렇게 작성된 스크립트는 POSIX [4] **sh** 표준을 따르게 됩니다.

"#!" 뒤에 나오는 경로는 정확해야 합니다. 만약 이를 틀리게 적는다면 스크립트를 돌렸을 때 거의 대부분 "Command not found"라는 에러 메세지만 보게 될 것입니다.

스크립트에서 내부 쉘 지시자를 안 쓰고 일반적인 시스템 명령들만 쓴다면 #!는 안 써도 괜찮습니다. 위의 2번 예제에서는 #!이 필요한데, **`lines=50`**이라는 쉘 전용 생성자를 써서 변수에 값을 대입하고 있기 때문입니다. **`#!/bin/sh`**이 리눅스에서 기본 쉘 해석기인 `/bin/bash`을 부르고 있는 것에 주의하십시오.

> 중요**:** 이 튜토리얼은 스크립트를 만들 때 모듈별 접근 방식을 사용하도록 유도합니다. 나중에 유용하게 쓸 수 있어 보이고 "자주 등장"(boilerplate)하는 코드 조각들을 모아 두세요. 이렇게 모아 두면 나중에 아주 다양하고 매력적인 루틴들을 만들 수 있을 겁니다. 예를 들어, 다음 스크립트 조각은 스크립트 시작 부분에 두어서 원하는 수 만큼의 매개변수를 받았는지 확인하는데 쓰일 수 있습니다.
>
> ```
> if [ $# -ne 원하는_매개변수_갯수 ]
> then
>   echo "사용법: `basename $0` 어쩌구저쩌구"
>   exit $WRONG_ARGS
> fi
> ```

2.1. 스크립트 실행하기

스크립트를 다 만들었고 실행시키려고 한다면 **`sh scriptname`** [5] 이나, **`bash scriptname`**이라고 치면 됩니다. (**`sh <scriptname`**은 스크립트가 표준입력(`stdin`)에서 읽는 것을 사실상 막기 때문에 별로 권장할 만한 방법이 아닙니다.) 더 편한 방법은 chmod를 써서 스크립트 자체를 실행할 수 있게 만드는 것입니다.

이렇게 하거나:

`chmod 555 scriptname` (아무나 읽고/실행 할 수 있게) [6]

아니면

`chmod +rx scriptname` (아무나 읽고/실행 할 수 있게)

`chmod u+rx scriptname` (스크립트 소유자만 읽고/실행할 수 있게)

이렇게 스크립트를 실행할 수 있게 해 놓았다면, **`./scriptname`** [7] 이라고 쳐서 실험해 볼 수 있습니다. 그 스크립트가 “#!”으로 시작한다면 해당하는 명령어 해석기를 불러서 스크립트를 실행 시키게 됩니다.

끝으로, 테스트와 디버깅이 끝난 다음에 여러분과 다른 사용자들이 그 스크립트를 쓸 수 있게 하려면 `/usr/local/bin` 디렉토리로 옮기면 됩니다(당연히 루트로). 이렇게 해 놓으면 명령어 줄에서 간단히 **scriptname[ENTER]**을 치는 것만으로 실행 시킬 수 있습니다.

2.2. 몸풀기 연습문제(Preliminary Exercises)

1. 시스템 관리자들은 일반적인 작업들을 자동으로 하기 위해서 가끔씩 스크립트를 만들어 씁니다. 이런 스크립트로 하기에 좋은 상황들을 몇 개 나열해 보세요.
2. 시간과 날짜, 현재 로그인해 있는 모든 사용자들, 시스템 업타임(uptime)을 보여주는 스크립트를 만들어 보세요. 그 다음에는 로그 파일에 그 정보들을 저장하도록 해 보세요.

주석

1. 몇몇 유닉스 버전(4.2BSD 에 기반한)에서는 매직 넘버로 4 바이트를 받아들이기 때문에 ! 다음에 빈 칸이 필요합니다. **`#! /bin/sh`**.
2. 해당 명령어 해석기(**sh**이나 **bash**)는 #!이 있는 줄을 처음 해석하려고 할텐데, 이 줄은 이미 명령어 해석기를 부르는 자신의 역할을 수행했고 #으로 시작하기 때문에 주석으로 올바르게 해석될 것입니다.
3. 그래서 이렇게 멋진 트릭도 가능해 집니다.

```
#!/bin/rm
```

```
# 자기 자신을 지우는 스크립트.

# 이 스크립트를 실행시키면 이 파일이 지워지는 것 말고는 아무일도 안 생깁니다.

WHATEVER=65

echo "확신하건데, 이 부분은 절대 출력되지 않을 겁니다."

exit $WHATEVER  # 여기서 exit로 빠져 나가지 못하니까 뭐라고 적든 상관없겠죠.
```

재미있는게 또 있는데, README 파일의 시작 부분에 **`#!/bin/more`** 라고 적고 실행 퍼미션을 주면, 자기 스스로 내용을 보여주는 문서 파일이 됩니다.

4. **P**ortable **O**perating **S**ystem **I**nterface, 유닉스류의 OS들을 위한 표준화 작업
5. 주의사항: Bash 스크립트를 **`sh scriptname`**이라고 실행 시키게 되면 Bash 전용의 확장된 기능이 꺼져서 실행이 안 될 수도 있습니다.
6. 쉘이 스크립트를 실행시키려면 스크립트를 읽어야 하기 때문에 실행 퍼미션뿐만 아니라 읽기 퍼미션도 있어야 됩니다.
7. 왜 간단히 **`scriptname`** 이라고 실행 시키지 않을까요? 여러분이 현재 있는 디렉토리($PWD)에 **scriptname**이 있는데도, 왜 실행되지 않을까요? 왜냐하면, 보안상의 이유로 현재 디렉토리를 나타내는 "." 은 사용자의 $PATH에 들어 있지 않기 때문입니다. 따라서 현재 디렉토리에 있는 스크립트를 실행 시키려면 **`./scriptname`**이라고 강제로 실행 경로를 알려줘야 합니다.

Part 2. 기초 단계

3장. 종료와 종료 상태(Exit and Exit Status)

> *사람들은 본쉘의 어두운 면을 모두 사용한다.*
>
> *Chet Ramey*

exit 명령어는 C 프로그램에서처럼 스크립트를 끝낼 때 씁니다. 또한, 스크립트의 부모 프로세스에게 어떤 값을 돌려 줄 수도 있습니다.

모든 명령어는 종료 상태(*exit status* (가끔은 리턴 상태(*return status*)라고도 하는)를 리턴합니다. 명령어가 성공시에는 0을 리턴하고 실패시에는 에러 코드로 해석될 수 있는 non-zero를 리턴합니다. 예외가 있기는 하지만, 유닉스 관례를 잘 따르는 명령어, 프로그램, 유틸리티는 성공했을 때 0을 리턴합니다.

비슷하게, 스크립트의 함수나 스크립트 자신도 종료 상태를 리턴합니다. 스크립트 함수나 스크립트에서 가장 마지막에 실행된 명령어가 종료 상태를 결정합니다. 스크립트에서 **`exit`** ***`nnn`*** 이라고 하면 *nnn*이라는 종료 상태를 쉘에게 전달해 줍니다(*nnn*은 0에서 255 사이의 십진수여야 합니다).

> 참고**:** 매개변수 없이 그냥 **exit**로 끝났을 경우에는, 마지막에 실행된 명령어(**exit** 자신은 빼고)의 종료 상태가 스크립트의 종료 상태가 됩니다.

`$?` 는 제일 마지막 명령어의 종료 상태를 보여줍니다. 함수가 리턴한 다음에 `$?`라고 하면 함수의 마지막 명령어의 종료 상태를 알려줍니다. bash에서는 이렇게 해서 함수의 "반환값"을 돌려 줍니다. 스크립트가 종료한 다음에는 명령어줄에서 `$?`로 스크립트 마지막 명령어의 종료 상태를 알 수가 있는데 관습적으로 **0**은 성공을 나타내고 1에서 255까지의 숫자는 에러를 나타냅니다.

예 **3-1.** 종료/종료 상태

```
#!/bin/bash

echo hello
echo $?    # 명령어가 성공했기 때문에 종료 상태 0이 리턴됨.

lskdf      # 알수없는 명령어.
echo $?    # 0이 아닌 종료 상태가 리턴됨.

echo

exit 113   # 쉘에게 113을 리턴함.
# 확인해 보려면 이 스크립트가 종료된 다음에 "echo $?"라고 쳐 보세요.

# 관습적으로 'exit 0'은 성공을 의미합니다.
# 0이 아닌 값은 에러나 예외상황을 나타냅니다.
```

$? 는 스크립트에서 실행시키 명령어의 결과를 확인하는데 특별히 유용하게 쓰입니다(예 12-8 와 예 12-13 참고).

> 참고**:** 논리적 “부정” 한정어(qualifier)인 ! 는 테스트나 명령어의 결과를 반대로 바꿔서 종료 상태에 영향을 미칩니다.

예 **3-2. !**으로 조건을 부정하기

```
true  # 쉘 내장명령어인 "true".
echo "\"true\"의 종료 상태 = $?"     # 0

! true
echo "\"! true\"의 종료 상태 = $?"   # 1
# 조심할 점은 "!"을 쓸 때, 빈 칸이 있어야 된다는 것입니다.
# 그냥 !true 라고 쓰면 "command not found" 에러가 납니다.

# Thanks, S.C.
```

경고

몇몇 종료 상태 코드들은 예약돼 있기 때문에 사용자가 스크립트에서 마음대로 쓰면 안 됩니다.

4장. 특수 문자

쉘 스크립트에서 쓰이는 특수 문자들

#

주석. #으로 시작하는 줄(#!만 빼고)은 주석입니다.

```
# 이 줄은 주석입니다.
```

명령어 끝에 주석이 나올 수도 있습니다.

```
echo "뒤에 주석이 나옵니다." # 주석이 여기에.
```

줄 첫부분에 나오는 공백문자뒤에도 주석을 쓸 수 있습니다.

```
	# 이 주석 앞에 탭이 있습니다.
```

경고

한 줄에서 주석뒤에 명령어가 올 수는 없습니다. 주석과 “실제 코드”를 구분할 방법이 없기 때문에 다른 명령어는 새로운 줄에 쓰세요.

참고: 당연한 이야기지만, **echo** 문에서 이스케이프된 #은 주석의 시작을 나타내지 않습니다. 비슷하게 몇몇 매개변수 치환이나 산술 상수 확장에 나오는 #도 주석을 나타내지 않습니다.

```
echo "이 # 은 주석의 시작이 아닙니다."
echo '이 # 은 주석의 시작이 아닙니다.'
echo 이 \# 은 주석의 시작이 아닙니다.
echo 이 # 은 주석의 시작을 나타냅니다.

echo ${PATH#*:}       # 매개변수 치환으로, 주석이 아니죠.
echo $(( 2#101011 ))  # 진법 변환, 주석이 아닙니다.

# Thanks, S.C.
```

표준 쿼우팅(quoting)과 이스케이프(escape) 문자들인 (" ' \) 들은 # 을 이스케이프 시킬 수 있습니다.

몇몇 패턴 매칭 연산자도 #을 사용합니다.

;

명령어 구분자. [세미콜론] 두 개 이상의 명령어를 한 줄에서 같이 쓸 수 있게 해줍니다.

```
echo hello; echo there
```

";"는 가끔 이스케이프 시킬 필요가 있습니다.

;;

case 옵션 종료자. [이중 세미콜론]

```
case "$variable" in
abc)  echo "$variable = abc" ;;
xyz)  echo "$variable = xyz" ;;
esac
```

.

"점"**(dot) 명령어.** [마침표] source 명령어와 동일합니다(예 11-14 참고). 이 명령어는 bash 내장 명령(builtin)입니다.

"점"(dot)이 정규 표현식(reqular expression)으로 해석될 때는, 한 개의 문자와 일치됩니다.

또다른 문맥에서는 그냥 **ls** 라고 쳤을 때, 보이지 않는 "숨김" 파일을 나타내는 파일명 접두어로도 쓰입니다.

```
bash$ touch .hidden-file
bash$ ls -l
total 10
 -rw-r--r--    1 bozo      4034 Jul 18 22:04 data1.addressbook
 -rw-r--r--    1 bozo      4602 May 25 13:58 data1.addressbook.bak
 -rw-r--r--    1 bozo       877 Dec 17  2000 employment.addressbook

bash$ ls -al
total 14
 drwxrwxr-x    2 bozo  bozo      1024 Aug 29 20:54 ./
 drwx------   52 bozo  bozo      3072 Aug 29 20:51 ../
 -rw-r--r--    1 bozo  bozo      4034 Jul 18 22:04 data1.addressbook
 -rw-r--r--    1 bozo  bozo      4602 May 25 13:58 data1.addressbook.bak
 -rw-r--r--    1 bozo  bozo       877 Dec 17  2000 employment.addressbook
 -rw-rw-r--    1 bozo  bozo         0 Aug 29 20:54 .hidden-file
```

"

부분 쿼우팅**(partial quoting).** [이중 쿼우트] "문자열" 이라고 하면 쉘이 문자열에 들어 있는 거의 대부분의 특수 문자를 해석하지 못하도록 막아줍니다. 6장을 참고하세요.

'

완전 쿼우팅**(full quoting).** [단일 쿼우트] **’문자열’** 이라고 하면 쉘이 문자열에 들어 있는 모든 특수 문자를 해석하지 못하도록 막아줍니다. "보다 더 강한 형태의 쿼우팅입니다. 6장을 참고하세요.

,

콤마 연산자**.** 콤마 연산자 는 연속적인 산술 연산을 하려고 할 때 쓰입니다. 모든 계산이 이루어진뒤, 마지막에 계산된 결과만 리턴됩니다.

```
let "t2 = ((a = 9, 15 / 3))"  # "a"를 세트하고 "t2"를 계산.
```

\

이스케이프**(escape).** [역슬래쉬] **`\X`**라고 하면 **X** 문자를 “이스케이프” 시키고, **’X’** 라고 “쿼우팅” 시키는 것과 동일한 효과를 갖습니다. \는 "나 ’이 문자 그대로 해석되도록 쿼우트 할 때 쓰일 수도 있습니다.

이스케이프된 문자들에 대한 설명이 6장에 자세하게 되어 있습니다.

/

파일명 경로 구분자**.** [슬래쉬] 파일명에 등장하는 각 요소들을 구분해 줍니다(`/home/bozo/projects/Makefile` 처럼).

나누기 산술 연산자도 됩니다.

`

명령어 치환**(command substitution).** [백틱(backticks)] **‘**명령어**‘** 라고 하면 명령어의 결과를 변수값으로 설정할 수가 있습니다. 다른 말로 backticks나 역쿼우트(backquote)라고도 합니다.

:

널 명령어**(null command).** 쉘의 “NOP”(*no op*, 아무 동작도 않함)에 해당합니다. 쉘 내장 명령인 true의 동의어라고도 볼 수 있습니다. 주의할 점은 :은 bash 내장 명령이기 때문에 종료 상태는 0이라는 것입니다.

```
:
echo $?   # 0
```

무한 루프:

```
while :
do
   첫번째 연산
   두번째 연산
   ...
```

```
   n번째 연산
done

# 이는 다음과 같습니다:
#     while true
#     do
#       ...
#     done
```

if/then 테스트 문의 Placeholder:

```
if condition
then :    # 아무것도 안 하고 계속 진행
else
   어떤 작업
fi
```

이진 연산의 placeholder 제공, 예 8-1 와 디폴트 매개변수 참고.

```
: ${username=`whoami`}
# "username"이 명령어나 내장 명령어가 아닌 경우에
# ${username=`whoami`}    에 : 없이 쓰면 에러가 납니다.
```

here document가 나올 곳의 placeholder를 제공. 예 17-8 참고.

매개변수 치환을 써서 변수의 문자열 평가(예 9-10 참고).

```
: ${HOSTNAME?} ${USER?} ${MAIL?}
# 필수적인 환경변수중 하나라도 세트가 안 돼 있다면 에러를 출력.
```

변수 확장/문자열 조각 대치(**Variable expansion / substring replacement**).

재지향 연산자인 >과 같이 써서 특정 파일의 퍼미션 변경 없이 크기를 0으로 만들어 줍니다. 파일이 없었다면 새로 만들어 냅니다.

```
: > data.xxx    # "data.xxx"은 이제 빈 파일입니다.

# cat /dev/null >data.xxx 라고 한 것과 동일하지만
# ":"가 내장 명령어이기 때문에 새 프로세스를 포크(fork) 시키지 않습니다.
```

예 12-11 참고.

역시 재지향 연산자인 >>과 같이 쓰면 파일의 억세스/변경 시간을 업데이트 해 줍니다(**`: >> new_file`**). 파일이 없었다면 새로 만들어 냅니다. touch와 같습니다.

> 참고**:** 보통 파일에만 사용하고 파이프나 심볼릭 링크, 특수 파일에는 사용하지 마세요.

권장하는 방법은 아닙니다만, 주석의 시작을 나타낼 때 쓸 수도 있습니다. 주석에 #을 쓰게 되면 그 줄의 나머지 부분에 대해서 에러 확인을 안 하기 때문에 어떤 문장도 올 수 있지만 :의 경우는 다릅니다.

```
: 이 주석은 에러를 발생시킵니다, ( if [ $x -eq 3] ).
```

“:”는 또한 `/etc/passwd`와 $PATH 변수에서 필드 구분자로도 쓰입니다.

```
bash$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/sbin:/usr/sbin:/usr/games
```

!

테스트나 종료 상태의 의미를 반대나 부정해 줍니다**.** ! 연산자는 해당 명령어의 종료 상태를 반대로 해 놓습니다(예 3-2 참고). 또한, 테스트 연산자의 의미도 거꾸로 바꿔 주는데 예를 들어, “equal”(=)을 “not-equal” (!=)로 해석하게 해 줍니다. ! 연산자는 bash 키워드입니다.

다른 상황에서는 간접 변수 참조의 의미로도 쓰입니다.

*

와일드 카드**.** [별표] * 문자는 정규 표현식에서 0개 이상의 문자를 나타내는 것과 동일하게 파일명 확장(globbing)에서 “와일드 카드”처럼 쓰입니다.

이중 별표, **, 는 수학의 누승(累乘, exponentiation) 연산자입니다.

?

와일드 카드(하나의 문자)**.** [물음표] ? 문자는 확장 정규 표현식에서 한 문자를 나타내는 것과 마찬가지로 글로빙(globbing)에서 파일명 확장을 나타내는 한 문자짜리 “와일드 카드”의 역할을 합니다.

?은 이중 소괄호에서 C 스타일의 삼중 연산자로도 쓰입니다. 예 9-22를 참고하세요.

$

변수 치환**.**

```
var1=5
var2=23skidoo

echo $var1      # 5
echo $var2      # 23skidoo
```

$은 정규 표현식에서 줄의 끝을 나타냅니다.

${}

매개변수 치환**.**

$*

$@

위치**(positional)** 매개변수**.**

()

명령어 그룹**.**

```
(a=hello; echo $a)
```

> **중요:** 소괄호로 묶인 명령어들은 서브쉘에서 동작합니다.
>
> 스크립트의 다른 곳에서는 소괄호 안의 서브쉘에 들어 있는 변수들을 볼 수가 없습니다. 부모 프로세스인 스크립트는 자식 프로세스(서브쉘)에서 만들어진 변수를 읽을 수가 없습니다.
>
> ```
> a=123
> ( a=321; )
>
> echo "a = $a"   # a = 123
> # 소괄호 안의 "a" 는 지역변수처럼 동작합니다.
> ```

배열 초기화**.**

```
Array=(element1 element2 element3)
```

{xxx,yyy,zzz,...}

중괄호 확장**.**

```
grep Linux file*.{txt,htm*}

# "fileA.txt", "file2.txt", "fileR.html", "file-87.htm" 등등의 파일에서
# "Linux"가 들어 있는 것을 모두 찾음
```

명령어는 중괄호안의 콤마로 분리 지정된 파일 스펙에 맞게 동작할 것입니다. [1] 파일명 확장(globbing)은 중괄호 안에서 지정된 파일 스펙에 적용됩니다.

경고

빈 칸은 쿼우트(quote)나 이스케이프(escape)되지 않고 중괄호에서 쓰일 수 없습니다.

```
echo {file1,file2}\ :{\ A," B",’ C’}
```

```
file1 : A file1 : B file1 : C file2 : A file2 : B
file2 : C
```

{}

코드 블럭. [중괄호] "인라인 그룹"이라고도 부르는 중괄호 한 쌍은 실제로 익명의 함수를 만들어 냅니다만 보통의 함수와는 달리 코드 블럭 안의 변수들을 스크립트의 다른 곳에서 볼 수가 있습니다.

```
bash$ { local a; a=123; }
bash: local: can only be used in a function
```

```
a=123
{ a=321; }
echo "a = $a"   # a = 321   (코드 블럭에서 설정된 값)

# Thanks, S.C.
```

중괄호로 묶인 코드 블럭은 I/O 재지향되거나 재지향을 받을 수 있습니다.

예 **4-1.** 코드 블럭과 **I/O** 재지향

```
#!/bin/bash
# /etc/fstab 읽기

File=/etc/fstab

{
read line1
read line2
} < $File

echo "$File 파일의 첫번째 줄:"
echo "$line1"
echo
echo "$File 파일의 두번째 줄:"
echo "$line2"

exit 0
```

예 **4-2.** 코드 블럭의 결과를 파일로 저장하기

```
#!/bin/bash
# rpm-check.sh

#  rpm 파일에 대해서 설치가능여부, 설치정보, 설치목록에 대해서 쿼리를 하고,
#+ 그 결과를 파일로 저장합니다.
#
# 이 스크립트는 코드 블럭이 어떻게 쓰이는지 보여줍니다.

SUCCESS=0
E_NOARGS=65
```

```
if [ -z "$1" ]
then
  echo "사용법: `basename $0` rpm-file"
  exit $E_NOARGS
fi

{
  echo
  echo "아카이브 정보:"
  rpm -qpi $1       # 설치정보 쿼리.
  echo
  echo "아카이브 목록:"
  rpm -qpl $1       # 설치목록 쿼리.
  echo
  rpm -i --test $1  # 설치가능여부 쿼리.
  if [ "$?" -eq $SUCCESS ]
  then
    echo "$1 는 설치될 수 있습니다."
  else
    echo "$1 는 설치될 수 없습니다."
  fi
  echo
} > "$1.test"       # 블럭의 모든 출력을 파일로 재지향.

echo "$1.test 파일에 rpm 테스트의 결과가 저장되었습니다."

# 여기서 쓰인 옵션에 대한 설명은 맨 페이지를 참고하세요.

exit 0
```

> 참고**:** 위에서 설명했던 (소괄호)로 묶인 명령어 그룹과는 달리 {중괄호}로 묶인 코드 블럭은 보통은 서브쉘을 띄우지 않습니다.. [2]

{} \;

경로명**.** 주로 find에서 쓰이고, 쉘 내장 명령이 아닙니다**.**

> 참고**:** “;” 는 **find** 명령어의 `-exec` 옵션이 여러개 나올 때 끝을 나타내기 때문에 쉘이 해석하는 것을 막기 위해서 이스케이프 시켜줘야 됩니다.

[ ]

테스트**.**

[ ] 사이의 테스트문. **[**는 쉘 내장 명령인 **test**와 동의어로서, 외부 명령어인 `/usr/bin/test`의 링크가 아닙니다**.**

[[ ]]

테스트.

[[ ]] 사이의 테스트문(쉘 키워드).

더 자세한 사항은 [[...]]을 참고하세요.

(( ))

정수 확장.

(( ))에 들어 있는 정수 표현식을 확장하고 평가해 줍니다.

더 자세한 설명은 ((...))를 참고하세요.

> >& >> <

재지향.

`scriptname >filename` 은 `scriptname`의 결과를 `filename`으로 재지향시킵니다. 이 때, `fielname`이 이미 있다면 덮어 써집니다.

`command >&2`는 `command`의 결과를 표준에러로 재지향 시킵니다.

`scriptname >>filename`은 `scriptname`의 결과를 `filename` 으로 덧붙입니다. 이 때, `filename`이 없다면 새로 만듭니다.

프로세스 치환(process substitution).

`(command)>`

`<(command)`

"<" 와 ">" 문자는 다른 문맥에서 문자열 비교 연산자로 동작합니다.

또 다른 문맥에서는 정수 비교 연산자로 동작합니다. 예 12-6를 참고하세요.

<<

here document에서 쓰이는 재지향.

|

파이프. 여러 명령어들을 연결하는 방법으로써, 한 명령어의 출력을 다음 명령어나 쉘에게 전달.

```
echo ls -l | sh
#  "echo ls -l" 의 출력을 쉘에게 전달하는데,
#+ 그냥 "ls -l" 라고 한 것과 똑같습니다.


cat *.lst | sort | uniq
# 모든 ".lst" 파일들을 합친 다음 정렬하고 중복된 줄들을 지웁니다.
```

파이프는 전통적인 프로세스간 통신방법으로, 한 프로세스의 표준출력(`stdout`)을 다른 프로세스의 표준입력(`stdin`)으로 보냅니다. 전형적인 경우에, cat이나 echo같은 명령어들은 데이타의 흐름을 필터에게 넘겨 필터가 처리하도록 합니다.

```
cat $filename | grep $search_word
```

명령어의 출력이나 명령어 자체를 스크립트로 파이프를 걸 수도 있습니다.

```
#!/bin/bash
# uppercase.sh : 입력을 대문자로 바꿔줌.

tr 'a-z' 'A-Z'
#  한 글자짜리 파일 이름이 생기는 걸 막기 위해서
#+ 문자 범위는 꼭 쿼우트 시켜야 합니다.

exit 0
```

자, 이제 **ls -l**의 출력에 파이프를 걸어 이 스크립트로 넘겨 봅시다.

```
bash$ ls -l | ./uppercase.sh
-RW-RW-R--    1 BOZO  BOZO       109 APR  7 19:49 1.TXT
 -RW-RW-R--    1 BOZO  BOZO       109 APR 14 16:48 2.TXT
 -RW-R--R--    1 BOZO  BOZO       725 APR 20 20:56 DATA-FILE
```

참고: 파이프로 묶인 각 프로세스의 표준출력은 다음 명령어의 표준입력으로 읽혀야 합니다. 이런식으로 동작하지 않는다면 데이타의 흐름은 블럭될 것이고 파이프는 생각했던대로 동작하지 않을 것입니다.

```
cat file1 file2 | ls -l | sort
# "cat file1 file2" 의 출력은 나타나지 않습니다.
```

파이프는 자식 프로세스로 돌기 때문에 스크립트의 변수값을 바꿀 수가 없습니다.

```
variable="initial_value"
echo "new_value" | read variable
echo "variable = $variable"     # variable = initial_value
```

파이프로 연결된 명령어중 하나가 취소된다면 전체 실행이 취소되는데 이를 **broken pipe**라고 하고, 이 때 **SIGPIPE** 시그널이 발생 됩니다.

>|

강제 재지향(**noclobber** 옵션이 켜 있더라도). 파일이 이미 존재하더라도 강제로 덮어 쓰게 합니다.

&

작업을 백그라운드로 돌리기. 명령어 뒤에 &를 붙여 주면 백그라운드로 실행됩니다.

```
bash$ sleep 10 &
[1] 850
[1]+  Done                    sleep 10
```

> ### 경고
>
> 스크립트에서 어떤 명령어를 백그라운드로 돌리게 되면 키가 눌리길 기다리면서 스크립트가 멈춰버립니다. 다행스럽게도 이런 상황을 피해 갈 수 있는 방법이 있습니다.

-

표준입력(**stdin**)과 표준출력(**stdout**) 서로간의 재지향. [대쉬]

```
(cd /source/directory && tar cf
 - . ) | (cd /dest/directory && tar xpvf -)
# 한 디렉토리의 전체 파일 구조를 다른 디렉토리로 옮김
# [Alan Cox <a.cox@swansea.ac.uk> 제공, 약간의 수정]
# 1) cd /source/directory    옮겨질 파일들이 있는 소스 디렉토리
# 2) &&                     "And-list": 'cd' 명령이 성공하면 다음 명령어
가 실행됨
# 3) tar cf - .              'c' 옵션은 새 아카이브를 만들라는 명령어
#                            'f'(file) 옵션은 그 뒤에 나오는 '-'에 의해 타
켓 파일을 표준 출력으로 지정해 주고,
#                            현재 디렉토리 트리(.)를 대상으로 하게 합니다.
# 4) |                       파이프를 걸고
# 5) ( ... )                 서브쉘
# 6) cd /dest/directory      옮길 디렉토리로 이동
# 7) &&                      위에서 설명했던 "And-list"
# 8) tar xpvf -              아카이브를 풀고('x'), 소유권과 파일 퍼미션을 유
지('
p')하고
#                            표준출력으로 메세지를 많이(verbose) 찍게 하
고('v')
#                            표준입력에서 데이터를 읽어 들임('f' 다음의 '-')
#
#                            'x'는 명령어고, 'p', 'v', 'f'는 옵션입니
다. 주의하세요.
# 헥헥~~~~



# 똑같지만 더 우아한 방법:
```

```
#   cd source-directory
#   tar cf - . | (cd ../target-directory; tar xzf -)
#
# cp -a /source/directory /dest      도 같은 결과가 나옵니다.

bunzip2 linux-2.4.3.tar.bz2 | tar xvf -
# --tar 파일을 풀어서  --    | --"tar" 에게 넘김  --
# "tar"에 "bunzip2"를 처리하는 패치가 안 돼 있다면
# 파이프를 써서 두 단계로 나누어 처리를 해 줘야 합니다.
# 여기서는 "bzip"으로 압축된 커널 소스를 푸는 것을 보여줍니다.
```

여기서 쓰인 "-"는 Bash 연산자가 아니고, **tar**나 **cat** 같은 몇몇 유닉스 유틸리티들이 인식해서 표준출력으로 쓰도록 해주는 옵션임에 주의하세요.

파일명이 나와야 할 곳에 -이 나오면 표준출력으로 결과를 재지향하든지(`tar cf`에서 가끔 쓰죠), 실제 파일에서 입력을 받지 않고 표준입력에서 받도록 재지향 하게 해 줍니다. 주로 파일을 다루는 유틸리티들을 파이프에서 필터로 쓸 때 이 방법을 씁니다.

```
bash$ file
사용법: file [-bciknvzL] [-f namefile] [-m magicfiles] file...
```

file이 명령어줄에서 옵션없이 불리면 에러 메세지를 내면서 실패합니다.

```
bash$ file -
#!/bin/bash
standard input:              Bourne-Again shell script text executable
```

이번에는 자신의 입력을 표준입력에서 받고 그 결과를 필터링 합니다.

- 는 표준출력을 다른 명령어로 파이프 시키는데 쓰일 수 있습니다. 이렇게 하면 파일의 앞쪽에 줄을 삽입하기같은 묘기도 부릴 수 있습니다.

diff를 써서 섹션을 가진 두 파일을 비교해 보기 바랍니다.

`grep bash file1 | diff file2 -`

마지막으로, tar에 -를 쓴 현실적인 예제입니다.

예 4-3. 최근 하루동안 변경된 파일들을 백업하기

```
#!/bin/bash

#  현재 디렉토리의 모든 파일중 최근 24시간 안에 변경된 파일들을
#+ 타르로 묶고 gzip으로 압축한 "타르볼"로 백업

NOARGS=0
E_BADARGS=65

if [ $# = $NOARGS ]
then
```

```
  echo "사용법: `basename $0` filename"
  exit $E_BADARGS
fi

tar cvf - `find . -mtime -1 -type f -print` > $1.tar
gzip $1.tar


#  너무 많은 파일이 발견되거나 파일명에 빈 칸이 들어있다면 위의 코드는
#+ 실패할 수도 있다고 Stephane Chazelas 가 지적해 주었습니다.

# 그가 제안한 다른 방법은 다음과 같습니다:
# -------------------------------------------------------------------
#   find . -mtime -1 -type f -print0 | xargs -0 tar rvf "$1.tar"
#      "find"의 GNU 버전을 쓴 방법.

#   find . -mtime -1 -type f -exec tar rvf "$1.tar" '{}' \;
#      다른 유닉스에서도 쓸 수 있지만 더 느린 방법.


exit 0
```

경고

파일명이 -로 시작하는 파일이 - 재지향 연산자와 같이 쓰이면 문제가 생길 수도 있습니다. 스크립트 내에서 이런 사항을 확인한 다음에 *./-FILENAME*이나 *$PWD/-FILENAME*등으로 바꿔서 처리해 줘야 합니다.

또한, 변수의 값이 -로 시작할 경우에도 비슷한 문제가 생길 수 있습니다.

```
var="-n"
echo $var
# "echo -n" 처럼 해석이 되어 결과가 안 나
타납니다.
```

-

바로 전 작업 디렉토리. [대쉬] **cd -** 라고 하면 $OLDPWD 환경 변수를 이용해서 바로 전 작업 디렉토리로 옮겨갑니다.

경고

방금 위에서 설명한 재지향 연산자인 “-”와 헷갈리면 안 됩니다. “-”는 문맥에 따라 알맞게 해석됩니다.

-

빼기**.** 산술 연산에서 쓰이는 빼기 부호.

=

Equals. 할당 연산자

```
a=28
echo $a   # 28
```

"="가 다른 문맥에서 쓰이면 문자열 비교 연산자입니다.

+

더하기**.** 덧셈 산술 연산자.

+가 다른 문맥에서 쓰이면 정규 표현식 연산자입니다.

%

나머지**(modulo).** 나눗셈의 나머지 산술 연산자.

%가 다른 문맥에서 쓰이면 패턴 매칭 연산자입니다.

~

홈 디렉토리**.** [틸드] 이 문자는 $HOME 내부 변수에 해당합니다. **~bozo** 는 bozo의 홈 디렉토리를 나타내고 **ls ~bozo** 는 그 홈 디렉토리의 내용을 보여줍니다. ~/ 은 현재 사용자의 홈 디렉토리를 나타내고, **ls ~/** 는 그 홈 디렉토리의 내용을 보여줍니다.

```
bash$ echo ~bozo
/home/bozo

bash$ echo ~
/home/bozo

bash$ echo ~/
/home/bozo/

bash$ echo ~:
/home/bozo:

bash$ echo ~nonexistent-user
~nonexistent-user
```

~+

현재 작업 디렉토리**.** 이 문자는 $PWD 내부 변수에 해당합니다.

~-

바로 전 작업 디렉토리. 이 문자는 $OLDPWD 내부 변수에 해당합니다.

제어 문자

터미널이나 텍스트 디스플레이의 동작을 변경. 제어 문자는 **CONTROL + key** 조합으로 나타낼 수 있습니다.

- **`Ctl-C`**

  포그라운드 작업을 끝냄.

- **`Ctl-D`**

  쉘에서 로그 아웃(exit와 비슷함).

  "EOF" (파일끝, end of file). 표준입력에서 들어오는 입력을 끝냄.

- **`Ctl-G`**

  "벨(BEL)"(삑 소리).

- **`Ctl-H`**

  백스페이스(backspace).

- **`Ctl-J`**

  캐리지 리턴(carriage return).

- **`Ctl-L`**

  폼피드(formfeed, 터미널 화면을 청소). clear 명령어와 똑같은 효과.

- **`Ctl-M`**

  뉴라인(Newline).

- **`Ctl-U`**

  입력줄을 지움.

- **`Ctl-Z`**

  포그라운드 작업을 잠시 멈춤.

공백문자(whitespace)

명령어나 변수의 구분자 역할**.** 공백문자는 빈칸, 탭, 빈줄들의 어떠한 조합들로 이루어져 있습니다. 변수 할당같은 상황에서 공백문자를 쓰면 문법 에러가 납니다.

빈줄은 스크립트 동작에 아무 영향도 주지 않기 때문에 기능별로 구분시켜서 보기 좋게 하는데 쓸 수 있습니다.

특수 변수인 $IFS는 어떤 명령어의 입력 필드를 구분해 주는데 이 변수의 디폴트값은 공백문자입니다.

주석

1. 쉘은 중괄호 확장을 시도할 것이고, 명령어는 확장된 결과에 따라 동작합니다.
2. 예외: 중괄호로 묶인 코드 블럭이 파이프의 일부분으로 돈다면 서브쉘로 돌 수도 있습니다.

```
ls | { read firstline; read secondline; }
# 에러. 중괄호 안의 코드 블럭은 서브쉘로 돌기 때문에 "ls"의 결과가
# 블럭 안의 변수로 전달될 수 없습니다.
echo "첫번째 줄은 $firstline; 두번째 줄은 $secondline"  # 동작하지 않을 겁니다.

# Thanks, S.C.
```

5장. 변수와 매개변수 소개

변수는 모든 프로그래밍 언어와 스크립트 언어에서 핵심입니다. 산술 연산, 양적인 것을 조작하기, 문자열 파싱등에 쓰이고, 심볼(무언가를 나타내는 토큰)을 추상화할 때 없어서는 안 됩니다. 변수는 단지 자신이 가지고 있는 데이타의 컴퓨터 메모리상 위치를 나타냅니다.

5.1. 변수 치환(Variable Substitution)

변수의 이름은 자신이 갖고 있는 데이타의 값을 담는 그릇입니다. 그 변수의 값을 참조하는 것을 변수 치환(variable substitution)이라고 합니다.

$

변수 이름과 그 값을 조심스럽게 살펴보죠. 만약에 **`variable1`**이 변수 이름이라면 **`$variable1`**은 그 변수가 갖고 있는 데이타 아이템인 값을 나타냅니다. 변수를 선언하거나 할당할 때, **unset** 될 때, **export**될 때에만 $없이 쓰입니다. 변수 할당은 =을 쓰거나(**var1=27** 처럼), read 문에서 쓰이거나, 루프문의 처음에서 쓰입니다(**for var2 in 1 2 3**).

참조되는 값을 큰따옴표(" ")로 묶어도 변수 치환이 일어나는 것을 막지 못합니다. 이를 부분적 쿼우팅(partial quoting)이나 “약한 쿼우팅(weak quoting)”이라고 합니다. 작은따옴표를 쓰게 되면 변수 이름이 그냥 문자 그대로 해석되어 아무런 일도 일어나지 않습니다. 이를 완전한 쿼우팅(full quoting)이나 “강한 쿼우팅(strong quoting)”이라고 합니다. 더 자세한 사항은 6장을 참고하세요.

`$variable`은 **`${variable}`**을 짧게 쓴 표현임에 주의하세요. **`$variable`**라고 써서 에러가 났을 경우에는 긴 형태로 써주면 해결될 수도 있습니다(다음에 나오는 9.3절를 참고하세요).

예 **5-1.** 변수 할당과 치환

```
#!/bin/bash

# 변수: 할당과 치환

a=375
hello=$a

#-------------------------------------------------------------------
----
# 변수를 초기화 할 때, = 양쪽에는 빈 칸이 들어가면 안 됩니다.

#  "VARIABLE =value" 라고 하면,
#+ 스크립트는 "VARIABLE" 명령어가 "=value"란 인자를 갖는것처럼 인식합니다.

#  "VARIABLE= value" 라고 하면,
#+ 스크립트는 "value" 명령어가 환경변수 "VARIABLE"을 ""로 세팅해서
```

```
#+ 실행되는 것으로 인식합니다.
#-----------------------------------------------------------------------
----


echo hello    # 변수 참조가 아니고 그냥 "hello"란 문자열입니다.

echo $hello
echo ${hello} # 위와 똑같습니다.

echo "$hello"
echo "${hello}"

echo

hello="A B  C   D"
echo $hello
echo "$hello"
# 이제 echo $hello 와 echo "$hello" 는 다른 결과가 나옵니다.
# 변수를 쿼우팅해주면 공백문자가 보존됩니다.

echo

echo '$hello'
#  작은 따옴표로 변수를 쿼우팅해주면 "$"가 문자 그대로 해석되기 때문에
#+ 변수 참조가 일어나지 않습니다.

# 쿼우팅의 종류에 따라 달라지는 결과에 주목하세요.


hello=    # 널 값을 갖도록 세팅.
echo "\$hello (널 값) = $hello"
#  변수를 널 값으로 세팅하는 것과 unset 하는 것은 비록 결과는 같지만
#+ 엄연히 다릅니다(다음 참조).

# --------------------------------------------------------------

#  여러 변수들을 공백문자로 구분해서 한 줄에서 세트할 수 있습니다.
#  하지만 이렇게 하면 코드의 가독성이 떨어지고
#+ 다른 시스템으로 이식할 수가 없을 수도 있기 때문에 조심해서 써야 됩니다.

var1=variable1  var2=variable2  var3=variable3
echo
echo "var1=$var1   var2=$var2  var3=$var3"

# 예전 버전의 "sh"에서는 문제가 생길수도 있습니다.

# --------------------------------------------------------------
```

```
echo; echo

numbers="one two three"
other_numbers="1 2 3"
# 변수에 공백문자가 들어 있다면 쿼우팅을 해줘야 합니다.
echo "numbers = $numbers"
echo "other_numbers = $other_numbers"
echo

echo "uninitialized_variable = $uninitialized_variable"
# 초기화 안 된 변수는 널 값을 갖습니다(아무 값도 없습니다).
uninitialized_variable=   #  선언만 하고 초기화 안함.
                          #+ (위에서처럼 널 값으로 세팅한 것과 똑같음).
echo "uninitialized_variable = $uninitialized_variable"
                          # 여전히 널 값을 갖고 있네요.

uninitialized_variable=23       # 세트.
unset uninitialized_variable    # 언세트.
echo "uninitialized_variable = $uninitialized_variable"
                                # 여전히 널 값입니다.

echo

exit 0
```

주의

초기화가 안 된 변수는 “null” 값을 갖는데, 이는 값이 할당 안 된 것이지 0 이라는 값을 갖는다는 이야기가 아닙니다! 값을 할당하기 전에 변수를 쓰게 되면 문제가 생길 수 있습니다.

5.2. 변수 할당(Variable Assignment)

=

할당 연산자(앞 뒤에 공백이 있으면 안 됨)

경고

할당이 아니라 테스트 연산자인 = 과 -eq과 헷갈리면 안 됩니다!

= 는 문맥에 따라 할당 연산자나 테스트 연산자로 해석됩니다.

예 5-2. 평범한 변수 할당

```
#!/bin/bash

echo

# When is a variable "naked", i.e., lacking the '$' in front?
# When it is being assigned, rather than referenced.

# 할당할 때
a=879
echo "\"a\" 의 값은 $a 입니다."

# 'let'으로 할당할 때
let a=16+5
echo "\"a\" 의 값은 이제 $a 입니다."

echo

# 'for' 루프에서(실제로는, 일종의 속임수 할당)
echo -n "루프에서 \"a\" 의 값은 "
for a in 7 8 9 11
do
  echo -n "$a 입니다."
done

echo
echo

# 'read' 문에서(역시 일종의 할당임)
echo -n "\"a\" 를 넣으세요."
read a
echo "\"a\" 의 값은 이제 $a 입니다."

echo

exit 0
```

예 5-3. 평범하고 재미있는 변수 할당

```
#!/bin/bash

a=23              # 평범한 경우
echo $a
b=$a
echo $b

# 이제 약간 재밌게 해 봅니다...
```

```
a=`echo Hello¡    # ’echo’ 명령어의 결과를 ’a’ 로 할당
echo $a

a=`ls -l`          # ’ls -l’ 명령어의 결과를 ’a’ 로 할당
echo $a

exit 0
```

$(...) 기법을 써서 변수 할당하기(역따옴표(backquotes)보다 새로운 방법)

```
# /etc/rc.d/rc.local 에서 발췌
R=$(cat /etc/redhat-release)
arch=$(uname -m)
```

5.3. Bash 변수는 타입이 없다(untyped)

Bash 는 다른 프로그래밍 언어들과는 달리, 변수를 “타입”으로 구분하지 않습니다. Bash 변수는 본질적으로 문자열이지만 Bash 가 문맥에 따라서 정수 연산이나 변수를 비교해 줍니다. 이 동작을 결정짓는 요소는 그 변수값이 숫자로만 되어 있는냐 아니냐 입니다.

예 5-4. 정수? 문자열?

```
#!/bin/bash
# int-or-string.sh
# 정수? 문자열?

a=2334                   # 정수.
let "a += 1"
echo "a = $a "           # 여전히 정수죠.
echo

b=${a/23/BB}             # 문자열로 변환.
echo "b = $b"            # BB35
declare -i b             # 정수로 선언해도 도움이 되질 않죠.
echo "b = $b"            # 여전히 BB35.

let "b += 1"             # BB35 + 1 =
echo "b = $b"            # 1
echo

c=BB34
echo "c = $c"            # BB34
d=${c/BB/23}             # 정수로 변환.
echo "d = $d"            # 2334
let "d += 1"             # 2334 + 1 =
echo "d = $d"            # 2335
```

```
# Bash 의 변수는 본질적으로 타입이 없습니다(untyped).

exit 0
```

변수의 타입이 정해져 있지 않다는 것은 축복일수도 있고 재앙일수도 있습니다. 스크립트를 짤 때 충분히 여러분 마음대로 할 수 있는 유연함을 주고, 코드를 짤 때 고생을 덜 하게도 하지만, 알지 못하는 에러가 생길 수도 있으며 적당하게 프로그래밍하는 습관에 물들 수도 있습니다.

Bash 가 이 일들을 해 주지 않기 때문에, 프로그래머가 스크립트에서 쓰이는 변수가 어떤 타입인지 계속 기억하고 있어야 합니다.

5.4. 특수한 변수 타입

지역 변수

지역 변수는 코드 블럭이나 함수에서만 나타납니다(함수에서의 지역 변수 참고).

환경 변수

쉘의 동작과 사용자 인터페이스에 영향을 미치는 변수

참고: 아주 일반적인 상황에서는 각 프로세스는 자신이 참조할 정보들을 담고 있는 변수들의 그룹인 “환경”이란 것을 갖고 있습니다. 이런 관점에서 보면, 쉘도 다른 프로세스들과 다를 바 없습니다.

쉘은 자신이 시작될 때마다 자신의 환경 변수에 대응하는 쉘 변수를 만들어 냅니다. 쉘 변수를 업데이트하거나 새롭게 추가하면 쉘은 자신의 환경 변수를 업데이트 시키고 쉘의 모든 자식 프로세스(쉘에서 실행시킨 명령어들)는 그 환경을 물려 받습니다.

경고

환경용으로 할당된 공간은 제한되어 있습니다. 환경 변수를 너무 많이 만들거나 한 환경 변수가 공간을 지나치게 많이 사용한다면 문제가 생길 수 있습니다.

```
bash$ eval "`seq 10000 | sed -
e 's/.*/export var&=ZZZZZZZZZZZZZ/'`"

bash$ du
bash: /usr/bin/du: Argument list too long
```

(명확한 설명을 해 주고 위의 예제를 제공해준 S. C. 에게 감사)

만약에 스크립트가 환경 변수를 세트하면 그 변수는 “export”되어야 합니다. 즉, 그 스크립트에 해당하는 환경으로 보고되어야 한다는 뜻입니다. export 명령어가 이 일을 해 줍니다.

> **참고:** 스크립트는 변수를 오직 자신의 자식 프로세스에게만 **export** 할 수 있습니다. 즉, 자신이 만들어낸 명령어나 프로세스들만 적용을 받습니다. 명령어줄에서 실행된 스크립트는 특정 변수를 명령어줄 환경으로 거꾸로 export 할 수 *없습니다*. 자식 프로세스는 자신을 만들어낸 부모 프로세스에게 특정 변수를 거꾸로 export 할 수 없습니다.

위치 매개변수(positional parameter)

명령어줄에서 스크립트로 넘어온 인자들로 $0, $1, $2, $3... 이런식으로 표현되는데 $0 은 스크립트의 이름 그 자체를 나타내고 $1 은 첫번째 인자를, $2 는 두번째를, $3 은 세번째를 나타내는 식입니다. [1] $9 다음의 인자는 ${10}, ${11}, ${12} 처럼 중괄호로 묶어줘야 합니다.

예 **5-5.** 위치 매개변수

```
#!/bin/bash

# 최소한 10 개의 매개변수를 줘서 이 스크립트를 실행시키세요. 예를 들면,
# ./scriptname 1 2 3 4 5 6 7 8 9 10

echo

echo "스크립트 이름은 \"$0\"."
# 현재 디렉토리를 나타내는 ./ 가 추가되어 있죠.
echo "스크립트 이름은 \"`basename $0`\"."
# 경로명을 떼어 냅니다('basename' 참고).

echo

if [ -n "$1" ]              # 테스트 할 변수를 쿼우트 해줬습니다.
then
 echo "첫번째 매개변수는 $1"  # # 을 이스케이프 시키기 위해서 쿼우트를 해 줘
야 됩니다.
fi

if [ -n "$2" ]
then
 echo "두번째 매개변수는 $2"
fi

if [ -n "$3" ]
then
 echo "세번째 매개변수는 $3"
```

```
fi

# ...

if [ -n "${10}" ]  # $9 보다 큰 매개변수는 {중괄호}로 감싸야 됩니다.
then
 echo "열번째 매개변수는 ${10}"
fi

echo

exit 0
```

어떤 스크립트들은 자신이 불린 이름에 따라 다르게 동작할 수도 있습니다. 이런 식의 동작을 원한다면 스크립트 안에서 자기가 어떻게 불렸는지를 나타내는 `$0`을 확인하면 됩니다. 스크립트가 어떤 이름으로 불리든지 간에 `$0`은 그 이름을 정확하게 심볼릭 링크하고 있기 때문입니다.

> **작은 정보:** 명령어줄 매개변수를 예상하고 있는 스크립트가 매개변수 없이 불린다면 널 값이 할당되어 원치 않는 결과를 가져옵니다. 이런 상황을 피하는 한 가지 방법은 위치 매개변수를 변수에 할당하는 문장에서 양쪽에 똑같은 아무 문자나 붙이면 됩니다.

```
variable1_=$1_
# 이렇게 해주면 위치 매개 변수가 비어있더라도 에러를 막아줍니다.

critical_argument01=$variable1_

# 실제로 변수를 쓸 곳에서 다음처럼 해주면 에러 방지용 문자가 잘려 나갑니다.
variable1=${variable1_/_/}   # $variable1_ 이 "_"로 시작할 때만 부효
과(Side effects)가 있습니다.
# 9장에서 논의할 매개변수 치환중의 한 가지 방법을 써서 변경 패턴을 삭제해 줍니다.

#  좀 더 확실한 방법은 원하는 위치 매개변수가 넘어 왔는지 테스트를 해 보는 것입니
다.
if [ -z $1 ]
then
  exit $POS_PARAMS_MISSING
fi
```

예 **5-6. wh, whois** 도메인 네임 룩업

```
#!/bin/bash

# ripe.net, cw.net, radb.net 중 하나의 서버에 대해서
# 'whois domain-name' 룩업을 실행
```

```
# 이 스크립트 이름을 'wh' 라고 해서 /usr/local/bin 에 넣어두세요.

# 그리고 다음처럼 심볼릭 링크를 거세요.
# ln -s /usr/local/bin/wh /usr/local/bin/wh-ripe
# ln -s /usr/local/bin/wh /usr/local/bin/wh-cw
# ln -s /usr/local/bin/wh /usr/local/bin/wh-radb


if [ -z "$1" ]
then
  echo "사용법: `basename $0` [domain-name]"
  exit 65
fi

case `basename $0` in
# 스크립트 이름을 확인해서 해당 서버를 부릅니다.
    "wh"     ) whois $1@whois.ripe.net;;
    "wh-ripe") whois $1@whois.ripe.net;;
    "wh-radb") whois $1@whois.radb.net;;
    "wh-cw"  ) whois $1@whois.cw.net;;
    *        ) echo "사용법: `basename $0` [domain-name]";;
esac

exit 0
```

---

shift 명령어는 위치 매개변수를 왼쪽으로 한 단계씩 이동시킵니다.

$1 <--- $2, $2 <--- $3, $3 <--- $4, 등등.

원래의 $1 은 없어지지만 $0 은 바뀌지 않습니다. 위치 매개변수가 많이 필요하다면 {중괄호} 표기법으로 가능하지만(예 5-5 참고), **shift** 를 써서 예전 10개를 쓸 수 있습니다.

예 **5-7. shift** 쓰기

```
#!/bin/bash
# 'shift' 로 모든 위치 매개변수를 처리하기.

#  이 스크립트의 이름을 shft 등의 이름으로 바꾸고
#+ ./shft a b c def 23 skidoo
#+ 같은 식으로 몇 개의 인자를 넘겨서 불러 보세요.

until [ -z "$1" ]  # 모든 매개변수를 다 쓸 때까지...
do
  echo -n "$1 "
  shift
done

echo   # 여분의 한 줄.
```

```
exit 0
```

주석

1. 관습적으로, 스크립트를 호출하는 프로세스는 *$0* 매개변수를, 호출하는 스크립트의 이름으로 세트합니다. **execv** 맨페이지를 참고하세요.

6장. 쿼우팅(quoting)

쿼우팅이란 문자열을 따옴표로 묶는 것을 말합니다. 이렇게 하는 이유는 문자열 안에 특수 문자가 들어가 있을 경우, 쉘이나 쉘 스크립트에 의해 그 특수 문자가 재해석이나 확장되는 것을 방지하기 위해서 입니다.(어떤 문자가 가진 글자 그대로의 뜻과는 다른 해석이 가능한 문자를 “특수 문자”라고 합니다. 예를 들면, 와일드 카드 문자인 *가 특수 문자입니다.)

```
bash$ ls -l [Vv]*
-rw-rw-r--    1 bozo  bozo       324 Apr  2 15:05 VIEWDATA.BAT
 -rw-rw-r--    1 bozo  bozo       507 May  4 14:25 vartrace.sh
 -rw-rw-r--    1 bozo  bozo       539 Apr 14 17:11 viewdata.sh

bash$ ls -l ’[Vv]*’
ls: [Vv]*: No such file or directory
```

참고: 몇몇 프로그램이나 유틸리티들은 쿼우트된 문자열에 들어 있는 특수 문자를 재해석하거나 확장 시킬 수 있습니다. 이것은 쿼우팅의 중요한 사용법으로써 쉘이 명령어줄 매개변수를 해석하지 않고 프로그램이 해석해서 확장하도록 해 줍니다.

```
bash$ grep ’[Ff]irst’ *.txt
file1.txt:This is the first line of file1.txt.
 file2.txt:This is the First line of file2.txt.
```

`grep [Ff]irst *.txt` 라고 하면 당연히 동작하지 않습니다.

변수를 참조할 때는 보통 큰 따옴표(" ")로 묶어 주는게 좋습니다. 이렇게 하면 $, ‘(backquote), \(이스케이프)를 제외한 모든 특수 문자들을 보존해 줍니다. 변수에 쿼우트를 걸어서(*`"$variable"`*) $을 특수 문자로 인식하게 되면 그 변수의 값으로 바꿔 줍니다 (위의 예 5-1 참고).

낱말 조각남(word splitting) [1] 을 피하려면 큰 따옴표를 쓰기 바랍니다. 이렇게 하면 인자에 공백문자가 들어 있어도 하나의 낱말로 인식하게 해 줍니다.

```
variable1="4개의 낱말로 이루어진 변수"
COMMAND 이것은 $variable1 입니다.    # COMMAND는 6개의 인자를 가지고 실행됩니다.
# "이것은" "4개의" "낱말로" "이루어진" "변수" "입니다."

COMMAND "이것은 $variable1 입니다."  # COMMAND는 1개의 인자를 가지고 실행됩니다.
# "이것은 4개의 낱말로 이루어진 변수 입니다."


variable2=""    # 비어 있습니다.
COMMAND $variable2 $variable2 $variable2        # COMMAND는 인자 없이 실행됩
니다.
```

```
COMMAND "$variable2" "$variable2" "$variable2"  # COMMAND는 3개의 빈 인자
를 가지고 실행됩니다.
COMMAND "$variable2 $variable2 $variable2"      # COMMAND는 2개의 빈칸을 가
진 하나의 인자를 가지고 실행됩니다.

# Thanks, S.C.
```

> 작은 정보: **echo**문의 인자를 큰 따옴표로 묶어주는 것은 낱말 조각남이 문제가 될 때에만 필요합니다.

예 6-1. 이상한 변수를 에코하기

```
#!/bin/bash
# weirdvars.sh: 이상한 변수 에코하기.

var="’(]\\{}\$\""
echo $var        # ’(]\{}$"
echo "$var"      # ’(]\{}$"     차이가 없죠?

echo

IFS=’\’
echo $var        # ’(] {}$"     \ 가 빈 칸으로 바뀌었네요.
echo "$var"      # ’(]\{}$"

# S.C. 제공

exit 0
```

작은 따옴표(’ ’)도 큰 따옴표와 비슷하게 동작하지만 $의 특별한 의미를 꺼 버려서 변수 참조가 일어나지 않게 합니다. 작은 따옴표안의 ’을 제외한 모든 특수 문자들은 단순히 문자 그대로 해석됩니다. 작은 따옴표(“완전한 쿼우팅”)를 큰 따옴표(“부분 인용”)보다 좀 더 엄격한 방법이라고 생각하면 됩니다.

> 참고: 작은 따옴표안에서는 이스케이프 문자(\)도 글자 그대로 인식되기 때문에 작은 따옴표로 묶인 문자열에 \을 써서 작은 따옴표 자체를 넣으려고 한다면 원하는 결과가 나오지 않습니다.

```
echo "Why can’t I write ’s between single quotes"

echo
# 간접적인 방법
echo ’Why can’\”t I write ’"’"’s between single quotes’
#    |-------|  |----------|   |-----------------------|
# 이스케이프와 큰 따옴표로 쿼우트된 작은 따옴표에 의해 3개의 문자열로 나뉘어져 있습니다.

# Stephane Chazelas 제공.
```

이스케이프(*Escaping*)는 하나의 문자를 쿼우팅하는 방법입니다. 어떤 문자 앞에 이스케이프 문자(\\)가 오면 쉘에게 그 문자를 문자 그대로 해석하게 해 줍니다.

> **경고**
>
> echo나 sed같은 몇몇 명령어들에서는 이스케이핑이 특수 문자의 특수한 의미를 키도록 해 주는 반대의 효과를 가질 수도 있습니다.

특별한 의미를 갖는 몇 개의 이스케이프 문자들

echo 와 **sed**와 쓰임

\n

   뉴라인(newline)

\r

   리턴(return)

\t

   탭(tab)

\v

   수직탭(vertical tab)

\b

   백스페이스(backspace)

\a

   “경고”(alert, 비프음이나 깜빡거림)

\0xx

   `0xx` 같은 8진수 아스키 표현으로 변환

예 **6-2.** 이스케이프된 문자들

```
#!/bin/bash
# escaped.sh: 이스케이프된 문자들

echo; echo

echo "\v\v\v\v"      # \v\v\v\v 라고 출력
# 'echo'가 이스케이프된 문자들을 출력하게 하려면 -e 옵션을 써야 됩니다.
```

```
echo -e "\v\v\v\v"   # 4 개의 수직탭 출력
echo -e "\042"       # " 출력(따옴표, 8진수 아스키 문자 42).


# Bash 버전 2 이후에서는 $'\xxx' 도 허용됩니다.
echo $'\n'
echo $'\a'
echo $'\t \042 \t'   # 탭으로 둘려쌓인 따옴표(").

# 변수에 아스키 문자를 할당하기.
# ------------------------------
quote=$'\042'        # " 를 변수로 할당.
echo "$quote 여기는 쿼우트된 부분이고, $quote 여기는 안 된 부분입니다."

echo

# 변수에 아스키 문자를 여러개 쓰기.
triple_underline=$'\137\137\137'  # 137 은 "_"의 8진수 아스키 코드.
echo "$triple_underline 밑줄 $triple_underline"

ABC=$'\101\102\103\010'           # 101, 102, 103 은 각각 8진수 A, B, C.
echo $ABC

echo; echo

escape=$'\033'                    # 033 은 이스케이프의 8진수.
echo "\"escape\" echoes as $escape"

echo; echo

exit 0
```

$' ' 문자열 확장의 다른 예제를 보고 싶으면 예 35-1 참고.

\"

큰 따옴표를 그냥 큰 따옴표로 해석

```
echo "Hello"                  # Hello
echo "\"Hello\", he said."    # "Hello", he said.
```

\$

달러 표시를 그냥 달러 표시로 해석(\$ 뒤에 오는 변수는 참조되지 않습니다)

```
echo "\$variable01"  # $variable01 이라고 찍힘
```

\\

백슬래쉬를 그냥 백슬래쉬로 해석

```
echo "\\"  # \ 라고 찍힘
```

참고: \ 의 동작은 자신이 이스케이프 됐는지, 쿼우트 됐는지, here document에서 쓰이는지에 따라 달라집니다.

```
echo \z              #  z
echo \\z             # \z
echo '\z'            # \z
echo '\\z'           # \\z
echo "\z"            # \z
echo "\\z"           # \z
echo `echo \z`       #  z
echo `echo \\z`      #  z
echo `echo \\\z`     # \z
echo `echo \\\\z`    # \z
echo `echo \\\\\\z`  # \z
echo `echo \\\\\\\z` # \\z
echo `echo "\z"`     # \z
echo `echo "\\z"`    # \z

cat <<EOF
\z
EOF                  # \z

cat <<EOF
\\z
EOF                  # \z

# Stephane Chazelas 제공.
```

명령어의 인자 리스트 중간에 나오는 빈칸을 이스케이프 시키면 낱말 조각남을 막을 수 있습니다.

```
file_list="/bin/cat /bin/gzip /bin/more /usr/bin/less /usr/bin/emacs-20.7"
# 명령어에 넘길 인자(들)의 파일 리스트.

# 리스트에 두 개를 더 추가하고 리스트를 보여줌.
ls -l /usr/X11R6/bin/xsetroot /sbin/dump $file_list

echo "----------------------------------------------------------------------
-----"

# 빈칸을 이스케이프 시키면 어떤 일이 일어날까요?
ls -l /usr/X11R6/bin/xsetroot\ /sbin/dump\ $file_list
# 에러: 파일들을 구분 지어줄 빈칸을 이스케이프 시켰기 때문에
#       제일 앞 세개의 파일이 'ls -l' 에게 하나의 인자로 넘어갑니다.
```

이스케이프는 한 명령어를 여러줄에 걸쳐 쓸 수 있게도 해 줍니다. 보통은, 줄이 다르면 다른 명령어를 나타내지만 줄 끝에 이스케이프를 걸면 뉴라인 문자를 이스케이프시키기 때문에 그 다음줄이 원래 줄과 한 줄로 이어지게 됩니다.

```
(cd /source/directory && tar cf - . ) | \
(cd /dest/directory && tar xpvf -)
# 다시 보는 알란 콕스의 디렉토리 트리 복사 명령어
# 여기서는 더 읽기 쉽도록 두 줄로 나누었습니다.

# 이렇게도 가능하죠:
tar cf - -C /source/directory |
tar xpvf - -C /dest/directory
# 밑의 참고를 보세요.
# (Thanks, Stephane Chazelas.)
```

> **참고:** 줄이 파이프 문자인 |으로 끝난다면 굳이 이스케이프 문자(\)를 적어줄 필요가 없습니다. 하지만, 여러줄에 걸친 하나의 명령어에서 줄 끝에 항상 이스케이프 문자를 적어주는 것은 아주 좋은 프로그래밍 습관입니다.

```
echo "foo
bar"
#foo
#bar

echo

echo 'foo
bar'    # 아직은 다른점이 없죠.
#foo
#bar

echo

echo foo\
bar     # 뉴라인이 이스케이프 됐습니다.
#foobar

echo

echo "foo\
bar"     # 약한 쿼우트 안에서는 \ 가 이스케이프 문자로 해석되기 때문에 똑같습니다.
#foobar

echo

echo 'foo\
```

```
bar'     # 강한 쿼우팅 안에서 \ 는 아무 의미 없이 그냥 \ 입니다.
#foor\
#bar

# Stephane Chazelas 제공.
```

주석

1. 여기서 "낱말 조각남"은 문자열이 조각나서 여러개의 인자로 인식되는 것을 뜻합니다.

7장. 테스트

이치에 맞고 완벽한 모든 프로그래밍 언어는 어떤 상태를 테스트하고 그 결과에 따라 동작할 수 있습니다. Bash 는 **test** 명령어, 다양한 중괄호와 소괄호 연산자, **if/then** 을 갖고 있습니다.

7.1. 테스트(Test Constructs)

- **if/then** 은 명령어 목록의 종료 상태가 0 (유닉스 관례상 0은 “성공”을 나타내므로)인지 테스트를 해보고 맞다면 다음 명령어들을 실행시킵니다.
- 테스트 전용 명령어로 **[** (왼쪽 대괄호 특수 문자)란 것이 있습니다. **test** 명령어와 동의어이고, 효율적인 이유로 내장 명령입니다. 이 명령어는 자신의 인자를 비교식이나 파일 테스트로 인식해 해당 연산의 결과에 따른 종료 상태(참은 0, 거짓은 1)를 리턴합니다.
- 또한, 다른 언어를 쓰던 프로그래머에게 어느 정도 더 친숙한 비교 연산을 제공해 주는 [[...]]를 Bash 2.02 버전부터 제공합니다. 주의할 점은 **[[**가 명령어가 아닌 키워드라는 것입니다.

  Bash 는 **`[[ $a -lt $b ]]`** 를 종료 상태를 리턴하는 하나의 요소로 이해합니다.

  ((...)) 와 let ... 은 자신이 계산한 산술식이 0이 아닌 값을 가질 경우에 종료 상태 0을 리턴합니다. 따라서, 이런 산술 확장은 산술 비교를 할 때 쓸 수 있습니다.

  ```
  let "1<2" 는 0 을 리턴("1<2" 가 "1" 로 확장되므로)
  (( 0 && 1 )) 은 1 을 리턴("0 && 1" 이 "0" 으로 확장되므로)
  ```

- **if** 는 대괄호로 조건을 묶지 않고도 아무 명령어나 테스트 할 수 있습니다.

  ```
  if cmp a b > /dev/null  # 결과를 무시.
  then echo "파일 a 와 b 는 같습니다."
  else echo "파일 a 와 b 는 다릅니다."
  fi

  if grep -q Bash file
  then echo "file에는 Bash가 적어도 한 번 이상 나옵니다."
  fi

  if 에러가_없을_때_종료_상태가_0인_명령어
  then echo "성공."
  else echo "실패."
  fi
  ```

- **if/then** 은 중첩된 비교나 테스트가 가능합니다.

  ```
  if echo "다음에 나오는 *if* 는 첫번째 *if* 의 비교 대상에 포함됩니다."

    if [[ $comparison = "integer" ]]
      then (( a < b ))
  ```

```
    else
      [[ $a < $b ]]
    fi

  then
    echo '$a 는 $b 보다 적습니다.'
  fi
```

이 자세한 "**if-test**" 확장 예제들은 **Stephane Chazelas**가 제공해 주었습니다.

예 **7-1.** 무엇이 참인가**?**

```
#!/bin/bash

echo

echo "\"0\" 테스트"
if [ 0 ]      # zero
then
  echo "0 은 참."
else
  echo "0 은 거짓."
fi

echo

echo "\"NULL\" 테스트"
if [ ]        # NULL (empty condition)
then
  echo "NULL 은 참."
else
  echo "NULL 은 거짓."
fi

echo

echo "\"xyz\" 테스트"
if [ xyz ]    # 문자열
then
  echo "임의의 문자열은 참."
else
  echo "임의의 문자열은 거짓."
fi

echo

echo "\"\$xyz\" 테스트"
if [ $xyz ]   # $xyz 가 널인지 테스트...
```

```
              # 하지만 단지 초기화되지 않은 변수일 때만.
then
  echo "초기화 안 된 변수는 참."
else
  echo "초기화 안 된 변수는 거짓."
fi

echo

echo "\"-n \$xyz\" 테스트"
if [ -n "$xyz" ]            # 좀 더 어렵게 보이게.
then
  echo "초기화 안 된 변수는 참."
else
  echo "초기화 안 된 변수는 거짓."
fi

echo

xyz=                        # 널 값으로 초기화.

echo "\"-n \$xyz\" 테스트"
if [ -n "$xyz" ]
then
  echo "널 변수는 참."
else
  echo "널 변수는 거짓."
fi


echo


# "false"가 참일 때.

echo "\"false\" 테스트"
if [ "false" ]              #  "false"는 그냥 문자열 같죠?
then
  echo "\"false\" 는 참."   #+ 그래서 참이 되네요.
else
  echo "\"false\" 는 거짓."
fi

echo

echo "\"\$false\" 테스트"  # 초기화 안 된 변수, 다시.
if [ "$false" ]
then
  echo "\"\$false\" 는 참."
```

```
else
  echo "\"\$false\" 는 거짓."
fi                            # 흠, 이게 우리가 원하던 거죠.


echo

exit 0
```

연습문제. 위에 나온 예 7-1의 동작을 설명해 보세요.

```
if [ condition-true ]
then
   command 1
   command 2
   ...
else
   # 옵션(필요 없다면 빠져도 됩니다).
   # 원래 조건이 실패했을 경우 동작할 코드들을 두세요.
   command 3
   command 4
   ...
fi
```

’if’와 ’then’을 같은 줄에 두려면 세미콜론을 쓰세요.

```
if [ -x "$filename" ]; then
```

Else if 와 elif

elif

`elif`는 else if의 단축형입니다. 바깥쪽 if/then의 안쪽에 중첩해서 쓰는 효과를 가져옵니다.

```
if [ condition1 ]
then
   command1
   command2
   command3
elif [ condition2 ]
# else if 와 같습니다.
then
   command4
   command5
else
   default-command
```

```
fi
```

`if test condition-true`와 **`if [ condition-true ]`** 은 완전히 똑같은 표현입니다. **[** 는 **test** 명령어를 부르는 토큰이기 때문에 **]** 가 꼭 필요하진 않지만 새 버전의 bash 에서는 그래도 있어야 됩니다.

> 참고**: test** 명령어는 파일 타입을 테스트하거나 문자열을 비교해 주는 bash 내장 명령이기 때문에, Bash 스크립트안에서 **test**는 **sh-utils** 패키지의 일부분인 `/usr/bin/test` 외부 명령어를 부르지 않습니다**.** 비슷하게, [도 `/usr/bin/test`로 링크되어 있는 `/usr/bin/[`를 부르지 않습니다.

```
bash$ type test
test is a shell builtin
bash$ type '['
[ is a shell builtin
bash$ type '[['
[[ is a shell keyword
bash$ type ']]'
]] is a shell keyword
bash$ type ']'
bash: type: ]: not found
```

예 7-2. [ ] 와 test 의 동일함

```
#!/bin/bash

echo

if test -z "$1"
then
  echo "명령어줄 인자가 없습니다."
else
  echo "첫번째 명령어줄 인자는 $1 입니다."
fi


if [ -z "$1" ]     # 위의 코드 블럭과 기능적으로 동일합니다.
#   if [ -z "$1"    라고 해도 동작하겠지만...
#+  Bash 는 오른쪽 대괄호가 빠졌다고 에러 메세지를 냅니다.
then
  echo "명령어줄 인자가 없습니다."
else
  echo "첫번째 명령어줄 인자는 $1 입니다."
fi

echo
```

```
exit 0
```

[[ ]] 는 쉘 상에서 [ ]과 동일합니다. 이 명령어는 **ksh88**에서 따 온 확장 테스트 명령어입니다.

> 참고: [[과]] 사이에서는 파일명 확장이나 낱말 조각남이 일어나지 않지만 매개변수 확장이나 명령어 치환은 일어납니다.

```
file=/etc/passwd

if [[ -e $file ]]
then
  echo "비밀번호 파일이 존재합니다."
fi
```

> 작은 정보: **[...]** 말고 **[[...]]** 를 쓰면 많은 논리적 에러들을 막을 수 있습니다. 예를 들어 &&, ||, <, > 연산자들은 [ ] 에서 에러를 내지만 [[ ]] 에서는 잘 동작합니다.

> 참고: **if**다음에 꼭 **test**나 테스트 대괄호([ ]나 [[ ]])가 나오지 않아도 됩니다.
>
> ```
> dir=/home/bozo
> if cd "$dir" 2>/dev/null; then   # "2>/dev/null" 는 에러 메세지를 숨겨줍니다.
>   echo "현재 디렉토리는 $dir 입니다."
> else
>   echo "$dir 로 옮겨갈 수 없습니다."
> fi
> ```
>
> "if COMMAND" 문은 COMMAND의 종료 상태를 리턴합니다.
>
> 비슷하게, 테스트 대괄호가 리스트와 같이 쓰이면 **if** 없이 단독으로 쓰일 수도 있습니다.
>
> ```
> var1=20
> var2=22
> [ "$var1" -ne "$var2" ] && echo "$var1 는 $var2 와 같지 않습니다."
>
> home=/home/bozo
> [ -d "$home" ] || echo "$home 디렉토리는 존재하지 않는 디렉토리입니다."
> ```

(( )) 문은 산술식을 평가해서 확장해 줍니다. 그 산술식이 0 으로 평가되면 종료 상태 1이나 “false”를 리턴하고 0 이 아닌 값으로 평가되면 종료 상태 0이나 “true”를 리턴합니다. 앞에서 얘기했던 **test**나 [ ]와 현격한 차이를 보여줍니다.

예 **7-3.** (( ))로 산술식 테스트 하기

```
#!/bin/bash
# 산술 테스트.
```

```
# (( ... )) 는 산술문을 평가하고 테스트 해 줍니다.
# 종료 상태는 [ ... ] 와 반대입니다!

(( 0 ))
echo "\"(( 0 ))\" 의 종료 상태는 $?."   # 1

(( 1 ))
echo "\"(( 1 ))\" 의 종료 상태는 $?."   # 0

(( 5 > 4 ))                             # 참
echo $?                                 # 0

(( 5 > 9 ))                             # 거짓
echo $?                                 # 1

exit 0
```

7.2. 파일 테스트 연산자

다음 조건이 맞다면 참을 리턴

-e

존재하는 파일

-f

보통 파일(디렉토리나 디바이스 파일이 아님)

-s

파일 크기가 0 이 아님

-d

파일이 디렉토리

-b

파일이 블럭 디바이스(플로피나 시디롬 등등)

-c

파일이 문자 디바이스(키보드, 모뎀, 사운드 카드 등등)

-p

파일이 파이프

-h

파일이 심볼릭 링크

-L

파일이 심볼릭 링크

-S

파일이 소켓

-t

파일 디스크립터가 터미널 디바이스와 연관이 있음

스크립트의 표준입력(**`[ -t 0 ]`**)이나 표준출력(**`[ -t 1 ]`**)이 터미널인지 아닌지를 확인하는데 쓸 수 있습니다.

-r

테스트를 돌리는 사용자가 읽기 퍼미션을 갖고 있음

-w

테스트를 돌리는 사용자가 쓰기 퍼미션을 갖고 있음

-x

테스트를 돌리는 사용자가 실행 퍼미션을 갖고 있음

-g

파일이나 디렉토리에 set-group-id(sgid) 플래그가 세트되어 있음

디렉토리에 *`sgid`* 플래그가 세트되어 있다면 그 디렉토리에서 만들어지는 파일은 파일 생성자의 그룹이 아니라 그 디렉토리 소유자의 그룹에 속하게 됩니다. 이는 워킹그룹이 공유하는 디렉토리에서 유용하게 쓸 수 있습니다.

-u

파일에 set-user-id(suid) 플래그가 세트되어 있음

root가 소유자인 어떤 실행 파일에 *`set-user-id`* 플래그가 세트되어 있다면 일반 사용자가 그 파일을 실행시키더라도 **root** 권한으로 실행됩니다. [1] suid 는 시스템 하드웨어에 접근할 필요가 있는 실행 파일(**pppd**나 **cdrecord** 같은)에 유용합니다. **suid** 플래그가 없다면 이런 실행 파일들은 일반 사용자들이 실행 시킬 수 없습니다.

```
-rwsr-xr-t    1 root       178236 Oct  2  2000 /usr/sbin/pppd
```

`suid` 플래그가 걸린 파일은 퍼미션에 **s**라고 나타납니다.

-k

*스티키 비트(sticky bit)*가 세트

보통은 "스키키 비트"라고 알려져 있는 세이브-텍스트-모드(**save-text-mode**) 플래그는 특별한 형태의 파일 퍼미션입니다. 어떤 파일에 이 플래그가 세트되어 있다면 그 파일은 더 빠른 접근을 위해 캐쉬 메모리에 계속 남아 있습니다. [2] 만약에 디렉토리에 세트되어 있다면 쓰기 퍼미션을 제한합니다. 스티키 비트가 세트되어 있다면 파일이나 디렉토리 퍼미션에 **t** 가 붙어서 보입니다.

```
drwxrwxrwt    7 root         1024 May 19 21:26 tmp/
```

사용자는 자기가 소유하지는 않고 쓰기 퍼미션과 스티키 비트가 세트되어 있는 디렉토리에 오직 자신이 소유한 파일만 지울 수 있습니다. 이렇다면 `/tmp`처럼 공동으로 접근 가능한 디렉토리에서 여러 사용자가 실수로 다른 사용자의 파일을 지우거나 덮어 쓰는 것을 막아 줍니다.

-O

자신이 소유자임

-G

그룹 아이디가 자신과 같음

-N

마지막으로 읽힌 후에 변경 됐음

f1 -nt f2

f1 파일이 *f2* 파일보다 최신임

f1 -ot f2

f1 파일이 *f2* 파일보다 예전것임

f1 -ef f2

f1 파일과 *f2* 파일이 같은 파일을 하드 링크하고 있음

!

"not" -- 앞에서 나왔던 테스트의 의미와 반대(조건이 안 맞으면 참).

예 29-1, 예 10-7, 예 10-3, 예 29-3, 예 A-2 를 보면 파일 테스트 연산자의 사용법을 보여줍니다.

7.3. 비교 연산자(이진)

정수 비교

-eq

같음

```
if [ "$a" -eq "$b" ]
```

-ne

같지 않음

```
if [ "$a" -ne "$b" ]
```

-gt

더 큼

```
if ["$a" -gt "$b" ]
```

-ge

더 크거나 같음

```
if [ "$a" -ge "$b" ]
```

-lt

더 작음

```
if [ "$a" -lt "$b" ]
```

-le

더 작거나 같음

```
if [ "$a" -le "$b" ]
```

<

더 작음(이중 소괄호에서)

```
(("$a" < "$b"))
```

<=

더 작거나 같음(이중 소괄호에서)

```
(("$a" <= "$b"))
```

>

더 큼(이중 소괄호에서)

`(("$a" > "$b"))`

>=

더 크거나 같음(이중 소괄호에서)

`(("$a" >= "$b"))`

문자열 비교

=

같음

`if [ "$a" = "$b" ]`

==

같음

`if [ "$a" == "$b" ]`

= 와 동의어입니다.

```
[[ $a == z* ]]     # $a 가 "z"로 시작하면 참(패턴 매칭)
[[ $a == "z*" ]]   # $a 가 z* 와 같다면 참

[ $a == z* ]       # 파일 globbing이나 낱말 조각남이 일어남
[ "$a" == "z*" ]   # $a 가 z* 와 같다면 참

# Thanks, S.C.
```

!=

같지 않음

`if [ "$a" != "$b" ]`

이 연산자는 [[...]] 에서 패턴 매칭을 사용합니다.

<

아스키 알파벳 순서에서 더 작음

`if [[ "$a" < "$b" ]]`

`if [ "$a" \< "$b" ]`

"<" 가 `[ ]` 에서 쓰일 때는 이스케이프를 시켜야 하는 것에 주의하세요.

\>

아스키 알파벳 순서에서 더 큼

```
if [[ "$a" > "$b" ]]
```

```
if [ "$a" \> "$b" ]
```

">" 가 `[ ]` 에서 쓰일 때는 이스케이프를 시켜야 하는 것에 주의하세요.

이 비교 연산자를 어떻게 응용하는지 예 26-4에서 살펴보세요.

-z

문자열이 "null"임. 즉, 길이가 0

-n

문자열이 "null"이 아님.

> **경고**
>
> 테스트 대괄호에서 **`-n`** 테스트를 받는 문자열은 꼭 쿼우트 시켜야 됩니다. 쿼우트 안 된 문자열을 **`! -z`**와 같이 쓰거나 테스트 대괄호 없이 단독(예 7-5 참고)으로 쓰면 보통은 동작하겠지만, 위험한 습관입니다. 테스트문에서 쓰이는 문자열은 항상 쿼우트를 시켜 주세요. [3]

예 **7-4.** 산술 비교와 문자열 비교

```
#!/bin/bash

a=4
b=5

#  여기서 "a"와 "b"는 정수나 문자열 양쪽 모두로 해석될 수 있습니다.
#  Bash 변수는 타입에 대해 관대하기 때문에
#+ 산술 비교와 문자열 비교에는 약간 애매한 부분이 있습니다.

#  Bash 는 숫자로만 이루어진 변수에 대해서
#+ 산술 비교도 허용하고 문자열 비교도 허용합니다.
#  주의해서 쓰기 바랍니다.

if [ "$a" -ne "$b" ]
then
  echo "$a 와 $b 는 같지 않습니다."
  echo "(산술 비교)"
fi
```

```
echo

if [ "$a" != "$b" ]
then
  echo "$a 는 $b 와 같지 않습니다."
  echo "(문자열 비교)"
fi

# 여기서 "-ne" 와 "!=" 는 둘 다 동작합니다.

echo

exit 0
```

예 **7-5.** 문자열이 널인지 테스트 하기

```
#!/bin/bash
# str-test.sh: Testing null strings and unquoted strings,
# but not strings and sealing wax, not to mention cabbages and kings...
# (옮긴이: ??? :)

# if [ ... ]  를 쓸께요.


#  문자열이 초기화 안 됐다면 정해진 값을 갖지 않는데
#+ 이런 상태를 "널"(null)이라고 부릅니다(0 과는 다릅니다).

if [ -n $string1 ]    # $string1 은 선언도 초기화도 안 됐습니다.
then
  echo "\"string1\" 은 널이 아닙니다."
else
  echo "String \"string1\" 은 널입니다."
fi
# 틀렸죠.
# 초기화가 안 됐는데도 널이 아닌 것으로 나오네요.


echo


# 다시 해보죠.

if [ -n "$string1" ]  # 이번엔 $string1 을 쿼우트 시켜서 해보죠.
then
  echo "\"string1\" 은 널이 아닙니다."
else
  echo "\"string1\" 은 널입니다."
fi      # 테스트문에서는 문자열을 꼭 쿼우트 시키세요!
```

```
echo


if [ $string1 ]       # 이번엔 달랑 $string1 만 두고 해보죠.
then
  echo "\"string1\" 은 널이 아닙니다."
else
  echo "\"string1\" 은 널입니다."
fi
# 이건 잘 되네요.
# 문자열을 쿼우트 시키는 게("$string1") 좋은 습관이긴 하지만
# [ ] 테스트 연산자는 혼자 쓰이면 문자열이 널인지 아닌지를 잘 알아냅니다.
#
# Stephane Chazelas 가 지적한 것처럼,
#    if [ $string 1 ]   는 인자가 "]" 하나고,
#    if [ "$string 1" ]  는 인자가 빈 "$string1"과 "]", 두 개입니다.



echo



string1=initialized

if [ $string1 ]       # , $string1 을 다시 혼자 써보죠.
then
  echo "\"string1\" 은 널이 아닙니다."
else
  echo "\"string1\" 은 널입니다."
fi
# 역시 결과가 맞게 잘 나오죠.
# 마찬가지로 이유로 쿼우트 해주는 것이("$string1") 좋습니다. 왜냐하면...


string1="a = b"

if [ $string1 ]       # $string1 을 또 혼자 씁니다.
then
  echo "\"string1\" 은 널이 아닙니다."
else
  echo "\"string1\" 은 널입니다."
fi
# 이제 "$string1"을 쿼우트 해 주지 않으면 틀린 결과가 나옵니다!

exit 0
# Florian Wisser가 날카롭게 지적해 준 것에도 감사합니다.
```

예 7-6. zmost

```
#!/bin/bash

# ’most’로 gzip으로 묶인 파일을 보기

NOARGS=65
NOTFOUND=66
NOTGZIP=67

if [ $# -eq 0 ] # if [ -z "$1" ] 라고 해도 같음
# $1 이 존재하지만 비어 있을 수 있습니다:  zmost "" arg2 arg3
then
  echo "사용법: `basename $0` filename" >&2
  # 표준에러로 에러 메세지를 출력.
  exit $NOARGS
  # 스크립트의 종료 코드(에러 코드)로 65를 리턴.
fi

filename=$1

if [ ! -f "$filename" ]   # $filename 에 빈 칸이 들어 있을 수도 있기 때문에 쿼우팅해줍니다.
then
  echo "$filename 파일을 찾을 수 없습니다!" >&2
  # 표준출력으로 에러 메세지를 출력.
  exit $NOTFOUND
fi

if [ ${filename##*.} != "gz" ]
# 변수 치환인 중괄호를 사용함.
then
  echo "$1 파일은 gzip 파일이 아닙니다!"
  exit $NOTGZIP
fi

zcat $1 | most

# ’most’ 파일 뷰어 쓰기(’less’와 비슷).
# ’more’나 ’less’로 바꿔도 괜찮음.


exit $?   # 파이프의 종료 상태를 리턴.
#  스크립트는 어떤 경우든지간에 마지막에 실행된 명령어의 종료 상태를
#+ 리턴하기 때문에 실제로는 "exit $?"가 필요없습니다.
```

복합 비교

-a

논리 and

exp1 -a exp2 는 exp1 과 exp2 모두 참일 경우에만 참을 리턴합니다.

-o

논리 or

exp1 -o exp2 는 exp1 이나 exp2 중 하나라도 참일 경우에 참을 리턴합니다.

이것들은 이중 대괄호에서 쓰이는 Bash 비교 연산자인 **&&**와 **||**와 비슷합니다.

```
[[ condition1 && condition2 ]]
```

-o 와 **-a** 연산자는 **test** 명령어나 테스트 대괄호안에서 동작합니다.

```
if [ "$exp1" -a "$exp2" ]
```

복합 비교 연산자가 실제로 쓰이는 예는 예 8-2 와 예 26-7를 참고.

7.4. 중첩된 if/then 조건 테스트

if/then를 쓴 조건 테스트는 중첩될 수도 있고, 그 결과만 보면 위에서 살펴본 **&&** 복합 비교 연산자를 쓴 것과 똑같습니다.

```
if [ condition1 ]
then
  if [ condition2 ]
  then
    do-something  # "conditino1"과 "condition2"가 모두 참일 경우에만
  fi
fi
```

중첩된 *if/then* 조건 테스트 예제는 예 35-3를 참고.

7.5. 여러분이 테스트문을 얼마나 이해했는지 테스트 해보기

시스템 전체에서 쓰이는 `xinitrc` 파일은 X 서버를 띄우는데 쓰일 수 있습니다. 이 파일에는 아주 많은 **if/then** 테스트가 나오는데 다음은 그 중 일부분입니다.

```
if [ -f $HOME/.Xclients ]; then
```

```
  exec $HOME/.Xclients
elif [ -f /etc/X11/xinit/Xclients ]; then
  exec /etc/X11/xinit/Xclients
else
     # 아마 이 부분은 절대 실행되지 않겠지만 혹시 모르니까.
     # (Xclients 에도 역시 안전 장치가 걸려 있습니다) 전혀 해가 없습니다.
     xclock -geometry 100x100-5+5 &
     xterm -geometry 80x50-50+150 &
     if [ -f /usr/bin/netscape -a -
f /usr/share/doc/HTML/index.html ]; then
             netscape /usr/share/doc/HTML/index.html &
     fi
fi
```

위 코드에 나오는 "테스트"를 설명해 보세요. 그 다음에는 `/etc/X11/xinit/xinitrc` 파일 전체를 살펴보고 거기에 나오는 **if/then** 를 분석해 보세요. 뒤에서 설명할 grep, sed, 정규 표현식을 참조해야 할지도 모릅니다.

주석

1. **suid**가 걸려 있는 파일은 보안 구멍을 가질 수 있기 때문에 주의해야 합니다. 쉘 스크립트에는 **suid** 플래그가 아무 영향도 미치지 않습니다.
2. 현대의 유닉스 시스템에서는 스티키 비트가 더 이상 파일에는 쓰이지 않고 오직 디렉토리에만 쓰입니다.
3. S.C. 가 지적한대로 복합 테스트에서는 쿼우팅만으로 충분치 않을 수도 있습니다. **`[ -n "$string" -o "$a" = "$b" ]`**의 경우 `$string`이 비어 있을 경우, 몇몇 Bash 버전에서는 에러를 낼 수도 있습니다. 안전하게 가려면 비어 있는 값을 가질 가능성이 있는 변수에 글자를 덧붙여 주면 됩니다. **`[ "x$string" != x -o "x$a" = "x$b" ]`**("x's"는 모두 상쇄됩니다).

8장. 연산자 이야기(Operations and Related Topics)

8.1. 연산자(Operators)

할당(assignment)

변수 할당

변수값을 초기화 하거나 변경하기

=

산술식과 문자열 할당을 해주는 다기능 할당 연산자.

```
var=27
category=minerals  # "=" 다음에 빈 칸이 들어가면 안 됩니다.
```

경고

“=” 할당 연산자와 = 테스트 연산자를 헷갈리면 안 됩니다.

```
#    테스트 연산자인 =

if [ "$string1" = "$string2" ]
# if [ "Xstring1" = "Xstring2" ] 라
고 해야 변수 값이 비어 있을 경우에
# 발생할지도 모를 에러를 막아주기 때문
에 더 안전합니다.
# "X"는 상쇄되어 없어집니다.
then
   command
fi
```

산술 연산자(arithmetic operators)

+

더하기

-

빼기

*

곱하기

/

나누기

**

누승(exponentiation)

```
# Bash 2.02 버전에서 누승 연산자인 "**"가 소개됐습니다.

let "z=5**3"
echo "z = $z"   # z = 125
```

%

modulo 나 mod (정수 나누기에서 나머지 값)

```
bash$ echo `expr 5 % 3`
2
```

이 연산자는 특정 범위에 속하는 숫자를 만들어 내거나(예 9-19, 예 9-20) 결과를 특정 포맷으로 만들어 주거나(예 26-6) 심지어는 소수(prime number)를 만들어내는데(예 A-11) 쓸 수도 있습니다.

+=

"plus-equal"(상수값 만큼 증가)

`let "var += 5"` 은 `var`의 값을 `5`만큼 증가시킵니다.

-=

"minus-equal"(상수값 만큼 감소)

*=

"times-equal"(상수값을 곱함)

`let "var *= 4"` 는 `var`의 값에 `4`배를 해 줍니다.

/=

"slash-equal"(상수값으로 나눔)

%=

"mod-equal"(상수값으로 나눈 나머지 값)

산술 연산자는 종종 **expr** 이나 **let** 식에서 쓰입니다.

예 **8-1.** 산술 연산자 쓰기

```
#!/bin/bash
# 5가지의 다른 방법으로 6까지 세기.

n=1; echo -n "$n "

let "n = $n + 1"   # let "n = n + 1"   이라고 해도 됩니다.
echo -n "$n "

: $((n = $n + 1))
#  ":" 가 없으면 Bash가 "$((n = $n + 1))"을
#+ 명령어로 해석하려고 하기 때문에 필요합니다.
echo -n "$n "

n=$(($n + 1))
echo -n "$n "

: $[ n = $n + 1 ]
#  ":" 가 없으면 Bash가 "$[n = $n + 1]"을
#+ 명령어로 해석하려고 하기 때문에 필요합니다.
#  "n"이 문자열로 초기화 되어 있어도 동작합니다.
echo -n "$n "

n=$[ $n + 1 ]
#  "n"이 문자열로 초기화 되어 있어도 동작합니다.
#* 이런 형태는 더 이상 지원되지 않고 이식성도 없기 때문에 쓰지 마세요.
echo -n "$n "; echo

# Thanks, Stephane Chazelas.

exit 0
```

참고**:** Bash 의 정수 변수는 실제로는 범위가 -2147483648 에서 2147483647 까지인 signed **long**(32-bit) 정수입니다. 그렇기 때문에 이 범위를 벗어나는 변수를 쓰면 에러가 납니다.

```
a=2147483646
echo "a = $a"      # a = 2147483646
let "a+=1"         # "a" 를 증가.
echo "a = $a"      # a = 2147483647
let "a+=1"         # "a" 를 다시 증가, 제한 범위를 벗어남.
echo "a = $a"      # a = -2147483648
                   #      ERROR (out of range)
```

경고

Bash 는 부동 소수점 연산을 이해하지 못합니다. 점이 들어간 숫자는 단순히 문자열로 취급됩니다.

```
a=1.5

let "b = $a + 1.3"  # 에러.
# t2.sh: let: b = 1.5 + 1.3: syntax error in ex-
pression (error token is ".5 + 1.3")
echo "b = $b"       # b=1
```

부동 소수점 연산이나 수학 함수들이 필요하다면 bc 를 쓰세요.

비트 연산자**(bitwise operators).** 비트 연산자는 쉘 스크립트에서 자주 쓰이지 않습니다. 이런 연산자들은 주로 포트나 소켓에서 값을 읽고 테스트하고 조작하는데 쓰입니다. “비트 조작(bit flipping)”은 C나 C++ 처럼 실행중에 비트연산을 할 수 있을 정도로 충분히 속도가 빠른 컴파일 언어가 더 알맞습니다.

비트 연산자

<<

비트 왼쪽 쉬프트(쉬프트 한 번당 2를 곱하는 것과 동일함)

<<=

“left-shift-equal”

`let "var <<= 2"` 는 `var`를 2 비트만큼 왼쪽으로 쉬프트(4 를 곱하는 것과 동일함)

>>

비트 오른쪽 쉬프트(쉬프트 한 번당 2로 나눔)

>>=

“right-shift-equal”(<<=와 반대)

&

비트 and

&=

“비트 and-equal”

|

비트 OR

|=

“비트 OR-equal”

~

비트 negate

!

비트 NOT

^

비트 XOR

^=

“비트 XOR-equal”

논리 연산자(logical operators)

&&

and (논리)

```
if [ $condition1 ] && [ $condition2 ]
# if [ condition1 -a condition2 ] 와 같음
# condition1과 condition2가 모두 참일경우 참을 리턴...

if [[ $condition1 && $condition2 ]]    # 역시 동작함.
# && 연산자는 [ ... ] 에서 쓰일 수 없습니다. 주의하세요.
```

> **참고:** && 는 상황에 따라 명령어들을 연결해 주는 and 리스트로도 쓰일 수도 있습니다.

||

or (논리적)

```
if [ $condition1 ] || [ $condition2 ]
# if [ condition1 -o condition2 ] 와 같음
# condition1이나 condition2중 하나만 참이면 참을 리턴...

if [[ $condition1 || $condition2 ]]    # 역시 동작함.
# || 연산자는 [ ... ] 에서 쓰일 수 없습니다. 주의하세요.
```

> **참고:** Bash는 논리 연산자로 연결된 각 문장의 종료 상태를 테스트합니다.

예 **8-2. && 와 ||** 를 쓴 복합 조건 테스트

```
#!/bin/bash

a=24
b=47

if [ "$a" -eq 24 ] && [ "$b" -eq 47 ]
then
  echo "첫번째 테스트 성공."
else
  echo "첫번째 테스트 실패."
fi

# 에러:    if [ "$a" -eq 24 && "$b" -eq 47 ]
#          라고 하면 ' [ "$a" -eq 24 ' 를 실행시키려고 하기 때문에
#          해당하는 ']' 를 찾지 못해 에러가 납니다.
#
#    if [[ $a -eq 24 && $b -eq 24 ]]   라고 하면 되겠죠.
#    (6번째 줄과 17번째 줄의 "&&" 는 다른 뜻을 갖습니다.)
#    Thanks, Stephane Chazelas.


if [ "$a" -eq 98 ] || [ "$b" -eq 47 ]
then
  echo "두번째 테스트 성공."
else
  echo "두번째 테스트 실패."
fi


# 복합문 조건 테스트에서는 -a 와 -o 옵션을 쓸 수도 있습니다.
# 이 점을 지적해준 Patrick Callahan 에게 감사합니다.


if [ "$a" -eq 24 -a "$b" -eq 47 ]
then
  echo "세번째 테스트 성공."
else
  echo "세번째 테스트 실패."
fi


if [ "$a" -eq 98 -o "$b" -eq 47 ]
then
  echo "네번째 테스트 성공."
else
  echo "네번째 테스트 실패."
fi
```

```
a=rhino
b=crocodile
if [ "$a" = rhino ] && [ "$b" = crocodile ]
then
  echo "다섯번째 테스트 성공."
else
  echo "다섯번째 테스트 실패."
fi

exit 0
```

&& 와 || 연산자는 산술식에서도 쓰일 수 있습니다.

```
bash$ echo $(( 1 && 2 )) $((3 && 0
)) $((4 || 0)) $((0 || 0))
1 0 1 0
```

기타 연산자(miscellaneous operators)

,

콤마 연산자(comma operator)

콤마 연산자는 두 개 이상의 산술 연산을 묶어 줍니다. 이 연산자로 묶인 모든 연산은 부효과(side effects)가 생길 가능성을 가지고 평가되고 제일 마지막 연산의 결과만 리턴됩니다.

```
let "t1 = ((5 + 3, 7 - 1, 15 - 4))"
echo "t1 = $t1"               # t1 = 11

let "t2 = ((a = 9, 15 / 3))"  # "a" 를 세트하고 "t2" 를 계산.
echo "t2 = $t2    a = $a"     # t2 = 5    a = 9
```

콤마 연산자는 주로 for 루프에서 쓰입니다. 예 10-11를 참고하세요.

8.2. 숫자 상수(Numerical Constants)

쉘 스크립트는 특별한 접두사나 표기법이 없다면 숫자를 10진수로 해석합니다. 만약에 숫자 앞에 *0*이 있다면 *8진수* (8진법)고, *0x*이 있으면 *16진수*(16진법)입니다. #이 들어간 숫자는 *진법#숫자*로 계산됩니다(이 옵션은 범위 제한이 있기 때문에 유용하지 않습니다).

예 **8-3.** 숫자 상수 표기법**:**

```
#!/bin/bash
# numbers.sh: 숫자 표시법.

# 10진수
let "d = 32"
echo "d = $d"
# 별로 특별한 게 없네요.


# 8진수: '0' 다음에 나오는 숫자
let "o = 071"
echo "o = $o"
# 결과는 10진수로 나타납니다.

# 16진수: '0x'나 '0X' 다음에 나오는 숫자
let "h = 0x7a"
echo "h = $h"
# 결과는 10진수로 나타납니다.

# 다른 진법: 진수#숫자
# 진수는 2 와 36 사이가 올 수 있습니다.
let "b = 32#77"
echo "b = $b"
#
# 이 표기법은 아주 제한된 범위의 숫자(2 - 36)에서만 동작합니다.
                # ... 10 개의 숫자 + 26 개의 알파벳 문자 = 36.
let "c = 2#47"  # 범위 초과 에러:
# numbers.sh: let: c = 2#47: value too great for base (error to-
ken is "2#47")
echo "c = $c"

echo

echo $((36#zz)) $((2#10101010)) $((16#AF16))

exit 0
# 자세한 설명 고마워요. S.C.
```

Part 3. 중급 단계(Beyond the Basics)

9장. 변수 재검토(Variables Revisited)

적당히 잘 쓰인 변수는 스크립트에 강력함과 유연함을 줄 수 있기 때문에 변수의 미묘한 차이점과 뉘앙스에 대해 자세히 배울 필요가 있습니다.

9.1. 내부 변수(Internal Variables)

내장(Builtin) 변수

Bash 스크립트의 동작에 영향을 미치는 변수

`$BASH`

Bash 실행 파일의 경로로, 보통은 `/bin/bash`임

`$BASH_ENV`

스크립트가 실행될 때 어디에서 bash 시작 파일을 읽을 것인지를 나타내는 환경 변수

`$BASH_VERSINFO[n]`

원소 갯수가 6개인 배열로, 현재 설치된 Bash 버전에 대한 정보를 담고 있습니다. 다음에 설명할 `$BASH_VERSION` 과 비슷하지만 좀 더 자세한 정보를 담고 있습니다.

```
# Bash 버전 정보:

for n in 0 1 2 3 4 5
do
  echo "BASH_VERSINFO[$n] = ${BASH_VERSINFO[$n]}"
done

# BASH_VERSINFO[0] = 2                      # Major version no.
# BASH_VERSINFO[1] = 04                     # Minor version no.
# BASH_VERSINFO[2] = 21                     # Patch level.
# BASH_VERSINFO[3] = 1                      # Build version.
# BASH_VERSINFO[4] = release                # Release status.
# BASH_VERSINFO[5] = i386-redhat-linux-gnu  # Architecture
                                            # ($MACHTYPE 과 동일).
```

`$BASH_VERSION`

시스템에 설치된 Bash 버전

```
bash$ echo $BASH_VERSION
2.04.12(1)-release
```

```
tcsh% echo $BASH_VERSION
BASH_VERSION: Undefined variable.
```

어떤 쉘로 동작중인지 알아보려고 할 때 $BASH_VERSION 을 확인해 보는 것은 아주 좋은 방법입니다. 왜냐하면, $SHELL 로는 충분한 정보를 얻을 수 없기 때문입니다.

`$DIRSTACK`

디렉토리 스택의 내용(pushd와 popd의 영향을 받음)

이 내장 변수는 dirs 명령어와 짝을 이룹니다.

`$EDITOR`

스크립트가 부르는 에디터로서, 보통은 **vi**나 **emacs**입니다.

`$EUID`

"유효" 사용자 아이디 값

su에 의해 쓰일 현재 사용자의 유효 아이디 값.

> **경고**
>
> `$EUID`는 $UID와 반드시 같지 않습니다.

`$FUNCNAME`

현재 함수의 이름

```
xyz23 ()
{
  echo "$FUNCNAME now executing."  # xyz23 now executing.
}

xyz23

echo "FUNCNAME = $FUNCNAME"        # FUNCNAME =
                                   # 함수 밖에서는 널 값을 갖습니다.
```

`$GLOBIGNORE`

globbing 시 포함되지 않을 파일명 패턴들의 목록.

`$GROUPS`

현재 사용자가 속해 있는 그룹

`/etc/passwd`에 적혀 있는 현재 사용자의 그룹 아이디 값을 보여줍니다.

`$HOME`

사용자의 홈 디렉토리로, 보통은 `/home/username` (예 9-10 참고)

`$HOSTNAME`

hostname 명령어는 부팅시 init 스크립트에서 시스템 이름을 설정해 줍니다. 하지만 `gethostname()` 함수로 bash 내부 변수인 `$HOSTNAME`을 설정해 줄 수도 있습니다. 예 9-10을 참고하세요.

`$HOSTTYPE`

host type

$MACHTYPE과 마찬가지로 시스템 하드웨어를 알려줍니다.

```
bash$ echo $HOSTTYPE
i686
```

`$IFS`

입력 필드 구분자

디폴트는 공백문자(빈칸, 탭, 뉴라인)지만 콤마로 구분된 데이타 파일을 파싱하려는 경우처럼 변경이 가능합니다. $*는 `$IFS`의 첫번째 문자를 사용하는 것에 주의하세요. 예 6-1 참고.

```
bash$ echo $IFS | cat -vte
$


bash$ bash -c 'set w x y z; IFS=":-;"; echo "$*"'
w:x:y:z
```

경고

`$IFS` 는 빈 칸을 다른 문자들과는 다르게 처리합니다.

예 9-1. $IFS 와 빈 칸

```
#!/bin/bash
# $IFS 는 공백문자를 다른 문자들과 다르
게 처리합니다.

output_args_one_per_line()
{
  for arg
  do echo "[$arg]"
  done
}

echo; echo "IFS=\" \""
echo "-------"

IFS=" "
var=" a  b c   "
output_args_one_per_line $var  # out-
put_args_one_per_line `echo " a  b c   "`
#
# [a]
# [b]
# [c]


echo; echo "IFS=:"
echo "-----"

IFS=:
var=":a::b:c:::"               # 위
와 같지만 ":" 를 " "로 바꿔줍니다.
output_args_one_per_line $var
#
# []
# [a]
# []
# [b]
# [c]
# []
# []
# []

# awk 의 "FS" 필드 구분자도 위와 같은 동
작을 합니다.

# Thank you, Stephane Chazelas.

echo

exit 0
```

(S. C. 에게 정확한 설명과 예제에 감사.)

`$IGNOREEOF`

EOF 무시: 로그 아웃하기 전에 몇 개의 end-of-files (control-D)를 무시할 것인지.

`$LC_COLLATE`

주로 `.bashrc` 나 `/etc/profile` 에서 세트되는 이 변수는 파일명 확장이나 패턴 매칭시의 대조(collation) 순서를 제어합니다. 이 값이 잘못 처리되면 파일명 globbing시 원치 않는 결과를 가져 올 수 있습니다.

> 참고**:** Bash 2.05 이후로, 파일명 globbing 은 대괄호에 들어 있는 문자 범위에 나타나는 대소문자를 더 이상 구분하지 않습니다. 예를 들어, **ls [A-M]*** 은 `File2.txt`와 `file1.txt` 모두와 일치될 것입니다. 자신만의 대괄호 매칭 동작을 원래대로 바꾸려면 `/etc/profile`이나 `~/.bashrc`등에 **`export LC_COLLATE=C`** 라고 해서 `LC_COLLATE`을 `C`로 세트해 주면 됩니다.

`$LC_CTYPE`

이 내부 변수는 globbing 과 패터 매칭의 문자 해석을 제어합니다.

`$LINENO`

쉘 스크립트에서 이 변수가 들어 있는 줄의 줄번호를 나타내는데, 스크립트에서 쓰일 때만 의미가 있고, 주로 디버깅에 쓰입니다.

```
last_cmd_arg=$_  # 저장.

echo "$LINENO 번째 줄, 변수 \"v1\" = $v1"
echo "처리된 마지막 명령어 인자 = $last_cmd_arg"
```

`$MACHTYPE`

머신 종류

시스템 하드웨어를 구분해 줍니다.

```
bash$ echo $MACHTYPE
i686-debian-linux-gnu
```

`$OLDPWD`

바로 전 작업 디렉토리 (“OLD-print-working-directory”)

`$OSTYPE`

운영 체제 종류

```
bash$ echo $OSTYPE
linux-gnu
```

`$PATH`

실행 파일의 경로, 보통은 `/usr/bin/`, `/usr/X11R6/bin/`, `/usr/local/bin`, 등등.

명령어가 주어지면 쉘은 실행 파일들을 위한 경로에 들어있는 디렉토리들에 대해서 자동으로 해쉬 테이블 탐색을 수행합니다. 경로에는 환경 변수인 `$PATH`에 디렉토리들이 콜론으로 구분되어 들어 있습니다. 보통 `$PATH` 정의는 `/etc/profile`이나 `~/.bashrc`에 들어 있습니다(27장 참고).

```
bash$ echo $PATH
/bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin:/sbin:/usr/sbin
```

`PATH=${PATH}:/opt/bin` 라고 하면 `/opt/bin`을 현재 경로에 추가 시킵니다. 이렇게 하면 경로에 임시로 디렉토리를 적절하게 추가할 수 있습니다. 스크립트가 끝나면 원래 `$PATH` 로 복구됩니다(스크립트같은 자식 프로세스는 부모 프로세스인 쉘의 환경 변수를 바꿀 수 없습니다).

> 참고: 보통은 현재 “작업 디렉토리”를 나타내는 `./`은 보안상의 이유로 `$PATH`에서 제외시킵니다.

`$PIPESTATUS`

마지막으로 실행된 파이프의 종료 상태. 아주 재미있는 것은, 이 결과와 마지막으로 실행된 명령어의 종료 상태가 똑같지는 않다는 것입니다.

```
bash$ echo $PIPESTATUS
0

bash$ ls -al | bogus_command
bash: bogus_command: command not found
bash$ echo $PIPESTATUS
141

bash$ ls -al | bogus_command
bash: bogus_command: command not found
bash$ echo $?
127
```

`$PPID`

어떤 프로세스의 부모 프로세스의 프로세스 아이디(`pid`)를 `$PPID`이라고 합니다. [1]

이것들을 pidof 명령어와 비교해 보세요.

`$PS1`

명령어줄에서 볼 수 있는 메인 프롬프트.

`$PS2`

2차 프롬프트로, 추가적인 입력이 필요할 때 “>” 로 표시됩니다.

`$PS3`

3차 프롬프트로 select 루프문에서 표시됩니다(예 10-27 참고).

`$PS4`

4차 프롬프트로, 스크립트에 -x 옵션이 걸려서 실행될 때 스크립트의 매 줄마다 “+”로 표시됩니다.

`$PWD`

작업 디렉토리(현재 있는 디렉토리)

내장 명령인 pwd와 비슷합니다.

```
#!/bin/bash

E_WRONG_DIRECTORY=73

clear # 화면 정리.

TargetDirectory=/home/bozo/projects/GreatAmericanNovel

cd $TargetDirectory
echo "$TargetDirectory 디렉토리의 오래된 파일을 지웁니다."

if [ "$PWD" != "$TargetDirectory" ]
then    # 실수로 다른 디렉토리를 지우지 않게 합니다.
  echo "지울 디렉토리가 아닙니다!"
  echo "$TargetDirectory 디렉토리가 아니라 $PWD 디렉토리입니다,"
  echo "취소합니다!"
  exit $E_WRONG_DIRECTORY
fi

rm -rf *
rm .[A-Za-z0-9]*    # 도트 파일 삭제.
# 이름이 여러개의 점으로 시작하는 파일도 지우려면 이렇게 하세
요.   rm -f .[^.]* ..?*
# (shopt -s dotglob; rm -f *)   라고 해도 됩니다.
# 지적해 줘서 고마워요. S.C.

# 파일이름에는 "/"를 제외하고 0 - 255 사이의 모든 문자가 들어갈 수 있습니다.
# 이상한 문자로 시작하는 파일을 지우는 것은 연습문제로 남겨 놓습니다.
```

```
# 필요하다면 다른 작업을 하세요.

echo
echo "끝."
echo "$TargetDirectory 디렉토리의 오래된 파일을 모두 삭제했습니다."
echo


exit 0
```

`$REPLY`

read에 변수가 안 주어졌을 때 저장되는 기본값이고, select 메뉴에서는 변수의 값이 아니라 선택한 숫자가 저장됩니다.

```
#!/bin/bash

echo
echo -n "제일 좋아하는 야채가 뭐에요? "
read

echo "제일 좋아하는 야채가 $REPLY 군요."
# REPLY 는 가장 최근의 "read"가 변수 없이 주어졌을 때 그 값을 담고 있습니다.

echo
echo -n "제일 좋아하는 과일은요? "
read fruit
echo "당신이 제일 좋아하는 과일은 $fruit 지만,"
echo "\$REPLY 의 값은 여전히 $REPLY 네요."
# $fruit 변수가 "read"의 값을 가져가 버렸기 때문에 $REPLY 는 여전히
# 앞에서 설정된 값을 갖고 있습니다.

echo

exit 0
```

`$SECONDS`

스크립트가 얼마나 돌았는지를 나타내는 초 단위 시간.

```
#!/bin/bash

ENDLESS_LOOP=1
INTERVAL=1

echo
echo "스크립트를 끝내려면 Control-C 를 누르세요."
echo
```

```
while [ $ENDLESS_LOOP ]
do
  if [ "$SECONDS" -eq 1 ]
  then
    units=second
  else
    units=seconds
  fi

  echo "이 스크립트는 $SECONDS $units 동안 돌고 있습니다."
  sleep $INTERVAL
done


exit 0
```

`$SHELLOPTS`

현재 켜 있는 쉘 옵션들의 목록으로서 읽기 전용 변수

`$SHLVL`

쉘 레벨로 Bash 가 얼마나 깊이 중첩되어 있는지를 나타냄. 만약에 명령어줄에서 $SHLVL 이 1 이었다면 스크립트에서는 2 가 됩니다.

`$TMOUT`

$TMOUT 환경 변수를 0 이 아닌 값 **time**으로 설정해 놓으면 그 시간이 지난 다음에는 로그아웃이 됩니다.

> 참고**:** 불행하게도 이 변수는 콘솔상의 쉘 프롬프트나 엑스텀(한텀)에서만 동작합니다. 예를 들어, read 문에서 *$TMOUT*을 같이 쓰고 싶겠지만 실제로는 제대로 동작하지 않습니다.(**ksh** 버전에서는 **read**의 타임아웃이 제대로 동작한다고 보고된 바 있습니다).

타임 아웃을 쉘 스크립트에서 구현할 수는 있으나 그렇게 효과적이지 않습니다. 한가지 가능한 방법은, 타임 아웃이 났을 때 타이밍 루프가 스크립트에게 시그널을 보내도록 세팅해야 하고 타이밍 루프가 발생시킨 (예 30-4 참고) 인터럽트를 처리할 시그널 처리 루틴도 만들어야 합니다, 휴~~.

예 **9-2.** 타임 아웃 처리 입력

```
#!/bin/bash
# timed-input.sh

# TMOUT=3             스크립트에서는 쓸모가 없습니다.

TIMELIMIT=3  # 여기서는 3초, 다른 값으로 세트할 수 있습니다.
```

```
PrintAnswer()
{
  if [ "$answer" = TIMEOUT ]
  then
    echo $answer
  else       # 두 번의 인스턴스를 구분하기 위해서.
    echo "제일 좋아하는 야채는 $answer 이군요."
    kill $!  # 백그라운드에서 도는 TimerOn 함수가 더 이상 필요없기 때문
에 kill 시킴.
             # $! 는 백그라운드에서 돌고 있는 가장 최근 작업의 PID 입니다.
  fi

}



TimerOn()
{
  sleep $TIMELIMIT && kill -s 14 $$ &
  # 3초를 기다리고 알람 시그널(sigalarm)을 스크립트에 보냄.
}

Int14Vector()
{
  answer="TIMEOUT"
  PrintAnswer
  exit 14
}

trap Int14Vector 14   # 타이머 인터럽트(14)는 우리 의도대로 타임아웃을 처리함.

echo "제일 좋아하는 야채가 뭐죠? "
TimerOn
read answer
PrintAnswer


# 이는 분명히 타임아웃 입력에 대한 미봉책입니다만,
# Bash 로 할 수 있는 최선을 다한 것입니다.
# (독자들에게 도전: 더 나은 해결책을 제시해 보세요.)

# 더 우아한 다른 것이 필요하다면 C 나 C++ 의 'alarm'이나 'setitimer' 같은
# 적당한 라이브러리 함수를 써서 구현하기 바랍니다.

exit 0
```

stty를 쓴 다른 방법.

예 9-3. 타임 아웃 처리 입력, 한 번 더

```
#!/bin/bash
# timeout.sh

# Stephane Chazelas 작성.
# 이 문서의 저자가 수정.

INTERVAL=5                # 타임아웃 인터벌

timedout_read() {
  timeout=$1
  varname=$2
  old_tty_settings=`stty -g`
  stty -icanon min 0 time ${timeout}0
  eval read $varname      # 아니면 그냥    read $varname
  stty "$old_tty_settings"
  # "stty" 맨 페이지 참조.
}

echo; echo -n "이름이 뭐죠? 빨리 대답해요! "
timedout_read $INTERVAL your_name

# 모든 터미널 타입에서 동작하지 않을 수도 있습니다.
# 최대 타임아웃은 어떤 터미널이냐에 달려있습니다.
# (보통은 25.5 초).

echo

if [ ! -z "$your_name" ]  # 타임아웃 전에 입력이 있다면...
then
  echo "아하, 이름이 $your_name 군요."
else
  echo "타임아웃."
fi

echo

# 이 스크립트는 "timed-input.sh" 스크립트와 약간 다르게 동작하는데,
# 키가 눌릴 때마다 타임아웃 카운터가 리셋됩니다.

exit 0
```

`$UID`

사용자 아이디 값

`/etc/passwd`에 저장되어 있는 현재 사용자의 사용자 식별 숫자

비록 su에 의해서 임시로 다른 사용자로 인식되더라도 현재 사용자의 실제 아이디를 나타냅니다. `$UID`는 읽기만 되는 변수로 명령어 줄이나 스크립트에서 변경할 수 없습니다. 그리고, id 내장 명령과 짝을 이룹니다.

예 9-4. 내가 루트인가?

```
#!/bin/bash
# am-i-root.sh:   내가 루트야 아니야?

ROOT_UID=0   # 루트 $UID는 0.

if [ "$UID" -eq "$ROOT_UID" ]  # Will the real "root" please stand up?
then
  echo "루트네요."
else
  echo "그냥 보통 사용자에요.(그래도 당신 어머니는 있는 그대로의 당신을 사랑하신
답니다)."
fi

exit 0


# ============================================================= #
# 스크립트가 이미 종료됐기 때문에 밑의 코드는 실행되지 않습니다.

# 루트인지 알아내는 다른 방법:

ROOTUSER_NAME=root

username=`id -nu`
if [ "$username" = "$ROOTUSER_NAME" ]
then
  echo "루티 투트 투트(Rooty, toot, toot), 당신은 루트 사용자군요."
else
  echo "그냥 보통 사람이군요."
fi

exit 0
```

예 2-2 참고.

> **참고:** `$ENV`, `$LOGNAME`, `$MAIL`, `$TERM`, `$USER`, `$USERNAME`은 bash 내장 명령이 아닙니다. 하지만 종종 bash 시스템 구동 파일에서 환경 변수로 설정이 됩니다. 사용자의 로긴 쉘을 나타내는 `$SHELL`은 `/etc/passwd`를 참고해 설정되거나 "init" 스크립트에서 설정되고, 역시 bash 내장명령어가 아닙니다.
>
> ```
> tcsh% echo $LOGNAME
> bozo
> tcsh% echo $SHELL
> ```

```
/bin/tcsh
tcsh% echo $TERM
rxvt

bash$ echo $LOGNAME
bozo
bash$ echo $SHELL
/bin/tcsh
bash$ echo $TERM
rxvt
```

위치 매개변수(Positional Parameters)

`$0`, `$1`, `$2`, etc.

위치 매개변수로서, 명령어줄에서 스크립트로 넘겨지거나 함수로 넘겨지거나 set 명령어로 강제로 설정됨(예 5-5 과 예 11-10 참고).

`$#`

명령어줄 인자[2] 의 갯수나 위치 매개변수들(예 34-2 참고)

`$*`

한 낱말로 표시되는 위치 매개변수들 모두

`$@`

$*과 똑같지만 각 매개변수는 쿼우트된 문자열로 취급됩니다. 즉, 해석되거나 확장없이 있는 그대로 넘겨집니다. 그 결과로 각 인자는 각각이 서로 다른 낱말로 구분돼서 표시됩니다.

예 **9-5. arglist: $*** 과 **$@** 로 인자를 나열하기

```
#!/bin/bash
# 이 스크립트를 부를 때 "one two three" 같은 인자를 줘서 부르세요.

E_BADARGS=65

if [ ! -n "$1" ]
then
  echo "사용법: `basename $0` argument1 argument2 etc."
  exit $E_BADARGS
fi

echo

index=1
```

```
echo "\"\$*\" 로 인자를 나열하기:"
for arg in "$*"  # "$*" 를 쿼우트하지 않으면 제대로 동작하지 않습니다.
do
  echo "Arg #$index = $arg"
  let "index+=1"
done             # $* 는 모든 인자를 하나의 낱말로 봅니다.
echo "전체 인자 목록은 하나의 낱말로 나타납니다."

echo

index=1

echo "\"\$@\" 로 인자를 나열하기:"
for arg in "$@"
do
  echo "Arg #$index = $arg"
  let "index+=1"
done             # $@ 는 인자들을 분리된 낱말로 봅니다.
echo "전체 인자 목록은 분리된 낱말로 나타납니다."

echo

exit 0
```

특수 매개변수인 `$@`에는 쉘 스크립트로 들어오는 입력을 필터링하는 툴로서 쓰입니다. **cat "$@"**은 자신의 매개변수를 표준입력이나 파일에서 받아 들일 수 있습니다. 예 12-17을 참고하세요.

> **경고**
>
> `$*`과 `$@`은 가끔 $IFS의 설정값에 따라 일관성이 없고 이상한 동작을 합니다.

예 9-6. 일관성 없는 **`$*`**과 **`$@`**의 동작

```
#!/bin/bash

# 쿼우트 여부에 따라 이상하게 동작하는
# Bash 내부 변수 "$*"와 "$@".
# 낱말 조각남과 라인피드가 일관성 없이 처리됩니다.

# 이 예제 스크립트는 Stephane Chazelas 가 제공하고,
# 본 문서의 저자가 약간 수정했습니다.


set -- "첫번째 인자" "두번째" "세번째:인자" "" "네번째: :인자"
# 스크립트의 인자를 $1, $2 등으로 세팅.
```

```
echo

echo 'IFS 는 그대로, "$*"'
c=0
for i in "$*"                # 쿼우트
do echo "$((c+=1)): [$i]"    # 매 인스턴스마다 똑같음.
                             # 인자 에코.
done
echo ---

echo 'IFS 는 그대로, $*'
c=0
for i in $*                  # 쿼우트 안 함
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS 는 그대로, "$@"'
c=0
for i in "$@"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS 는 그대로, $@'
c=0
for i in $@
do echo "$((c+=1)): [$i]"
done
echo ---

IFS=:
echo 'IFS=":", "$*"'
c=0
for i in "$*"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", $*'
c=0
for i in $*
do echo "$((c+=1)): [$i]"
done
echo ---

var=$*
echo 'IFS=":", "$var" (var=$*)'
```

```
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", $var (var=$*)'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done
echo ---

var="$*"
echo 'IFS=":", $var (var="$*")'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", "$var" (var="$*")'
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", "$@"'
c=0
for i in "$@"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", $@'
c=0
for i in $@
do echo "$((c+=1)): [$i]"
done
echo ---

var=$@
echo 'IFS=":", $var (var=$@)'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done
echo ---
```

```
echo 'IFS=":", "$var" (var=$@)'
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

var="$@"
echo 'IFS=":", "$var" (var="$@")'
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", $var (var="$@")'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done

echo

# 이 스크립트를 ksh 이나 zsh -y 로 실행해 보세요.

exit 0
```

참고: **$@**과 **$***는 큰따옴표로 쿼우트 됐을 때만 달라집니다.

예 9-7. `$IFS` 가 비어 있을 때 `$*`와 `$@`

```
#!/bin/bash

# $IFS 가 빈 상태로 세트됐다면,
# "$*" 와 "$@" 는 위치 매개변수를 우리가 생각하는대로 에코하지 않습니다.

mecho ()       # 위치 매개변수 에코.
{
echo "$1,$2,$3";
}


IFS=""         # 빈 상태로 세트.
set a b c      # 위치 매개변수.

mecho "$*"     # abc,,
```

```
mecho $*       # a,b,c

mecho $@       # a,b,c
mecho "$@"     # a,b,c

# $IFS 가 비어 있을 경우에 $* 와 $@ 의 동작은
# Bash 나 sh 의 버전에 따라 달라지기 때문에
# 스크립트에서 이 "기능"에 쓰는 것은 권장하지 않습니다.


# Thanks, S.C.

exit 0
```

다른 특수 매개변수

`$-`

스크립트로 넘겨진 플래그들

> **경고**
>
> 원래 **ksh**에서 쓰던 것을 bash에서 채택한 것인데 불행히도 잘 동작하지 않는 것 같습니다. 스크립트 자신이 대화형인지 아닌지 스스로 확인하려고 하는 경우에나 쓸만합니다.

`$!`

백그라운드로 돌고 있는 가장 최근 작업의 PID (process id)

`$_`

바로 이전에 실행된 명령어의 제일 마지막 인자로 설정되는 특수 변수.

예 9-8. 밑줄 변수(underscore variable)

```
#!/bin/bash

echo $_          # /bin/bash
# 스크립트를 돌리기 위해 실행된 /bin/bash.

du >/dev/null    # 명령어 출력이 없음.
echo $_          # du

ls -al           # 명령어 출력이 없음.
echo $_          # -al  (마지막 인자)

:
```

```
echo $_          # :
```

`$?`

명령어나 함수, 스크립트 자신의 종료 상태(예 23-3 참고).

`$$`

스크립트 자신의 프로세스 아이디로 보통 임시 파일 이름을 만들 때 사용합니다(예 A-8와 예 30-5, 예 12-23 참고).

9.2. 문자열 조작

Bash 는 놀랍도록 많은 문자열 조작 연산을 제공합니다. 하지만 불행하게도 이 도구들은 하나로 통합되어 있지 않습니다. 몇 개는 매개변수 치환의 서브셋이고 다른 것은 유닉스의 expr 명령어의 기능에 해당합니다. 이렇기 때문에 이런 혼동스러움에 대한 언급도 없이 명령어 문법에 일관성이 없고 기능이 중복되어 있습니다.

문자열 길이

${#string}

expr length $string

expr "$string" : '.*'

```
stringZ=abcABC123ABCabc

echo ${#stringZ}                 # 15
echo `expr length $stringZ`      # 15
echo `expr "$stringZ" : '.*'`    # 15
```

문자열 시작에서부터 매칭되는 문자열조각(**substring**)의 길이

expr match "$string" '$substring'

$substring 은 정규 표현식입니다.

expr "$string" : '$substring'

$substring 은 정규 표현식입니다.

```
stringZ=abcABC123ABCabc
#       |------|

echo `expr match "$stringZ" 'abc[A-Z]*.2'`   # 8
echo `expr "$stringZ" : 'abc[A-Z]*.2'`       # 8
```

인덱스

expr index $string $substring

$string 에서 일치하는 $substring 의 첫 문자의 위치.

```
stringZ=abcABC123ABCabc
echo `expr index "$stringZ" C12`             # 6
                                             # C 의 위치.

echo `expr index "$stringZ" 1c`              # 3
# 'c' (3번째 위치에 있는) 가 '1' 보다 먼저 일치됨.
```

C 의 **strchr()**와 거의 비슷합니다.

문자열조각 추출(Substring Extraction)

${string:position}

$string 의 *$position* 부터의 문자열조각을 추출.

`string` 매개변수가 "*" 이거나 "@" 라면 `position`에서 시작하는 위치 매개변수[3] 를 추출해 냅니다.

${string:position:length}

*$string*의 *$position*부터 *$length*만큼의 문자를 추출해 냅니다.

```
stringZ=abcABC123ABCabc
#       0123456789.....
#       0 부터 시작하는 인덱싱.

echo ${stringZ:0}                            # abcABC123ABCabc
echo ${stringZ:1}                            # bcABC123ABCabc
echo ${stringZ:7}                            # 23ABCabc

echo ${stringZ:7:3}                          # 23A
                                             # 3글자짜리 문자열조각.
```

`string` 매개변수가 "*" 나 "@" 라면 위치 `position`에서 시작하는 매개변수의 최대 `length`를 추출해 냅니다.

```
echo ${*:2}          # 두번째 이후의 위치 매개변수를 에코.
echo ${@:2}          # 위와 같음.

echo ${*:2:3}        # 2,3,4번(3개) 위치 매개변수를 에코.
```

expr substr $string $position $length

*`$string`*의 *`$position`*부터 *`$length`*만큼의 문자를 추출해 냅니다.

```
stringZ=abcABC123ABCabc
#       123456789......
#       1 부터 시작하는 인덱싱.

echo `expr substr $stringZ 1 2`              # ab
echo `expr substr $stringZ 4 3`              # ABC
```

expr match "$string" ’\($substring\)’

*`$string`*의 처음에서부터 정규 표현식인 *`$substring`*을 추출해 냅니다.

expr "$string" : ’\($substring\)’

*`$string`*의 처음에서부터 정규 표현식인 *`$substring`*을 추출해 냅니다.

```
stringZ=abcABC123ABCabc

echo `expr match "$stringZ" '\(.[b-c]*[A-Z]..[0-9]\)'`   # abcABC1
echo `expr "$stringZ" : '\(.[b-c]*[A-Z]..[0-9]\)'`       # abcABC1
# 위의 두 가지 형태는 동일합니다.
```

문자열조각 삭제(Substring Removal)

${string#substring}

*`$string`*의 앞 부분에서부터 가장 짧게 일치하는 *`$substring`*을 삭제.

${string##substring}

*`$string`*의 앞 부분에서부터 가장 길게 일치하는 *`$substring`*을 삭제.

```
stringZ=abcABC123ABCabc
#       |----|
#       |----------|

echo ${stringZ#a*C}      # 123ABCabc
# 'a'와 'C' 사이에서 가장 짧게 일치되는 부분을 삭제.

echo ${stringZ##a*C}     # abc
# 'a'와 'C' 사이에서 가장 길게 일치되는 부분을 삭제.
```

${string%substring}

*`$string`*의 뒷 부분에서부터 가장 짧게 일치하는 *`$substring`*을 삭제.

${string%%substring}

*`$string`*의 뒷 부분에서부터 가장 길게 일치하는 *`$substring`*을 삭제.

```
stringZ=abcABC123ABCabc
#                    ||
#        |------------|

echo ${stringZ%b*c}      # abcABC123ABCa
# $stringZ의 뒷 부분부터 계산해서 'b'와 'c' 사이에서 가장 짧게 일치하는 부분
을 삭제.

echo ${stringZ%%b*c}     # a
# $stringZ의 뒷 부분부터 계산해서 'b'와 'c' 사이에서 가장 길게 일치하는 부분
을 삭제.
```

예 9-9. 그래픽 파일을 다른 포맷 확장자로 이름을 바꾸면서 변환

```
#!/bin/bash
#  cvt.sh:
#  특정 디렉토리의 모든 MacPaint 이미지 파일을 "pbm" 포맷으로 변환.

#  Brian Henderson(bryanh@giraffe-data.com)이 관리하고 있는 "netpbm" 패키
지의
#+ "macptopbm" 을 사용함.
#  Netpbm 은 거의 대부분의 리눅스 배포판에 포함되어 있습니다.

OPERATION=macptopbm
SUFFIX=pbm          # 새 파일이름 확장자.

if [ -n "$1" ]
then
  directory=$1      # 디렉토리 이름이 인자로 주어질 경우...
else
  directory=$PWD    # 아니면 현재 디렉토리에 대해서.
fi

# 대상 디렉토리의 모든 파일을 ".mac" 확장자의 MacPaint 이미지 파일이라고 가정.

for file in $directory/*    # 파일이름 globbing.
do
  filename=${file%.*c}      #  파일이름에서 ".mac" 확장자를 떼어냄
                            #+ ('.*c' 는 '.' 과 'c'를 포함해서 둘 사이의
                            #+ 모든 것과 일치함).
  $OPERATION $file > $filename.$SUFFIX
```

```
                              # 변환된 파일을 새 파일이름으로 재지향.
    rm -f $file               # 변환후 원래 파일 삭제.
    echo "$filename.$SUFFIX"  # 결과를 표준출력으로 로깅.
  done

  exit 0
```

문자열 조각 대치(Substring Replacement)

${string/substring/replacement}

처음 일치하는 *$substring*을 *$replacement*로 대치.

${string//substring/replacement}

일치하는 모든 *$substring*을 *$replacement*로 대치.

```
stringZ=abcABC123ABCabc

echo ${stringZ/abc/xyz}           # xyzABC123ABCabc
                                  # 처음 일치하는 'abc'를 'xyz'로 대치.

echo ${stringZ//abc/xyz}          # xyzABC123ABCxyz
                                  # 일치하는 모든 'abc'를 'xyz'로 대치.
```

${string/#substring/replacement}

*$substring*이 *$string*의 맨 앞에서 일치하면 *$replacement*로 대치.

${string/%substring/replacement}

*$substring*이 *$string*의 맨 뒤에서 일치하면 *$replacement*로 대치.

```
stringZ=abcABC123ABCabc

echo ${stringZ/#abc/XYZ}          # XYZABC123ABCabc
                                  # 맨 앞에서 일치하는 'abc'를 'xyz'로 대
치.

echo ${stringZ/%abc/XYZ}          # abcABC123ABCXYZ
                                  # 맨 뒤에서 일치하는 'abc'를 'xyz'로 대
치.
```

스크립트에서 문자열 조작에 대한 더 자세한 사항은 9.3절와 expr 명령어에서 문자열 조작과 관련된 부분을 참고하세요. 스크립트 예제는 다음을 참고하세요.

1. 예 12-6
2. 예 9-11

3. 예 9-12
4. 예 9-13
5. 예 9-15

9.3. 매개변수 치환(Parameter Substitution)

변수를 조작하거나 확장시키기

`${parameter}`

변수인 *parameter*의 값이란 뜻으로서, *$parameter*라고 한 것과 같습니다. 어떤 문맥에서는 *${parameter}*라고 확실히 써 줘야 동작하는 수도 있습니다.

문자열 변수들을 연결할 때 쓰일 수 있습니다.

```
your_id=${USER}-on-${HOSTNAME}
echo "$your_id"
#
echo "Old \$PATH = $PATH"
PATH=${PATH}:/opt/bin  # 스크립트가 도는 동안 $PATH 에 /opt/bin 을 추가.
echo "New \$PATH = $PATH"
```

`${parameter-default}`

매개변수가 세트되지 않았다면 default를 사용합니다.

```
echo ${username-`whoami`}
# $username이 여전히 세트되어 있지 않다면 `whoami`의 결과를 에코.
```

> 참고: 이렇게 하면 *${parameter:-default}*라고 하는 것과 거의 비슷하지만 :이 있을 때는 매개변수가 선언만 되어 값이 널일 경우에도 기본값을 적용시킵니다.

```
#!/bin/bash

username0=
# username0 는 선언만 되고 널로 세트됐습니다.
echo "username0 = ${username0-`whoami`}"
# 에코 되지 않습니다.

echo "username1 = ${username1-`whoami`}"
# username1 는 선언되지 않았습니다.
# 에코 됩니다.
```

```
username2=
# username2 는 선언만 되고 널고 세트됐습니다.
echo "username2 = ${username2:-`whoami`}"
# 조건 테스트시 - 가 아니고 :- 를 썼기 때문에 에코 됩니다.

exit 0
```

${parameter=default}
${parameter:=default}

매개변수가 세트 되어 있지 않다면 기본값으로 세트.

두 형태는 거의 비슷하지만 :이 있을 때는 위의 경우처럼 **$parameter**가 선언만 되고 값이 널일 경우에도 기본값으로 세트 시킨다는 차이점이 있습니다[4]

```
echo ${username=`whoami`}
# "username" 변수를 `whoami`의 결과로 세트.
```

${parameter+alt_value}
${parameter:+alt_value}

매개변수가 세트되어 있다면 **alt_value**를 쓰고 아니라면 널 스트링을 씁니다.

이 두 형태는 거의 비슷하지만 **parameter**가 선언되고 널일 경우에 :이 있고 없고의 차이가 나타납니다. 아래를 보세요.

```
echo "###### \${parameter+alt_value} ########"
echo

a=${param1+xyz}
echo "a = $a"      # a =

param2=
a=${param2+xyz}
echo "a = $a"      # a = xyz

param3=123
a=${param3+xyz}
echo "a = $a"      # a = xyz

echo
echo "###### \${parameter:+alt_value} ########"
echo

a=${param4:+xyz}
echo "a = $a"      # a =

param5=
```

```
    a=${param5:+xyz}
    echo "a = $a"      # a =
    # a=${param5+xyz}    와 결과가 다르죠?

    param6=123
    a=${param6+xyz}
    echo "a = $a"      # a = xyz
```

`${parameter?err_msg}`
`${parameter:?err_msg}`

매개변수가 세트되어 있으면 그 값을 쓰고 아니라면 err_msg를 출력.

두 형태도 역시 비슷하지만 :은 **parameter**가 선언만 되고 널일 경우에만 차이점이 나타납니다.

예 **9-10.** 매개변수 치환과 **:** 쓰기

```
#!/bin/bash

#  몇개의 시스템 환경 변수 확인
#  예를 들어, 콘솔 사용자의 이름을 나타내는 $USER 가 세트 돼 있지 않다면,
#+ 시스템은 여러분을 인식하지 못합니다.

: ${HOSTNAME?} ${USER?} ${HOME?} ${MAIL?}
  echo
  echo "시스템 이름은 $HOSTNAME 입니다."
  echo "여러분의 이름은 $USER 입니다."
  echo "여러분의 홈디렉토리는 $HOME 입니다."
  echo "여러분의 메일 INBOX는 $MAIL 에 있습니다."
  echo
  echo "이 메세지를 읽고 있다면,"
  echo "중요한 환경 변수가 세트 되어 있다는 뜻입니다."
  echo
  echo

# ------------------------------------------------------

#  ${variablename?} 도 스크립트에서 변수가 세트 되어 있는지를
#+ 확인할 수 있습니다.

ThisVariable=Value-of-ThisVariable
#  문자열 변수는 변수 이름에 쓸 수 없는 문자가
#+ 세트될 수도 있으니 주의하세요.
: ${ThisVariable?}
echo "ThisVariable 의 값은 $ThisVariable."
echo
echo
```

```
: ${ZZXy23AB?"ZZXy23AB 는 세트되지 않았습니다."}
#  ZZXy23AB 에 값이 세트되어 있지 않다면,
#+ 에러 메세지를 뿌리고 종료할 것입니다.

# 에러 메세지를 지정할 수 있습니다.
# : ${ZZXy23AB?"ZZXy23AB 는 세트되지 않았습니다."}


# 다음과 똑같은 결과:  dummy_variable=${ZZXy23AB?}
#                      dummy_variable=${ZZXy23AB?"ZXy23AB 는 세트되지 않았습
니다."}
#
#                      echo ${ZZXy23AB?} >/dev/null



echo "위에서 스크립트가 종료됐기 때문에 이 메세지는 안 보입니다."

HERE=0
exit $HERE   # 여기서 종료되지 *않습니다*.
```

매개변수 치환과 확장. 다음 식들은 **expr** 문자열 연산중 **match**에 대해 부족한 부분을 보완해 줍니다 (예 12-6 참고). 주로, 파일 경로명을 파싱할 때 쓰입니다.

변수 길이/문자열조각(substring) 삭제

`${#var}`

문자열 길이 (`$var`의 문자 갯수). 배열의 경우에, **${#array}**라고 하면 배열의 첫번째 요소의 길이를 알려줍니다.

참고**:** 예외:

- **${#*}** 와 **${#@}** 는 위치 매개변수의 갯수를 알려줍니다.
- 배열에 대해 **${#array[*]}** 나 **${#array[@]}** 라고 하면 배열 요소의 갯수를 알려줍니다.

예 **9-11.** 변수의 길이

```
#!/bin/bash
# length.sh

E_NO_ARGS=65

if [ $# -eq 0 ]  # 이 스크립트에서는 명령어줄 인자가 필요합니다.
then
```

```
  echo "하나 이상의 명령어줄 인자가 필요합니다."
  exit $E_NO_ARGS
fi

var01=abcdEFGH28ij

echo "var01 = ${var01}"
echo "var01 의 길이 = ${#var01}"

echo "스크립트로 넘어온 명령어줄 인자 갯수 = ${#@}"
echo "스크립트로 넘어온 명령어줄 인자 갯수 = ${#*}"

exit 0
```

${var#pattern}
${var##pattern}

`$pattern`이 `$var`의 *앞 부분*과 가장 길거나 가장 짧게 일치하는 부분을 삭제.

예 A-6에서 발췌한 사용법 예제:

```
# "days-between.sh" 예제에서 쓰인 함수.
# 주어진 인자의 앞 부분에 들어있는 하나 이상의 0 을 삭제.

strip_leading_zero () # 날짜나 월의 앞 부분에 나오는 0을 삭제하지 않으면
{                     # Bash 가 8진수로 해석하기 때문
에(POSIX.2, sect 2.9.2.1)
  val=${1#0}          # 이렇게 해 주는게 좋습니다.
  return $val
}
```

다른 사용법 예제:

```
echo `basename $PWD`        # 현재 디렉토리의 basename.
echo "${PWD##*/}"           # 현재 디렉토리의 basename.
echo
echo `basename $0`          # 스크립트 이름.
echo $0                     # 스크립트 이름.
echo "${0##*/}"             # 스크립트 이름.
echo
filename=test.data
echo "${filename##*.}"      # 데이타
                            # 전체 파일이름에서 확장자.
```

${var%pattern}
${var%%pattern}

`$pattern`이 `$var`의 *뒷 부분*과 가장 짧거나 가장 길게 일치하는 부분을 삭제.

Bash 버전 2에는 옵션이 더 늘었습니다.

예 **9-12.** 매개변수 치환에서의 패턴 매칭

```
#!/bin/bash
# 매개변수 치환 연산자인 # ## % %% 를 써서 패턴 매칭하기.

var1=abcd12345abc6789
pattern1=a*c  # * (와일드 카드)는 a 와 c 사이의 모든 문자와 일치합니다.

echo
echo "var1 = $var1"           # abcd12345abc6789
echo "var1 = ${var1}"         # abcd12345abc6789   (다른 형태)
echo "${var1} 에 들어 있는 글자수 = ${#var1}"
echo "pattern1 = $pattern1"   # a*c  ('a'와 'c' 사이의 모든 문자)
echo


echo '${var1#$pattern1}  =' "${var1#$pattern1}"    #         d12345abc6789
# 앞에서부터 가장 짧게 일치하는 3 글자를 삭제             abcd12345abc6789
# ^^^^^^^^^^                                              |-|
echo '${var1##$pattern1} =' "${var1##$pattern1}"   #                  6789
# 앞에서부터 가장 길게 일치하는 12 글자를 삭제            abcd12345abc6789
# ^^^^^^^^^^                                              |----------|

echo; echo

pattern2=b*9            # 'b'와 '9' 사이의 모든 문자.
echo "var1 = $var1"     # abcd12345abc6789 를 계속 씁니다.
echo "pattern2 = $pattern2"
echo

echo '${var1%pattern2}  =' "${var1%$pattern2}"     #     abcd12345a
# 뒤에서부터 가장 짧게 일치하는 6 글자를 삭제            abcd12345abc6789
# ^^^^^^^^^^                                                       |----|
echo '${var1%%pattern2} =' "${var1%%$pattern2}"    #     a
# 뒤에서부터 가장 길게 일치하는 12 글자를 삭제           abcd12345abc6789
# ^^^^^^^^^^                                              |-------------|

# 이렇게 외우세요.
#           # 과 ## 은 문자열의 앞쪽에서부터 동작을 하고,
#           % 와 %% 는 뒤쪽에서부터 동작을 합니다.

echo

exit 0
```

예 **9-13.** 파일 확장자 바꾸기**:**

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

```
#                  rfe
#                  ---

# 파일 확장자 바꾸기(Renaming file extensions).
#
#         rfe 원래확장자 새확장자
#
# 예제:
# 현재 디렉토리의 *.gif 를 *.jpg 로 바꾸려면,
#          rfe gif jpg

ARGS=2
E_BADARGS=65

if [ $# -ne $ARGS ]
then
  echo "사용법: `basename $0` 원래확장자 새확장자"
  exit $E_BADARGS
fi

for filename in *.$1
# 첫번째 인자로 끝나는 파일 목록을 전부 탐색.
do
  mv $filename ${filename%$1}$2
  # 첫번째 인자와 일치하는 부분을 떼어내고,
  # 두번째 인자를 덧붙임.
done

exit 0
```

변수 확장/문자열조각 대치

여기서 소개하는 것들은 **ksh**에서 따 온 것입니다.

`${var:pos}`

`var` 변수가 *`pos`* 옵셋부터 시작하도록 확장.

`${var:pos:len}`

변수 *`var`*가 *`pos`*에서 최대 *`len`*만큼의 길이를 갖도록 확장. 이 연산자의 창조적인 사용 예제를 보려면 예 A-9 참고.

`${var/patt/replacement}`

*`var`*에 첫번째로 일치하는 *`patt`*을 *`replacement`*로 대치시킴.

*`replacement`*를 안 적으면 *`patt`*이 지워짐.

`${var//patt/replacement}`

전역 대치(**Global replacement**). *`var`*에서 일치하는 모든 *`patt`*을 *`replacement`*로 대치시킴.

위의 경우와 비슷하게 *`replacement`*를 안 적어주면 모든 *`patt`*이 지워짐.

예 **9-14.** 임의의 문자열을 파싱하기 위해 패턴 매칭 사용하기

```
#!/bin/bash

var1=abcd-1234-defg
echo "var1 = $var1"

t=${var1#*-*}
echo "var1 (첫번째 - 를 포함한 부분까지 잘라냄) = $t"
#  # 이 가장 짧은 문자열과 매치하고 * 가 빈 문자열을 포함한 모든것과
#+ 일치하기 때문에 t=${var1#*-}  도 똑같이 동작합니다.
# (S. C. 가 지적해 주었습니다.)

t=${var1##*-*}
echo "var1에 \"-\" 이 들어있다면 빈 문자열을 리턴함...    var1 = $t"


t=${var1%*-*}
echo "var1 (제일 마지막의 - 부터 끝까지 잘라냄) = $t"

echo

# -------------------------------------------
path_name=/home/bozo/ideas/thoughts.for.today
# -------------------------------------------
echo "path_name = $path_name"
t=${path_name##/*/}
echo "접두어가 잘려진 path_name = $t"
# 이 경우에서는    t=`basename $path_name` 이라고 해도 똑같습니다.
#  t=${path_name%/}; t=${t##*/}   라고 하는 것이 좀 더 일반적인 해법이지만,
#+ 가끔은 실패할 수도 있습니다.
#  $path_name 이 뉴라인으로 끝날 경우에는 `basename $path_name` 은 제대로 동
작하지 않겠지만,
#+ 위 표현식은 제대로 동작할 것입니다.
# (Thanks, S.C.)

t=${path_name%/*.*}
# t=`dirname $path_name`    라고 하는 것과 똑같습니다.
echo "접미어가 잘려진 path_name = $t"
# 이렇게 하면 "../", "/foo////", # "foo/", "/" 같은 경우에는 실패할 것입니
다.
#  basename 은 접미어가 없고 dirname 은 있을 경우에 접미어를 삭제하는 것
#+ 역시 복잡한 문제입니다.
```

```
# (Thanks, S.C.)

echo

t=${path_name:11}
echo "첫번째 11개 문자가 잘려나간 $path_name = $t"
t=${path_name:11:5}
echo "첫번째 11개 문자가 잘려나가고 길이가 5인 $path_name = $t"

echo

t=${path_name/bozo/clown}
echo "\"bozo\" 가 \"clown\" 으로 대치된 $path_name = $t"
t=${path_name/today/}
echo "\"today\" 가 삭제된 $path_name = $t"
t=${path_name//o/O}
echo "모든 소문자 o 를 대문자로 만든 $path_name = $t"
t=${path_name//o/}
echo "모든 소문자 o 를 삭제한 $path_name = $t"

exit 0
```

`${var/#patt/replacement}`

*`var`*의 접두어(prefix)가 *`patt`*과 일치하면 *`patt`*을 *`replacement`*로 치환.

`${var/%patt/replacement}`

*`var`*의 접미어(suffix)가 *`patt`*과 일치하면 *`patt`*을 *`replacement`*로 치환.

예 **9-15.** 문자열의 접두, 접미어에서 일치하는 패턴 찾기

```
#!/bin/bash
# 문자열의 접두어/접미어 에서 패턴 치환 하기.

v0=abc1234zip1234abc    # 원래 변수.
echo "v0 = $v0"         # abc1234zip1234abc
echo

# 문자열의 접두어(시작)에서 일치.
v1=${v0/#abc/ABCDEF}    # abc1234zip1234abc
                        # |-|
echo "v1 = $v1"         # ABCDE1234zip1234abc
                        # |---|

# 문자열의 접미어(끝)에서 일치.
v2=${v0/%abc/ABCDEF}    # abc1234zip123abc
                        #              |-|
echo "v2 = $v2"         # abc1234zip1234ABCDEF
```

```
                         #                |----|

echo

# -------------------------------------------------
#  문자열의 시작과 끝에서 일치가 일어나야지만
#+ 대치가 일어납니다.
# -------------------------------------------------
v3=${v0/#123/000}       # 일치하지만 시작 부분이 아닙니다.
echo "v3 = $v3"         # abc1234zip1234abc
                        # *대치가 안 일어납니다*.
v4=${v0/%123/000}       # 일치하지만 끝 부분이 아닙니다.
echo "v4 = $v4"         # abc1234zip1234abc
                        # *대치가 안 일어납니다*.

exit 0
```

`${!varprefix*}`
`${!varprefix@}`

이미 선언된 변수들 중에 **varprefix**로 시작하는 변수들.

```
xyz23=whatever
xyz24=

a=${!xyz*}      # 선언된 변수중 "xyz"로 시작하는 변수로 확장.
echo "a = $a"   # a = xyz23 xyz24
a=${!xyz@}      # 위와 같음.
echo "a = $a"   # a = xyz23 xyz24

# Bash 2.04 버전에서 이 기능이 추가됨.
```

9.4. 변수 타입 지정: declare 나 typeset

declare나 **typeset** 내장 명령(이 둘은 동의어입니다.) 키워드는 변수의 특성을 제한할 수 있습니다. 이것은 몇몇 프로그래밍 언어에서 볼 수 있는 불완전한 형태의 타입 지정입니다. **declare** 명령어는 bash 버전 2 이후부터 가능합니다. **typeset** 명령어는 ksh 스크립트에서도 가능합니다.

declare/typeset 옵션

-r *읽기 전용*

```
declare -r var1
```

(**`declare -r var1`** 는 **`readonly var1`** 와 똑같이 동작합니다)

C에서 **const** 형한정어(qualifier)와 거의 비슷하고, 이런 변수의 값을 바꾸려고 하면 에러 메세지가 납니다.

-i *정수*

```
declare -i number
# 이 스크립트는 이후 나오는 모든 "number"를 정수로 취급할 것입니다.

number=3
echo "number = $number"     # number = 3

number=three
echo "number = $number"     # number = 0
# "three"를 정수로 계산하려는 시도.
```

expr 이나 let 이 없이 정수로 선언된 변수에 대한 연산을 허용하는 몇몇 산술 연산이 있습니다.

-a *배열*

```
declare -a indices
```

`indices` 변수는 배열로 취급됩니다.

-f *함수*

```
declare -f
```

스크립트에서 인자 없이 **`declare -f`** 가 나오는 줄에서는 스크립트안에서 정의된 모든 함수들의 목록을 보여줍니다.

```
declare -f function_name
```

스크립트 안에서 **`declare -f function_name`** 라고 하면 그냥 그 함수 이름을 보여줍니다.

-x export

```
declare -x var3
```

이 선언은 스크립트 외부 환경에서도 이 변수를 쓸 수 있게 해 줍니다.

var=$value

```
declare -x var3=373
```

declare 명령어는 한 문장 안에서 선언과 동시에 그 값을 할당할 수 있게 해 줍니다.

예 **9-16. declare**를 써서 변수 타입 지정하기

```
#!/bin/bash

func1 ()
{
echo 여기는 함수에요.
}

declare -f        # 위 함수를 나열.

echo

declare -i var1   # var1 은 정수.
var1=2367
echo "var1 은 $var1 로 선언됐습니다."
var1=var1+1       # 정수 선언은 'let' 이 필요없습니다.
echo "1이 증가된 var1 은 $var1 입니다."
# Attempt to change variable declared as integer
echo "var1을 부동형 값인 2367.1 로 바꾸려는 시도."
var1=2367.1       # 에러 메세지를 내고 값은 변하지 않습니다.
echo "var1 은 여전히 $var1 입니다."

echo

declare -r var2=13.36         # 'declare' 는 변수 타입을 설정하고 동시에
                              #+ 그 값을 할당할 수 있게 해 줍니다.
echo "var2 는 $var2 로 선언됐습니다." # 읽기 전용 변수 값을 변경하려는 시도.
var2=13.37                    # 에러 메세지를 내고 스크립트 종료.

echo "var2 는 여전히 $var2 입니다."    # 여기는 실행되지 않을 것입니다.

exit 0                        # 여기서 종료되지 않습니다.
```

9.5. 변수 간접 참조

어떤 변수값이 다음에 나올 변수의 이름이라고 가정해 봅시다. 그렇다면 그 어떤 변수로 다음에 나올 변수의 값을 알아낼 수 있을까요? 예를 들어, *`a=letter_of_alphabet`* 이고

`letter_of_alphabet=z` 라고 할 때 *a*를 참조하면 *z*가 나올까요? 이는 우리 생각처럼 제대로 동작을 하고 이를 간접 참조라고 부릅니다. `eval var1=\$$var2` 처럼 약간은 이상하게 생긴 형태로 써먹을 수 있습니다.

예 9-17. 간접 참조

```
#!/bin/bash
# 변수 간접 참조.

a=letter_of_alphabet
letter_of_alphabet=z

echo

# 직접 참조.
echo "a = $a"

# 간접 참조.
eval a=\$$a
echo "이제 a = $a"

echo


# 이제 2차 참조(second order reference)를 바꿔보도록 하죠.

t=table_cell_3
table_cell_3=24
echo "\"table_cell_3\" = $table_cell_3"
echo -n "역참조(dereferenced)된 \"t\" = "; eval echo \$$t
# 이 간단한 예제에서는,
#   eval t=\$$t; echo "\"t\" = $t"
# 라고 해도 됩는데 왜 그럴까요?

echo

t=table_cell_3
NEW_VAL=387
table_cell_3=$NEW_VAL
echo "\"table_cell_3\" 의 값을 $NEW_VAL 로 바꿉니다."
echo "\"table_cell_3\" 은 이제 $table_cell_3 이고,"
echo -n "역참조된 \"t\" 는 "; eval echo \$$t
# "eval" 은 "echo"와 "\$$t" 두 개의 인자를 받아서 $table_cell_3 과 똑같이 세트해
줍니다.
echo

# (S.C. 가 위 동작을 자세히 설명해 주었습니다.)
```

```
# "Bash, 버전 2" 장에서 설명할 ${!t} 표기법도 쓸 수 있습니다.
# "ex78.sh" 예제를 참고하세요.

exit 0
```

예 9-18. `awk`에게 간접 참조를 넘기기

```
#!/bin/bash

# "column totaler" 스크립트의 다른 버전으로
# 간접 참조를 사용해서,
# 대상 파일에서 지정된 컬럼을 모두 더해 줍니다.

ARGS=2
E_WRONGARGS=65

if [ $# -ne "$ARGS" ] # 명령어줄 인자 갯수 체크.
then
   echo "사용법: `basename $0` filename column-number"
   exit $E_WRONGARGS
fi

filename=$1
column_number=$2

#===== 여기까지는 원래 스크립트와 똑같네요. =====#


# 여러줄짜리 awk 스크립트는 awk ' ..... '  라고 하면 됩니다.


# awk 스크립트 시작.
# ------------------------------------------------
awk "

{ total += \$${column_number} # 간접 참조
}
END {
     print total
     }

     " "$filename"
# ------------------------------------------------
# awk 스크립트 끝.

# 변수 간접 참조는 쉘 스크립트 안에서 awk 스크립트를 쓸 때
# 쉘 변수 참조시 까다로운 여러가지를 피하게 해 줍니다.
# Thanks, Stephane Chazelas.
```

```
exit 0
```

> **경고**
>
> 이런 식의 간접 참조는 약간 까다로운 부분이 있는데, 만약에 두번째 변수의 값이 바뀌면 첫번째 변수는 참조하던 것을 제대로 풀어줘야 합니다(위의 예제처럼). 다행스럽게도, bash 버전 2(예 35-2 참고)에서 새롭게 나타난 *`${!variable}`* 표기법으로 간접 참조를 좀 더 직관적으로 할 수 있게 되었습니다.

9.6. $RANDOM: 랜덤한 정수 만들기

참고**:** $RANDOM 은 bash 내부 함수(상수가 아님)로 0에서 32767사이의 의사난수(pseudorandom)를 리턴합니다. $RANDOM 은 암호화 키를 발생 시키는데 쓸 수 *없습니다.*

예 **9-19.** 랜덤한 숫자 만들기

```
#!/bin/bash

# $RANDOM 은 불릴 때마다 다른 무작위 정수 값을 리턴합니다.
# 명칭상의 범위(nominal range): 0 - 32767(16 비트 양의 정수).

MAXCOUNT=10
count=1

echo
echo "$MAXCOUNT 개의 랜덤한 숫자:"
echo "-----------------"
while [ "$count" -le $MAXCOUNT ]      # 10 ($MAXCOUNT) 개의 랜덤 정수 발생.
do
  number=$RANDOM
  echo $number
  let "count += 1"  # 카운터 증가.
done
echo "-----------------"

# 어떤 범위의 랜덤 값이 필요하다면 '나머지(modulo)' 연산자를 쓰면,
# 어떤 수를 나눈 나머지 값을 리턴해 줍니다.

RANGE=500

echo
```

```
number=$RANDOM
let "number %= $RANGE"
echo "$RANGE 보다 작은 랜덤한 숫자  ---  $number"

echo

# 어떤 값보다 큰 랜덤한 정수가 필요하다면
# 그 값보다 작은 수는 무시하는 테스트 문을 걸면 됩니다.

FLOOR=200

number=0   # 초기화
while [ "$number" -le $FLOOR ]
do
  number=$RANDOM
done
echo "$FLOOR 보다 큰 랜덤한 숫자  ---  $number"
echo


# 상한값과 하한값 사이의 수가 필요하다면 위의 두 테크닉을 같이 쓰면 됩니다.
number=0   # 초기화
while [ "$number" -le $FLOOR ]
do
  number=$RANDOM
  let "number %= $RANGE"  # $number 가 $RANGE 안에 들어오게.
done
echo "$FLOOR 와 $RANGE 사이의 랜덤한 숫자  ---  $number"
echo


# "참"이나 "거짓"중에 하나를 고르도록 할 수도 있습니다.
BINARY=2
number=$RANDOM
T=1

let "number %= $BINARY"
# let "number >>= 14"    더 좋은 분포의 랜덤값을 줍니다.
# (마지막 두 비트를 제외하고 모두 오른쪽으로 쉬프트 시킴).
if [ "$number" -eq $T ]
then
  echo "TRUE"
else
  echo "FALSE"
fi

echo
```

```
# 주사위 던지기를 흉내내 볼까요?
SPOTS=7   # 7 의 나머지(modulo)는 0 - 6.
DICE=2
ZERO=0
die1=0
die2=0

# 정확한 확률을 위해서 두 개의 주사위를 따로 던집시다.

  while [ "$die1" -eq $ZERO ]     # 주사위에 0은 없죠.
  do
    let "die1 = $RANDOM % $SPOTS" # 첫번째 주사위를 굴리고.
  done

  while [ "$die2" -eq $ZERO ]
  do
    let "die2 = $RANDOM % $SPOTS" # 두번째를 굴리면.
  done

let "throw = $die1 + $die2"
echo "두 주사위를 던진 결과 = $throw"
echo


exit 0
```

RANDOM 이 얼마나 랜덤할까요? 이걸 확인해 보는 가장 좋은 방법은 RANDOM 이 발생 시키는 "랜덤"한 숫자의 분포를 추적하는 스크립트를 만들어 보는 것입니다. 그럼 RANDOM 주사위를 몇 번 던져 봅시다.

예 9-20. RANDOM 으로 주사위를 던지기

```
#!/bin/bash
# RANDOM 이 얼마나 랜덤한가?

RANDOM=$$       # 스크립트의 프로세스 ID 를 써서 랜덤 넘버 발생기의 seed를 다시 생
성.

PIPS=6          # 주사위는 눈이 6개죠.
MAXTHROWS=600   # 시간이 남아 돌면 이 숫자를 더 늘려보세요.
throw=0         # 던진 숫자.

zeroes=0        # 초기화를 안 하면 0이 아니라 널이 되기 때문에
ones=0          # 0으로 초기화 해야 됩니다.
twos=0
threes=0
fours=0
fives=0
```

```
sixes=0

print_result ()
{
echo
echo "ones =   $ones"
echo "twos =   $twos"
echo "threes = $threes"
echo "fours =  $fours"
echo "fives =  $fives"
echo "sixes =  $sixes"
echo
}

update_count()
{
case "$1" in
  0) let "ones += 1";;   # 주사위에 "0"이 없으니까 이걸 1이라고 하고
  1) let "twos += 1";;   # 이걸 2라고 하고.... 등등..
  2) let "threes += 1";;
  3) let "fours += 1";;
  4) let "fives += 1";;
  5) let "sixes += 1";;
esac
}

echo


while [ "$throw" -lt "$MAXTHROWS" ]
do
  let "die1 = RANDOM % $PIPS"
  update_count $die1
  let "throw += 1"
done

print_result

# RANDOM 이 정말 랜덤하다면 결과 분포는 확실히 고르게 나올 것입니다.
# $MAXTHROWS 가 600 일 때는 각각은 100 에서 20 정도의 차이를 두고 분산되어야 합니
다.
#
# 주의할 것은 RANDOM 이 의사난수(pseudorandom) 발생기이기 때문에
# 진짜로 랜덤한 것은 아니라는 점입니다.

# 쉬운 연습문제 하나 낼께요.
# 이 스크립트를 동전 1000 번 뒤집는 걸로 바꿔보세요.
# "앞"이나 "뒤" 중에 하나가 나오겠죠?
```

```
exit 0
```

위의 예제에서 살펴본 것처럼 RANDOM 발생기가 실행될 때마다 랜덤용 seed를 “다시” 만들어 주는 것이 좋습니다. RANDOM에 같은 seed를 쓰면 같은 순서의 숫자가 반복됩니다(이는 C의 *random()* 함수의 동작을 그대로 반영해 줍니다).

예 9-21. RANDOM 에 seed를 다시 지정해 주기

```
#!/bin/bash
# seeding-random.sh: RANDOM 변수에 seed 적용.

MAXCOUNT=25       # 발생시킬 숫자 갯수.

random_numbers ()
{
count=0
while [ "$count" -lt "$MAXCOUNT" ]
do
  number=$RANDOM
  echo -n "$number "
  let "count += 1"
done
}

echo; echo

RANDOM=1          # 랜덤 넘버 발생기에 RANDOM seed 세팅.
random_numbers

echo; echo

RANDOM=1          # 똑같은 seed 사용....
random_numbers    # ... 완전히 똑같은 수열이 발생.

echo; echo

RANDOM=2          # 다른 seed 로 재시도...
random_numbers    # 다른 수열 발생.

echo; echo

# RANDOM=$$  라고 하는 것은 RANDOM 에 스크립트의 프로세스 ID로 seed를 세팅하는 것
입니다.
# 'time' 이나 'date'로 할 수도 있겠네요.

# 더 멋지게 해 보죠.
SEED=$(head -1 /dev/urandom | od -N 1 | awk '{ print $2 }')
# /dev/urandom(시스템에서 제공하는 의사난수 "디바이스")에서 얻은 의사난수
# 그 다음 "od"를 이용해서 출력 가능한(8진수) 숫자로 이루어진 줄로 변환.
```

```
# 마지막으로 "awk"를 써서 SEED 에서 쓸 한 개의 숫자를 뽑아냄.
RANDOM=$SEED
random_numbers

echo; echo

exit 0
```

> 참고: `/dev/urandom` 디바이스 파일은 `$RANDOM` 보다 더 "랜덤"한 의사난수를 발생 시켜줍니다. **`dd if=/dev/urandom of=targetfile bs=1 count=XX`** 라고 하면 잘 분산된 가상랜덤 값을 갖는 파일을 만들어 줍니다. 하지만, 이런 숫자를 스크립트의 변수에 할당하는 것은 od(위의 예제처럼)나 dd(예 12-33 참고)로 필터링을 하는 등의 처리가 필요합니다.

9.7. 이중소괄호(The Double Parentheses Construct)

let 명령어와 비슷하게 **((...))** 도 산술 확장과 계산을 할 수 있습니다. **`a=$((5 + 3))`** 같은 간단한 형태의 식은 "a" 를 "5 + 3"인 8로 만들어 줍니다. 어쨌든 이 이중소괄호는 C 형태의 변수 조작을 가능하게 해주는 메카니즘입니다.

예 **9-22. C** 형태의 변수 조작

```
#!/bin/bash
# ((...)) 를 써서 C 형태로 변수 조작하기.


echo

(( a = 23 ))  # "=" 양쪽에 빈 칸을 두어 변수 세팅하기, C 형태.
echo "a (initial value) = $a"

(( a++ ))     # 'a'를 후위증가, C 형태.
echo "a (after a++) = $a"

(( a-- ))     # 'a'를 후위감소, C 형태.
echo "a (after a--) = $a"


(( ++a ))     # 'a'를 전위증가, C 형태.
echo "a (after ++a) = $a"

(( --a ))     # 'a'를 전위감소, C 형태.
echo "a (after --a) = $a"

echo
```

```
(( t = a<45?7:11 ))   # C 형태의 3중 연산자.
echo "If a < 45, then t = 7, else t = 11."
echo "t = $t "        # 되네요!

echo


# -----------------------------
# 이스터 에그(Easter Egg) 경고!
# -----------------------------
#  Bash에는 ksh에서 많은 부분을 따온 C 형태의 연산자가
#+ 문서화되지 않은 형태로 많이 존재합니다.
#  Bash 문서에서는 ((...)) 를 쉘 연산이라고 합니다만,
#+ 그 이상의 것이 존재합니다.
#  비밀을 밝혀서 미안해요, Chet.

# ((...)) 를 쓴 "for", "while" 루프도 참고하세요.

# 이 이스터 에그들은 Bash 버전 2.04 이후에서만 동작합니다.

exit 0
```

예 10-11 참고.

주석

1. 현재 돌고 있는 스크립트의 pid 는 `$$` 입니다.
2. "인자"(argument)나 "매개변수" (parameter)는 흔히 서로 바꿔서 쓰기 때문에 이 문서에서는 스크립트나 함수로 넘겨지는 변수를 나타내는 같은 의미로 쓰입니다.
3. 이 규칙은 명령어줄 인자나 함수로 넘겨지는 매개변수에도 적용됩니다.
4. 비대화형 스크립트에서 $parameter 가 널이면 127 종료 상태를 갖고 종료됩니다("command not found" 에러 코드).

10장. 루프와 분기(Loops and Branches)

코드 블럭에 대한 연산은 구조화되고, 유기적인 쉘 스크립트에 있어서 핵심적 역할을 합니다. 루프와 분기문들로서 이런 기능들을 수행할 수 있습니다.

10.1. 루프

루프란 루프 제어 조건이 참인 동안에 여러 명령어들을 반복적으로 수행하는 코드 블럭입니다.

for 루프

for (in)

다음은 기본적인 루프문인데 C의 루프문과는 상당한 차이를 보입니다.

for *arg* in [*list*]
do
command...
done

참고: 루프의 각 단계마다 *list*의 값들이 *arg*에 들어갑니다.

```
for arg in "$var1" "$var2" "$var3" ... "$varN"
# 루프 1 단계, $arg = $var1
# 루프 2 단계, $arg = $var2
# 루프 3 단계, $arg = $var3
# ...
# 루프 N 단계, $arg = $varN

# [list]의 인자들은 쿼우팅을 해서 낱말 조각남(word splitting)을 막아 줘야함.
```

list 인자에는 와일드 카드가 올 수도 있습니다.

do를 **for**와 한 줄에 쓴다면 list 뒤에 세미콜론이 있어야 합니다.

for *arg* in [*list*] ; do

예 **10-1.** 간단한 **for** 루프

```
#!/bin/bash
# 떠돌이별 목록.
```

```
for planet in Mercury Venus Earth Mars Jupiter Saturn Uranus Nep-
tune Pluto
do
  echo $planet
done

echo

# 따옴표로 묶인 전체 '목록'은 한 개의 변수를 만들어 냅니다.
for planet in "Mercury Venus Earth Mars Jupiter Saturn Uranus Nep-
tune Pluto"
do
  echo $planet
done

exit 0
```

> 참고: 각 `[list]` 항목은 다중 인자를 가질수 있습니다. 이는 인자가 그룹으로 되어 있을 때 유용한데 이런 경우에는 **set** 명령어(예 11-10 참고)를 써서 `[list]` 각 항목이 위치 매개변수로 파싱되어 할당되도록 해 주면 됩니다.

예 10-2. 각 [list] 항목이 인자를 두 개씩 갖는 for 문

```
#!/bin/bash
# 떠돌이별 재검토.

# 각 떠돌이별의 이름과 해(sun)까지 거리를 한 쌍으로 묶음.

for planet in "Mer-
cury 36" "Venus 67" "Earth 93"  "Mars 142" "Jupiter 483"
do
  set -- $planet  # "planet" 변수를 파싱해서 위치 매개변수로 세팅.
  # "--" 를 쓰면 $planet 이 널이거나 대쉬 문자로 시작하는 등의 까다로운 상황
을 처리해 줍니다.

  # 원래의 위치 매개변수는 덮어써지기 때문에 다른 곳에 저장해 놓아야 할지도 모릅니
다.
  # 배열을 써서 해 볼 수 있겠네요.
  #        original_params=("$@")

  echo "$1 해까지 거리 $2,000,000 마일"
done

# (S.C. 가 확실한 설명을 더 해 줬습니다.)

exit 0
```

for 문의 **`[list]`**에 변수가 들어갈 수도 있습니다.

예 **10-3. Fileinfo:** 변수에 들어 있는 파일 목록에 대해 동작

```
#!/bin/bash
# fileinfo.sh

FILES="/usr/sbin/privatepw
/usr/sbin/pwck
/usr/sbin/go500gw
/usr/bin/fakefile
/sbin/mkreiserfs
/sbin/ypbind"     # 여러분이 궁금해하는 파일들 목록.
                  # 더미 파일인 /usr/bin/fakefile 을 그냥 넣었습니다.

echo

for file in $FILES
do

  if [ ! -e "$file" ]       # 파일이 존재하는지 확인.
  then
    echo "$file 은 존재하지 않는 파일입니다."; echo
    continue                # 다음 파일을 확인.
   fi

  ls -l $file | awk '{ print $9 "         파일 크기: " $5 }'  # 2개의 필
드를 출력.
  whatis `basename $file`   # 파일에 대한 정보.
  echo
done

exit 0
```

`[list]`에는 파일명 globbing이 올 수도 있습니다. 즉, 파일명 확장을 위한 와일드 카드를 쓸 수 있습니다.

예 **10-4. for** 문에서 파일 조작하기

```
#!/bin/bash
# list-glob.sh: "globbing"으로 for 루트의 [list] 만들어내기.

echo

for file in *
do
  ls -l "$file"  # $PWD(현재 디렉토리)의 모든 파일을 나열.
  # 와일드 카드 문자인 "*"는 모든 문자와 일치합니다. 기억나죠?
```

```
  # 하지만 "globbing"에서는 점(dot) 파일은 일치하지 않습니다.

  #  만약에 패턴이 아무 파일과 일치하지 않는다면 그냥 자기 자신으로 확장되는데,
  #+ 이걸 피하려면 nullglob 옵션을 주면 됩니다.
  #  (shopt -s nullglob).
  #  S.C.의 지적 사항.
done

echo; echo

for file in [jx]*
do
  rm -f $file    # $PWD에서 "j"나 "x"로만 시작하는 모든 파일을 지움.
  echo "\"$file\" 이 지워졌습니다."
done

echo

exit 0
```

for 문에서 **`in [list]`**을 안 써주면 명령어 줄에서 넘어온 인자인 $@에 대해서 동작합니다. 이런 식으로 처리하는 멋진 예제를 보려면 예 A-11를 참고하세요.

예 10-5. `in [list]`가 빠진 for 문

```
#!/bin/bash

# 인자를 줘서 실행시켜도 보고 안 줘서 실행시켜 본 다음, 어떻게 되는지 보세요.

for a
do
 echo -n "$a "
done

# 'in list' 가 없기 때문에 '$@'에 대해서 동작합니다.
# ('$@'는 공백문자를 포함하는 명령어줄 인자 리스트).

echo

exit 0
```

for 문의 **`[list]`**에 명령어 치환를 쓸 수도 있습니다. 예 12-31과 예 10-9, 예 12-29를 참고하세요.

예 10-6. for 문의 [list]에 명령어 치환 쓰기

```
#!/bin/bash
# for 루프의 [list]에 명령어 치환을 쓰기.
```

```
NUMBERS="9 7 3 8 37.53"

for number in `echo $NUMBERS`  # for number in 9 7 3 8 37.53
do
  echo -n "$number "
done

echo
exit 0
```

다음은 [list]를 만들어 낼 때 좀 더 복잡한 명령어 치환을 쓰는 예제입니다.

예 10-7. 이진 파일에 **grep** 걸기

```
#!/bin/bash
# bin-grep.sh: 이진 파일에서 일치하는 문자열 찾아내기.

# "grep"을 이진 파일에 걸기.
# "grep -a"라고 해도 비슷합니다.

E_BADARGS=65
E_NOFILE=66

if [ $# -ne 2 ]
then
  echo "사용법: `basename $0` string filename"
  exit $E_BADARGS
fi

if [ ! -f "$2" ]
then
  echo "\"$2\" 은 존재하지 않는 파일입니다."
  exit $E_NOFILE
fi


for word in $( strings "$2" | grep "$1" )
# "strings" 명령어는 이진 파일에 들어 있는 문자열들을 보여주고,
# 그 출력을 "grep"에 파이프로 걸어 원하는 문자열을 찾아냅니다.
do
  echo $word
done

# S.C. 가 위의 for 루프는 다음과 같이 더 간단하게 할 수 있다고 지적해 주었습니
다.
#    strings "$2" | grep "$1" | tr -s "$IFS" '[\n*]'
```

```
# "./bin-grep.sh mem /bin/ls"  같은 식으로 해서 이 스크립트를 연습해 보세요.

exit 0
```

다음도 명령어 치환으로 [list]를 만들어 내는 예제입니다.

예 **10-8.** 특정 디렉토리의 모든 바이너리 파일에 대해 원저작자**(authorship)**를 확인 하기

```
#!/bin/bash
# findstring.sh: 주어진 디렉토리의 이진 파일에서 특정한 문자열 찾아내기.

directory=/usr/bin/
fstring="Free Software Foundation"  # FSF에서 만든 파일에는 어떤 것이 있
나 알아볼까요?

for file in $( find $directory -type f -name '*' | sort )
do
  strings -f $file | grep "$fstring" | sed -e "s%$directory%%"
  # 여기서는 "sed" 표현식의 $directory 가 원래 구분자인 "/"를 포함하고 있기 때
문에
  # "/" 구분자를 다른 것(%)으로 바꿔줘야 합니다.
  # 이렇게 하지 않으면 에러 메세지가 납니다(sed -e "s/$directory//" 라
고 한 번 해보세요).
done

exit 0

# 쉬운 연습문제 하나 내겠습니다:
# $directory 와 $fstring 을 명령어줄에서 받아 들일 수 있도록 수정해 보세요.
```

for 루프의 출력은 다른 명령어로 파이프를 걸 수도 있습니다.

예 **10-9.** 디렉토리에 들어 있는 심볼릭 링크들을 나열하기

```
#!/bin/bash
# symlinks.sh: 디렉토리의 심볼릭 링크 파일들을 나열하기.

ARGS=1                 # 명령어줄 인자는 한 개만 필요.

if [ $# -ne "$ARGS" ]  # 인자가 1 개가 아니면...
then
  directory=`pwd`      # 현재 작업 디렉토리
else
  directory=$1
fi

echo "\"$directory\" 디렉토리의 심볼릭 링크들"

for file in "$( find $directory -type l )"   # -type l = 심볼릭 링크
```

```
do
  echo "$file"
done | sort                                   # 파일 목록을 정렬.

#  Dominik 'Aeneas' Schnitzer 가 지적한대로,
#+ $( find $directory -type l ) 을 쿼우트 안 해주면
#+ 파일이름에 공백문자가 들어있는 파일에 대해서는 제대로 동작하지 않습니다.

exit 0
```

방금 전의 예제에 약간의 변경만 가하면 루프의 표준출력을 파일로 재지향 시킬 수 있습니다.

예 **10-10.** 디렉토리에 들어 있는 심볼릭 링크들을 파일로 저장하기

```
#!/bin/bash
# symlinks.sh: 디렉토리에 들어 있는 심볼릭 링크를 나열하기.

ARGS=1                 # 명령어줄 인자가 한 개 있어야 됩니다.
OUTFILE=symlinks.list  # 저장할 파일

if [ $# -ne "$ARGS" ]  # 인자가 1개가 아니라면...
then
  directory=`pwd`      # 현재 작업 디렉토리
else
  directory=$1
fi

echo "\"$directory\" 디렉토리의 심볼릭 링크들"

for file in "$( find $directory -type l )"   # -type l = 심볼릭 링크
do
  echo "$file"
done | sort > "$OUTFILE"                      # 루프의 표준 출력이
#           ^^^^^^^^^^^^                        저장될 파일로 재지향 됩니
다.

exit 0
```

이중 소괄호를 쓰면 C 프로그래머들에게 익숙한 형태의 **for** 루프 문법도 쓸 수 있습니다.

예 **10-11. C** 형태의 **for** 루프

```
#!/bin/bash
# 10까지 세는 두 가지 방법.

echo
```

```
# 표준 문법.
for a in 1 2 3 4 5 6 7 8 9 10
do
  echo -n "$a "
done

echo; echo

# +==========================================+

# 이제는 C 형태의 문법을 써서 똑같은 일을 해 보겠습니다.

LIMIT=10

for ((a=1; a <= LIMIT ; a++))  # 이중 소괄호와 "$" 없는 "LIMIT".
do
  echo -n "$a "
done                           # 'ksh93' 에서 빌려온 문법.

echo; echo

# +=========================================================================+

# C 의 "콤마 연산자"를 써서 두 변수를 동시에 증가시켜 보겠습니다.

for ((a=1, b=1; a <= LIMIT ; a++, b++))  # 콤마를 쓰면 여러 연산을 함
께 할 수 있습니다.
do
  echo -n "$a-$b "
done

echo; echo

exit 0
```

예 26-6와 예 26-7도 참고하세요.

자, 이제는 “실제”로 어떻게 쓰이는지 봅시다.

예 10-12. 배치 모드로 **efax** 사용하기

```
#!/bin/bash

EXPECTED_ARGS=2
E_BADARGS=65

if [ $# -ne $EXPECTED_ARGS ]
```

```
# 적당한 명령어 줄 인자가 넘어 왔는지 확인.
then
   echo "사용법: `basename $0` phone# text-file"
   exit $E_BADARGS
fi


if [ ! -f "$2" ]
then
  echo "$2 는 텍스트 파일이 아닙니다."
  exit $E_BADARGS
fi


fax make $2              # 주어진 텍스트 파일에서 팩스 포맷의 파일을 생성.

for file in $(ls $2.0*)  # 변환된 파일을 이어 붙임.
                         # 변수 목록에서 와일드 카드를 사용.
do
  fil="$fil $file"
done

efax -d /dev/ttyS3 -o1 -t "T$1" $fil   # 동작.


# S.C. 가 지적했듯이,
#    efax -d /dev/ttyS3 -o1 -t "T$1" $2.0*
# 라고 하면 for 루프를 안 써도 됩니다만,
# 그렇게 좋은 방법은 아닙니다. ^^

exit 0
```

while

while은 루프 최상단에서 특정 조건을 확인하면서 그 조건이 참일 동안 루프를 계속 돌도록 해 줍니다(종료 상태 0을 리턴합니다).

while [*condition*]
do
command...
done

for/in 경우처럼 **do**를 조건 테스트문과 같은 줄에 쓰려면 세미콜론을 써 줘야 합니다.

while [*condition*] ; do

여기서 설명했던 표준 형태가 아닌 getopts construct같은 특별한 형태의 **while** 문도 있다는 것에 주의하시기 바랍니다.

예 **10-13.** 간단한 **while** 루프

```
#!/bin/bash

var0=0
LIMIT=10

while [ "$var0" -lt "$LIMIT" ]
do
  echo -n "$var0 "        # -n 은 뉴라인을 없애줍니다.
  var0=`expr $var0 + 1`   # var0=$(($var0+1)) 이라고 해도 동작합니다.
done

echo

exit 0
```

예 **10-14.** 다른 **while** 루프

```
#!/bin/bash

echo

while [ "$var1" != "end" ]     # while test "$var1" != "end"
do                             # 라고 해도 동작함.
  echo "변수값을 넣으세요 #1 (끝내려면 end) "
  read var1                    # 'read $var1' 이 아니죠? 왜 그럴까요?
  echo "변수 #1 = $var1"       # "#" 때문에 쿼우트를 해줘야 됩니다.
  # 종료 조건이 루프 처음에 테스트되기 때문에 'end'도 에코됩니다.
  echo
done

exit 0
```

while 문은 다중 조건을 가질 수 있습니다만 오직 마지막 조건이 루프를 끝낼 조건을 결정합니다. 그렇기 때문에 이럴 경우에는 문법이 약간 달라집니다.

예 **10-15.** 다중 조건 **while** 루프

```
#!/bin/bash

var1=unset
previous=$var1

while echo "이전 변수 = $previous"
```

```
      echo
      previous=$var1
      [ "$var1" != end ] # 바로 전의 "var1"이 무엇이었는지 계속 확인.
      # "while"에는 4가지 조건이 있지만 오직 마지막 조건이 루프를 제어합니다.
      # *마지막* 종료 상태가 중요하다는 말씀.
do
echo "변수값을 넣으세요 #1 (끝내려면 end) "
  read var1
  echo "변수 #1 = $var1"
done

# 이 스크립트가 어떻게 돌아가는지 알아내 보세요.
# 약간 미묘한(tricky) 부분이 있습니다.

exit 0
```

for 루프처럼 **while**도 이중소괄호를 써서 C 형태의 문법을 적용할 수 있습니다(예 9-22 참고).

예 **10-16. C** 형태의 문법을 쓰는 **while** 루프

```
#!/bin/bash
# wh-loopc.sh: "while" 루프에서 10까지 세기.

LIMIT=10
a=1

while [ "$a" -le $LIMIT ]
do
  echo -n "$a "
  let "a+=1"
done           # 아직은 별로 놀랄게 없네요.

echo; echo

# +=================================================================+

# 이제 똑같은 것을 C 형태의 문법으로 해 봅시다.

((a = 1))      # a=1
# 이중 소괄호에서는 변수를 세팅할 때 C 처럼 빈 칸을 넣어도 됩니다.

while (( a <= LIMIT ))   # 이중 소괄호, 변수 앞에 "$"가 없네요.
do
  echo -n "$a "
  ((a += 1))   # let "a+=1"
  # 역시 되는군요.
  # 이중 소괄호를 쓰면 C 문법처럼 변수를 증가시킬 수 있군요.
done
```

```
echo

# 이제 C 프로그래머도 Bash 를 쓸 때 편안하게 쓸 수 있겠습니다.

exit 0
```

참고**: while** 루프 마지막에 <을 써 표준입력을 파일에서 재지향 받을 수 있습니다.

until

until은 루프 최상단에서 특정 조건을 확인하면서 그 조건이 거짓일 동안 루프를 계속 돌도록 해 줍니다(**while**과 반대).

until [*condition-is-true*]
do
command...
done

주의할 점은 **until**이 몇몇 프로그래밍 언어에서 비슷한 형태와는 다르게 루프 처음에서 끝내는 조건을 검사한다는 것 입니다.

for/in 경우처럼 **do**를 조건문과 한 줄에 같이 쓰려면 세미콜론을 적어줘야 합니다.

until [*condition-is-true*] ; do

예 **10-17. until** 루프

```
#!/bin/bash

until [ "$var1" = end ] # 테스트 조건이 루프 최상단에 들어갑니다.
do
  echo "변수값을 넣으세요 #1 "
  echo "(끝내려면 end)"
  read var1
  echo "변수 #1 = $var1"
done

exit 0
```

10.2. 중첩된 루프

중첩된 루프는 루프 안에 루프가 들어 있는 형태를 말합니다. 바깥쪽 루프의 매 단계마다 안쪽 루프를 돌리는데, 이 전체 동작은 바깥쪽 루프가 끝날 때까지 계속 됩니다. 당연한 얘기지만, 안쪽 루프나 바깥쪽 루프에서 **break**가 나타나면 전체 동작을 중단시킵니다.

예 **10-18.** 중첩된 루프

```
#!/bin/bash
# Nested "for" loops.

outer=1             # 바깥쪽 루프 카운트 셋.

# 바깥쪽 루프 시작.
for a in 1 2 3 4 5
do
  echo "바깥쪽 루프의 $outer 단계."
  echo "---------------------"
  inner=1           # 안쪽 루프 리셋.

  # 안쪽 루프 시작.
  for b in 1 2 3 4 5
  do
    echo "안쪽 루프의 $inner 단계."
    let "inner+=1"  # 안쪽 루프 카운터 증가.
  done
  # 안쪽 루프의 끝.

  let "outer+=1"    # 바깥쪽 루프 카운터 증가.
  echo              # 바깥쪽 루프 매 단계마다 빈 줄 삽입.
done
# 바깥쪽 루프의 끝.

exit 0
```

중첩된 "while" 루프의 실례를 보려면 예 26-4을 참고하고, "until"안에 중첩된 "while"을 보려면 예 26-5을 참고하세요.

10.3. 루프 제어

루프의 동작에 영향을 미치는 명령어들

break
continue

break와 **continue** 루프 제어 명령어 [1] 는 다른 프로그래밍 언어들과 정확히 같은 동작을 합니다. **break** 명령어는 자신이 속해 있는 루프를 끝내고, **continue**는 해당 루프 사이클 내에 남아 있는 나머지 명령어들을 건너 뛰고 다음 단계의 루프를 수행합니다.

예 10-19. 루프에서 **break**와 **continue**의 영향

```
#!/bin/bash

LIMIT=19  # 상한선

echo
echo "3,11을 제외하고 1부터 20까지 출력."

a=0

while [ $a -le "$LIMIT" ]
do
 a=$(($a+1))

 if [ "$a" -eq 3 ] || [ "$a" -eq 11 ]  # 3,11을 제외
 then
   continue  # 이번 루프의 나머지 부분을 건너뜀.
 fi

 echo -n "$a "
done

# 독자들을 위한 연습문제:
# 루프가 왜 20까지 찍을까요?

echo; echo

echo 1부터 20까지 찍지만 2다음에 무슨 일인가가 일어납니다.

##################################################################

# 똑같은 루프지만 'continue'를 'break'로 바꿨습니다.

a=0

while [ "$a" -le "$LIMIT" ]
do
 a=$(($a+1))

 if [ "$a" -gt 2 ]
 then
   break  # 루프 나머지를 모두 건너뜀.
 fi

 echo -n "$a "
done

echo; echo; echo
```

```
exit 0
```

break 명령어는 매개변수를 받을 수도 있습니다. 매개변수 없는 **break**는 자신이 속해 있는 제일 안쪽 루프를 끝내지만, **break N**은 *N* 레벨의 루프를 빠져나갑니다.

예 **10-20.** 여러 단계의 루프에서 탈출하기

```
#!/bin/bash
# break-levels.sh: 루프에서 탈출하기.

# "break N" 은 N 레벨의 루프를 빠져나갑니다.

for outerloop in 1 2 3 4 5
do
  echo -n "$outerloop 그룹:   "

  for innerloop in 1 2 3 4 5
  do
    echo -n "$innerloop "

    if [ "$innerloop" -eq 3 ]
    then
      break  # break 2  라고 하면 어떻게 될까요?
             # (안쪽과 바깥쪽 루프 모두에서 "탈출"(break)합니다.)
    fi
  done

  echo
done

echo

exit 0
```

continue 명령어도 **break**와 비슷하게 매개변수를 받아 들일수 있습니다. 매개변수 없는 **continue**는 자신이 속한 루프의 현재 작업을 끝내고 다음번으로 건너 뛰지만 **continue N**은 자신이 속한 루프 레벨의 모든 단계를 건너 뛰고 N 레벨 위에 속하는 루프의 다음 단계로 건너 뜁니다.

예 **10-21.** 더 상위 루프 레벨에서 계속하기**(continue)**

```
#!/bin/bash
# "continue N" 명령어, N 번째 루프에서 계속하기(continue).

for outer in I II III IV V           # 바깥쪽 루프
do
  echo; echo -n "Group $outer: "
```

```
  for inner in 1 2 3 4 5 6 7 8 9 10  # 안쪽 루프
  do

    if [ "$inner" -eq 7 ]
    then
      continue 2  # "바깥쪽 루프"인 2번째 레벨의 루프에서 계속 진행합니다.
                  # 윗줄을 그냥 "continue"라고 하면 보통의 루프 동작이 일어납
니다.
    fi

    echo -n "$inner "  # 8 9 10 은 절대 에코되지 않습니다.
  done

done

echo; echo

# 독자들을 위한 연습문제:
# "continue N"을 쓰는 쓸모있는 스크립트를 짜 보세요.

exit 0
```

경고

continue N은 아무리 의미 있는 상황에서 썼더라도 이해하기 어렵고 쓰기 까다롭기 때문에 안 쓰는게 제일 좋습니다.

10.4. 테스트와 분기(Testing and Branching)

case와 **select**는 코드 블럭을 반복해서 수행하지 않기 때문에 기술적으로 루프가 아닙니다. 하지만 루프가 하는 것처럼 특정 블럭의 위나 아래에서 주어진 조건에 따라 프로그램 흐름을 조정해 줍니다.

코드 블럭에서 프로그램 흐름을 조절하기

case (in) / esac

case는 C/C++의 **switch**와 동일합니다. 조건에 따라 여러개의 코드 블럭중 하나로 분기할 수 있게 해주는데, 여러개의 if/then/else의 간단한 표기법처럼 동작하기 때문에 메뉴같은 것을 만들 때 적당합니다.

case "$*variable*" in
"$*condition1*")

```
command...
;;
"$condition2" )
command...
;;
esac
```

참고:

- 낱말 조각남(word splitting)이 일어나지 않기 때문에 꼭 variable을 쿼우팅 하지 않아도 됩니다.
- 각 조건들은 오른쪽 괄호,)로 끝납니다.
- 각 조건 블럭은 이중 세미콜론, ;;.
- 전체 **case** 블럭은 **esac**로 끝납니다(*case*를 거꾸로 스펠링).

예 10-22. case 쓰기

```
#!/bin/bash

echo; echo "아무키나 누른 다음 리턴을 치세요."
read Keypress

case "$Keypress" in
  [a-z]   ) echo "소문자";;
  [A-Z]   ) echo "대문자";;
  [0-9]   ) echo "숫자";;
  *       ) echo "구두점이나, 공백문자 등등";;
esac  # [대괄호]속 범위의 문자들을 받아 들입니다.

# 독자들용 연습문제:
# 이 스크립트는 한개의 키누름만 받아들이고 종료합니다.
# 이를 키가 눌릴때마다 무슨 키인지 계속 보여주면서
# 키가 "X"일 경우에만 종료하도록 고쳐보세요.
# 힌트: "while"루프로 다 감싸면?

exit 0
```

예 10-23. case로 메뉴 만들기

```
#!/bin/bash

# 조잡한 주소 데이타베이스

clear # 화면을 정리하고
```

```
echo "          주소록"
echo "          ------"
echo "다음중 한 명을 고르세요:"
echo
echo "[E]vans, Roland"
echo "[J]ones, Mildred"
echo "[S]mith, Julie"
echo "[Z]ane, Morris"
echo

read person

case "$person" in
# 변수가 쿼우트 된 것에 주의하세요.

  "E" | "e" )
  # 대소문자 모두 인식.
  echo
  echo "Roland Evans"
  echo "4321 Floppy Dr."
  echo "Hardscrabble, CO 80753"
  echo "(303) 734-9874"
  echo "(303) 734-9892 fax"
  echo "revans@zzy.net"
  echo "Business partner & old friend"
  ;;
# 이중 세미콜론이 각 옵션을 끝내게 해 주는 것을 눈여겨 봐두세요.

  "J" | "j" )
  echo
  echo "Mildred Jones"
  echo "249 E. 7th St., Apt. 19"
  echo "New York, NY 10009"
  echo "(212) 533-2814"
  echo "(212) 533-9972 fax"
  echo "milliej@loisaida.com"
  echo "Girlfriend"
  echo "Birthday: Feb. 11"
  ;;

# Smith & Zane 은 나중에 추가.

          * )
   # 디폴트 옵션.
   # 그냥 리턴을 쳐도 여기로 옵니다.
   echo
   echo "아직 등록이 안 돼 있습니다."
  ;;
```

```
esac

echo

# 독자용 연습문제:
# 입력을 한 번만 받고 끝내지 말고 계속 받을 수 있게 고쳐보세요.

exit 0
```

명령어줄 매개변수를 확인하려고 할 때, 아주 창의적인 방법으로 **case**를 사용하는 법을 보여 드리겠습니다.

```
#! /bin/bash

case "$1" in
"") echo "사용법: ${0##*/} <filename>"; exit 65;;  # 명령어 줄 매개변수
를 안 적었거나
                                                  # 첫번째 매개변
수가 비어 있을 때
# ${0##*/} 는 매개변수 치환인 ${var##pattern} 이기 때문에 결과는 $0 이 됩니
다.

-*) FILENAME=./$1;;   # 첫번째 인자($1)인 filename이 대쉬(-)로 시작한다면
                      # ./$1 로 바꿉니다.
                      # 따라서, 다른 명령어들은 이것을 옵션으로 해석하지 못하
게 됩니다.

* ) FILENAME=$1;;     # 아무 것도 아니면, $1.
esac
```

예 10-24. case용 변수를 만들기 위해서 명령어 치환 쓰기

```
#!/bin/bash
# "case"용 변수를 만들기 위해서 명령어 치환 쓰기..

case $( arch ) in   # "arch" 는 머신 아키텍쳐를 리턴합니다.
i386 ) echo "80386 기반의 머신";;
i486 ) echo "80486 기반의 머신";;
i586 ) echo "Pentium 기반의 머신";;
i686 ) echo "Pentium2+ 기반의 머신";;
*    ) echo "다른 형태의 머신";;
esac

exit 0
```

case는 globbing용 패턴으로 문자열을 필터링 할 수 있습니다.

예 10-25. 간단한 문자열 매칭

```
#!/bin/bash
# match-string.sh: 간단한 문자열 매칭

match_string ()
{
  MATCH=0
  NOMATCH=90
  PARAMS=2     # 이 함수는 2개의 인자가 필요합니다.
  BAD_PARAMS=91

  [ $# -eq $PARAMS ] || return $BAD_PARAMS

  case "$1" in
  "$2") return $MATCH;;
  *   ) return $NOMATCH;;
  esac

}


a=one
b=two
c=three
d=two


match_string $a     # 잘못된 인자 갯수
echo $?             # 91

match_string $a $b  # 일치하지 않음
echo $?             # 90

match_string $b $d  # 일치함
echo $?             # 0


exit 0
```

예 10-26. 입력이 알파벳인지 확인하기

```
#!/bin/bash
# 문자열을 필터링하기 위해서 "case" 구문 쓰기.

SUCCESS=0
FAILURE=-1

isalpha ()  # 입력 문자열의 "첫번째 문자"가 알파벳인지 아닌지 확인.
```

```
{
if [ -z "$1" ]                # 인자 없이 넘어왔군.
then
  return $FAILURE
fi

case "$1" in
[a-zA-Z]*) return $SUCCESS;;  # 문자로 시작하는지.
*        ) return $FAILURE;;
esac
}             # C 의 "isalpha()" 함수와 비교해 보세요.


isalpha2 ()   # "문자열 전체"가 알파벳인지 확인.
{
  [ $# -eq 1 ] || return $FAILURE

  case $1 in
  *[!a-zA-Z]*|"") return $FAILURE;;
               *) return $SUCCESS;;
  esac
}



check_var ()  # isalpha() 의 프론트엔드(front-end)
{
if isalpha "$@"
then
  echo "$* = 알파벳"
else
  echo "$* = 알파벳 아님"  # 인자가 없어도 "알파벳 아님"임.
fi
}

a=23skidoo
b=H3llo
c=-What?
d=`echo $b`   # 명령어 치환.

check_var $a
check_var $b
check_var $c
check_var $d
check_var     # 인자없이 부르면 어떻게 될까요?


# S.C. 가 더 좋게 고침.
```

```
exit 0
```

select

select는 Korn 쉘에서 따온 것인데 메뉴를 만들때 쓸 수 있습니다.

select *`variable`* [in *`list`*]
do
`command`...
break
done

사용자가 list에 있는 것중 하나를 고를 수 있게 해 줍니다. 기본적으로 `PS3(#? )` 프롬프트를 쓰고 이 값은 바꿀 수 있다는 것에 주의하기 바랍니다.

예 **10-27. select**로 메뉴 만들기

```
#!/bin/bash

PS3='제일 좋아하는 야채를 고르세요: ' # 프롬프트 문자열 세트.

echo

select vegetable in "콩" "당근" "감자" "양파" "순무"
do
  echo
  echo "제일 좋아하는 야채가 $vegetable 이네요."
  echo "깔깔~~"
  echo
  break  # 여기에 'break'가 없으면 무한 루프를 돕니다.
done

exit 0
```

`in list`를 안 쓰면 **select**는 스크립트나 **select**를 포함하고 있는 함수로 넘어온 명령어줄 인자(`$@`)을 사용합니다.

`in list`가 빠졌을 경우를

for *`variable`* [in *`list`*]

의 경우와 비교해 보세요.

예 **10-28.** 함수에서 **select**를 써서 메뉴 만들기

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

```
PS3='제일 좋아하는 야채를 고르세요: '

echo

choice_of()
{
select vegetable
# select 에 [in list] 가 빠져있기 때문에, 함수로 넘어온 인자를 씁니다.
do
  echo
  echo "제일 좋아하는 야채가 $vegetable 군요."
  echo "껄껄~~"
  echo
  break
done
}

choice_of 콩  쌀  당근  무  토마토  시금치
#         $1  $2  $3    $4  $5      $6
#         choice_of() 함수로 넘어갑니다.

exit 0
```

주석

1. while나 case가 키워드인 반면에 **break**나 **continue**는 쉘 내장명령입니다.

11장. 내부 명령어(Internal Commands and Builtins)

내부 명령(builtin) 은 Bash 툴 셋에 포함된 명령어로 말 그대로 **bulit in**(고유의, 붙박이의)된 명령어입니다. 내부 명령은 시스템 명령어와 이름이 같을 수도 있지만 이런 경우는 Bash가 내부적으로 다시 구현해 놓은 것입니다. [1] 예를 들어, 하는 일이 거의 동일한 bash의 **echo**는 `/bin/echo`와 다릅니다.

키워드(keyword)는 예약된 낱말, 토큰, 연산자를 말합니다. 키워드는 쉘에서 특별한 의미를 가지면서, 쉘 문법을 형성해 줍니다. 예를 들면, “for”, “while”, “!”가 키워드입니다. 내부 명령(builtin)과 비슷하게 키워드도 Bash 내부에 하드코드(hard-coded)되어 있습니다.

I/O

echo

변수나 표현식을 표준출력으로 출력(예 5-1 참고).

```
echo Hello
echo $a
```

echo 에서 이스케이프 문자들을 찍으려면 `-e` 옵션을 주어야 합니다. 예 6-2를 참고하세요.

보통, 각각의 **echo**는 뉴라인을 출력해 주지만 `-n` 옵션을 주면 이 뉴라인을 안 찍어 줍니다.

참고: **echo**는 파이프에서 여러 명령어중의 하나로 쓰일 수 있습니다.

```
if echo "$VAR" | grep -q txt   # if [[ $VAR = *txt* ]]
then
  echo "$VAR 에는 \"txt\"" 문자열이 들어 있습니다."
fi
```

참고: **echo**를 명령어 치환과 같이 써서 변수 값을 세트할 수 있습니다.

`a=`echo "HELLO" | tr A-Z a-z``

예 12-15, 예 12-2, 예 12-28, 예 12-29를 참고하세요.

경고

echo ‘command‘는 *command* 출력의 모든 라인피드를 지워버리는 것에 주의하기 바랍니다. 보통 \n이 `$IFS`의 공백문자중 하나이기 때문에, Bash는 *command*의 출력에 나타나는 라인피드를 빈 칸으로 바꾸어 **echo**의 인자로 만들어 버립니다.

```
bash$ printf '\n\n1\n2\n3\n\n\n\n'


 1
 2
 3



bash $




bash$ echo "`printf '\n\n1\n2\n3\n\n\n\n'`"


 1
 2
 3
bash $
```

참고**:** 이 명령어는 쉘 내장 명령으로서 `/bin/echo`와 비슷한 동작을 하지만 엄연히 다릅니다.

```
bash$ type -a echo
echo is a shell builtin
echo is /bin/echo
```

printf

printf는 형식화된 출력을 해주는 명령어로서, **echo**의 확장판입니다. C 언어의 `printf`보다 기능이 제한된 변종으로, 문법은 C와 약간 다릅니다.

printf *`format-string`*... *`parameter`*...

`/bin/printf`나 `/usr/bin/printf`의 bash 내장 명령 버전입니다. 더 자세한 내용은 **printf**(시스템 명령어)의 맨 페이지를 참고하세요.

경고

bash의 옛날 버전은 **printf**를 지원하지 않을 수도 있습니다.

예 **11-1. printf**가 실제로 쓰이는 예제

```
#!/bin/bash
# printf demo

PI=3.14159265358979
DecimalConstant=31373
Message1="안녕들 하신가,"
Message2="지구인 여러분."

echo

printf "Pi를 소수점 이하 2 자리만 표시 = %1.2f" $PI
echo
printf "Pi를 소수점 이하 9 자리만 표시 = %1.9f" $PI  # 반올림이 잘 됐죠?

printf "\n"                                  # 라인피드를 찍음.
                                             # 'echo'와 똑같음.

printf "상수 = \t%d\n" $DecimalConstant  # 탭이 들어갔죠(\t)

printf "%s %s \n" $Message1 $Message2

echo

# ==========================================#
# C 함수인 'sprintf'를 시뮬레이션.
# 형식화된 문자열로 변수를 로딩.

echo

Pi12=$(printf "%1.12f" $PI)
echo "Pi를 소수점 12 자리만 표시 = $Pi12"

Msg=`printf "%s %s \n" $Message1 $Message2`
echo $Msg; echo $Msg

# 이렇게 하면, 'sprintf' 함수는 Bash에서 로딩할 수 있는 모듈로
# 접근이 가능해 지지만, 이식성이 있는 구현은 아닙니다.

exit 0
```

printf는 에러 메세지를 형식화 해서 출력해 줄 때 아주 유용하게 쓰입니다.

```
E_BADDIR=65

var=nonexistent_directory

error()
{
  printf "$@" >&2
  # 인자로 넘어온 위치 매개변수를 형식화해서 표준에러로 출력
  echo
  exit $E_BADDIR
}

cd $var || error $"%s 로 바꿀 수 없습니다." "$var"

# Thanks, S.C.
```

read

변수값을 표준입력에서 “읽어 들입니다.” 즉, 키보드에서 사용자의 입력을 받아 처리합니다. `-a` 옵션을 주면 **read**는 변수를 배열로 받아 들입니다(예 26-2 참고).

예 **11-2. read**로 변수 할당하기

```
#!/bin/bash

echo -n "'var1' 변수값을 넣으세요: "
# -n 옵션은 뉴라인을 제거해 줍니다.

read var1
# var1 변수가 read에 의해 세트되기 때문에 var1 앞에 '$'가 없습니다. 주의하세요.

echo "var1 = $var1"


echo

# 하나의 'read' 문으로 여러개의 변수를 세트할 수 있습니다.
echo -n "'var2'와 'var3'의 값을 넣으세요(빈 칸이나 탭으로 구분): "
read var2 var3
echo "var2 = $var2      var3 = $var3"
# 한 개 값만 주게 되면, 다른 변수(들)은 세트가 안 된 상태(널)로 남게 됩니다.

exit 0
```

read로 입력을 넣을 때 보통은 뉴라인 전에 \를 넣어 뉴라인을 무시하게 합니다. `-r` 옵션을 주면 \ 가 문자 그대로 해석되도록 합니다.

예 **11-3. read**로 여러줄의 입력 넣기

```
#!/bin/bash

echo

echo "\\ 로 끝나는 문자열을 입력하고 <ENTER> 를 누르세요."
echo "그 다음에 두 번째 문자열을 입력하고 <ENTER> 를 다시 누르세요."
read var1     # "var1"을 읽을 때 "\" 때문에 뉴라인이 제거됩니다.
              #     첫번째 줄 \
              #     두번째 줄

echo "var1 = $var1"
#     var1 = 첫번째 줄 두번째 줄

# "\"로 끝나는 줄을 만날 때 마다,
# var1에 문자를 계속 입력하기 위해서 다음 줄을 위한 프롬프트를 받게 됩니다.

echo; echo

echo "\\ 로 끝나는 다른 문자열을 입력하고 <ENTER> 를 누르세요."
read -r var2  # -r 옵션은 "\"를 문자 그대로 읽어 들입니다.
              #     첫번째 줄 \

echo "var2 = $var2"
#     var2 = 첫번째 줄 \

# 입력 데이타는 첫번째 <ENTER> 에서 끝나게 됩니다.

echo

exit 0
```

read 명령어는 프롬프트를 보여준 다음 **ENTER**없이 키누름을 받아 들이는 재밌는 옵션을 갖고 있습니다.

```
# ENTER 없이 키누름을 읽음.

read -s -n1 -p "키를 누르세요 " keypress
echo; echo "당신이 누른 키는 "\"$keypress\""입니다."

# -s 는 입력을 에코하지 말라는 옵션입니다.
# -n N 은 딱 N 개의 문자만 받아 들이라는 옵션입니다.
# -p 는 입력을 읽기 전에 다음에 나오는 프롬프트를 에코하라는 옵션입니다.

# 이 옵션들은 순서가 바로 되어 있어야 하기 때문에 쓰기가 약간 까다롭습니다.
```

read 명령어는 표준입력으로 재지향된 파일에서 변수값을 "읽을" 수도 있습니다. 입력 파일이 한 줄 이상이라면 첫 번째 줄만 변수로 할당됩니다. **read**가 하나 이상의 매개변수

를 갖고 있다면 각 변수는 공백 문자로 구분되는 연속적인 문자열로 할당됩니다. 조심하세요!

예 **11-4. read**를 파일 재지향과 같이 쓰기

```
#!/bin/bash

read var1 <data-file
echo "var1 = $var1"
# "data-file"의 첫번째 줄 전체가 var1으로 세팅

read var2 var3 <data-file
echo "var2 = $var2   var3 = $var3"
# "read"의 직관적이지 않은 행동에 주의하세요.
# 1) 입력 파일의 제일 처음으로 돌아가서,
# 2) 각 변수는 줄 전체가 아닌 공백문자로 나누어진 문자열로 세트됨.
# 3) 마지막 변수는 그 줄의 나머지로 세트.
# 4) 변수 갯수가 공백문자로 나누어진 문자열보다 많다면 나머지 변수들은 세트되지 않
음.

echo "------------------------------------------------"

# 위의 문제를 루프로 해결해 보겠습니다.
while read line
do
  echo "$line"
done <data-file
# Heiner Steven 이 이 부분을 지적해 주었습니다.

echo "------------------------------------------------"

# "read"가 읽어 들이는 줄이 공백문자가 아닌 다른 문자로 구분되도록 하려면
# $IFS(내부 필드 구분자, Internal Field Separator)를 쓰면 됩니다.

echo "모든 사용자 목록:"
OIFS=$IFS; IFS=:       # /etc/passwd 는 필드 구분자로 ":"를 씁니다.
while read name passwd uid gid fullname ignore
do
  echo "$name ($fullname)"
done </etc/passwd   # I/O 재지향.
IFS=$OIFS              # 원래 $IFS 를 복구시킴.
# 이 코드도 Heiner Steven 이 제공해 주었습니다.

exit 0
```

파일시스템

cd

익숙한 명령어인 **cd**는 스크립트에서 어떤 명령어가 특정한 디렉토리에서 실행될 필요가 있을 때 그 디렉토리로 옮겨가기 위해 쓰입니다.

```
(cd /source/directory && tar cf -
. ) | (cd /dest/directory && tar xpvf -)
```

[앞서 인용했던 알란 콕스의 예제]

cd가 심볼릭 링크를 무시하고 원래 디렉토리로 가도록 하기 위해서는 `-P`(물리적인, physical) 옵션을 주면 됩니다.

cd - 라고 하면 바로 전 작업 디렉토리인 $OLDPWD 로 옮겨 갑니다.

pwd

현재 작업 디렉토리를 출력(Print Working Directory). 이 명령어는 사용자(혹은 스크립트)의 현재 디렉토리(예 11-5 참고)를 알려줍니다. 내부 변수인 $PWD의 값을 읽는 것과 동일한 효과를 가져옵니다.

pushd
popd
dirs

이 명령어들은 작업 디렉토리를 즐겨찾기에 기억시켜주는 메카니즘으로, 디렉토리간에 순서대로 왔다 갔다 할 수 있게 해 줍니다. 디렉토리 이름을 기억하기 위해서 푸쉬다운 스택을 사용합니다. 옵션을 줘서 디렉토리 스택에 대해서 다양한 조작을 할 수 있습니다.

`pushd dir-name`은 *`dir-name`*을 디렉토리 스택에 넣고 동시에 현재 디렉토리를 그 디렉토리로 옮겨 줍니다.

popd는 디렉토리 스택의 제일 위에 있는 디렉토리를 지우고(pop) 동시에 현재 디렉토리를 그 디렉토리로 옮겨 줍니다.

dirs은 디렉토리 스택의 목록을 보여줍니다 ($DIRSTACK과 같음). **pushd**나 **popd**가 성공한다면 **dirs**가 자동으로 불립니다.

디렉토리 이름을 하드 코딩하지 않고 디렉토리를 여기 저기로 옮겨 다녀야 하는 스크립트가 이 명령어를 쓰면 아주 좋습니다. 스크립트 내에서 디렉토리 스택의 내용을 담고 있는 `$DIRSTACK` 배열 변수에 묵시적으로 접근이 가능하기 때문에 주의해야 합니다.

예 **11-5.** 현재 작업 디렉토리 변경하기

```
#!/bin/bash

dir1=/usr/local
dir2=/var/spool

pushd $dir1
# 자동으로 'dirs'를 실행합니다(디렉토리 스택의 내용을 표준출력으로 뿌림).
echo "Now in directory `pwd`." # 역따옴표(backquote)를 건 'pwd'.
```

```
# 'dir1'에서 아무 일이나 하고,
pushd $dir2
echo "지금은 `pwd` 디렉토리에 있습니다."

# 'dir2'에서 아무 일이나 한 다음,
echo "DIRSTACK 배열의 top 항목은 $DIRSTACK 입니다."
popd
echo "이제 `pwd` 디렉토리로 돌아왔습니다."

# 'dir1'에서 아무 일이나 하세요.
popd
echo "이제 원래의 `pwd` 디렉토리로 돌아왔습니다."

exit 0
```

변수

let

let 명령어는 변수에 대해서 산술 연산을 수행합니다. 많은 경우에 있어서 expr보다 좀 간단한 기능을 수행합니다.

예 **11-6. let**으로 몇 가지 산술 연산을 하기**.**

```
#!/bin/bash

echo

let a=11          # 'a=11' 과 똑같습니다.
let a=a+5         # let "a = a + 5" 와 똑같습니다.
                  # (큰따옴표와 빈 칸을 쓰면 좀 더 읽기가 편하죠)
echo "11 + 5 = $a"

let "a <<= 3"     # let "a = a << 3" 과 똑같습니다.
echo "\"\$a\" (=16) 를 3 번 왼쪽 쉬프트 = $a"

let "a /= 4"      # let "a = a / 4" 와 똑같습니다.
echo "128 / 4 = $a"

let "a -= 5"      # let "a = a - 5" 와 똑같습니다.
echo "32 - 5 = $a"

let "a = a * 10"  # let "a = a * 10" 과 똑같습니다.
echo "27 * 10 = $a"

let "a %= 8"      # let "a = a % 8" 과 똑같습니다.
echo "270 modulo 8 = $a  (270 / 8 = 33, 나머지는 $a)"
```

```
echo

exit 0
```

eval

`eval arg1 [arg2] ... [argN]`

목록에 들어 있는 인자를 명령어로 변환(스크립트 안에서 코드를 만들어 낼 때 유용함).

예 **11-7. eval**의 효과 보여주기

```
#!/bin/bash

y=`eval ls -l`   # y=`ls -l` 과 비슷하지만,
echo $y          # 라인피드가 지워집니다.

y=`eval df`      # y=`df` 와 비슷하지만,
echo $y          # 라인피드가 지워집니다.

# 라인피드(LF)가 없어지기 때문에 출력을 파싱하기가 더 쉬울 것입니다.

exit 0
```

예 **11-8.** 강제로 로그 아웃 시키기

```
#!/bin/bash

y=`eval ps ax | sed -n '/ppp/p' | awk '{ print $1 }'`
# 'ppp'의 프로세스 번호를 찾은 다음,

kill -9 $y   # 죽이고,

# 위에서 한 것은 다음처럼 해도 됩니다.
#  kill -9 `ps ax | awk '/ppp/ { print $1 }'


chmod 666 /dev/ttyS3
# ppp에 SIGKILL을 날리면 직렬 포트의 퍼미션이 바뀌기 때문에
# 이전 퍼미션으로 복구시켜야 됩니다.

rm /var/lock/LCK..ttyS3   # 직렬 포트용 잠금 파일을 지웁니다.

exit 0
```

예 **11-9. "rot13"** 버전

```
#!/bin/bash
# 'eval'로 "rot13"을 구현.
# (옮긴이: rot13 이란 각 알파벳을 13번 로테이트(그래서 이름이 rot13) 시키는
# 간단한 암호화입니다.)
# "rot13.sh" 예제와 비교해 보세요.

setvar_rot_13()              # "rot13" 스크램블(암호화)
{
  local varname=$1 varvalue=$2
  eval $varname='$(echo "$varvalue" | tr a-z n-za-m)'
}


setvar_rot_13 var "foobar"   # "foobar" 를 rot13 시키면,
echo $var                    # sbbone

echo $var | tr a-z n-za-m    # foobar
                             # 원래 변수로 되돌림.

# Stephane Chazelas 제공.

exit 0
```

> ## 경고
>
> **eval** 명령어는 아주 위험한 상황을 가져올 수 있기 때문에 합당한 다른 방법이 있다면 이것을 쓰지 않는 것이 좋습니다. **`eval $COMMANDS`**는 **rm -rf *** 처럼 유쾌하지 않은 값을 갖고 있을 수도 있는 *COMMANDS*을 실행 시킵니다. 모르는 사람이 작성한 잘 모르는 코드에 대해서 **eval**를 실행 시키는 것은 아주 위험합니다.

set

set 명령어는 내부 스크립트 변수값을 바꿔줍니다. 스크립트의 행동을 결정하는 옵션 플래그를 키거나 끄는 역할에 쓰이기도 하고 특정 명령어의 결과(**`` set `command` ``**)를 위치 매개변수로 리셋 시켜서 스크립트가 그 명령어의 결과를 필드별로 파싱할 수 있게 해줍니다.

예 **11-10.** 위치 매개변수와 **set** 쓰기

```
#!/bin/bash

# "set-test" 스크립트

# 3개의 인자를 줘서 실행시키세요.
# 예를 들면, "./set-test one two three".
```

```
echo
echo "set \`uname -a\` 하기 전의 위치 매개변수:"
echo "첫번째 명령어줄 인자 = $1"
echo "두번째 명령어줄 인자 = $2"
echo "세번째 명령어줄 인자 = $3"

echo

set `uname -a` # `uname -a` 의 출력을 위치 매개변수로 세트

echo "set \`uname -a\` 한 다음의 위치 매개변수:"
# $1, $2, $3... 이 `uname -a` 의 결과로 다시 초기화됩니다.
echo "'uname -a' 의 첫번째 필드 = $1"
echo "'uname -a' 의 두번째 필드 = $2"
echo "'uname -a' 의 세번째 필드 = $3"
echo

exit 0
```

예 10-2 참고.

unset

unset 명령어는 쉘 변수를 효과적으로 널(null)로 세트를 해서 그 변수를 지우는 효과를 가져옵니다. 이 명령어는 위치 매개변수에 대해서 동작하지 않는 것에 주의하세요.

```
bash$ unset PATH

bash$ echo $PATH


bash$
```

예 **11-11.** 변수를 "언셋"**(unset)** 하기

```
#!/bin/bash
# unset.sh: 변수를 언셋하기.

variable=hello                       # 초기화.
echo "variable = $variable"

unset variable                       # 언셋.
                                     # variable=   라고 하는 것과 동일
echo "(unset) variable = $variable"  # $variable 는 널.

exit 0
```

export

export 명령어는 현재 실행중인 스크립트나 쉘의 모든 자식 프로세스가 변수를 사용할 수 있게 해 줍니다. 불행하게도 스크립트나 쉘을 부른 부모 프로세스에게 변수를 다시 **export** 할 방법은 없습니다. **export**는 시스템 구동(startup) 파일에서 환경 변수를 초기화 하고 그 다음에 생성될 사용자 프로세스들이 그 변수에 접근할 수 있게 해주는 아주 중요한 용도로 쓰입니다.

예 **11-12. export**를 써서**,** 내장된 **awk** 스크립트에 변수를 전달하기

```
#!/bin/bash

# "column totaler" 스크립트(col-totaler.sh)의 또 다른 버전.
# 대상 파일의 주어진 지정된 컬럼을 모두 더해줌.
# 여기서는 스크립트 변수를 'awk'에게 전달하기 위해서 환경(environment)을 사용합
니다.

ARGS=2
E_WRONGARGS=65

if [ $# -ne "$ARGS" ] # 원하는 수 만큼의 명령어줄 인자가 넘어왔는지 확인.
then
   echo "사용법: `basename $0` filename column-number"
   exit $E_WRONGARGS
fi

filename=$1
column_number=$2

#===== 여기까지는 원래 스크립트와 똑같습니다 =====#

export column_number
# column_number 를 환경으로 export 해서 awk 스크립트에서 다시 꺼내갈수 있
게 함.


# awk 스크립트 시작.
# ------------------------------------------------
awk '{ total += $ENVIRON["column_number"]
}
END { print total }' $filename
# ------------------------------------------------
# awk script 끝.


# Stephane Chazelas 제공.

exit 0
```

작은 정보: **export var1=xxx**라고 해서 초기화와 export를 한번에 할 수도 있습니다.

declare
typeset

declare 와 typeset 명령어는 변수의 특성을 지정하거나 제한해 줍니다.

readonly

declare -r과 같은 역할을 하는 명령어로서, 어떤 변수를 읽기 전용으로 만들어 주는 것인데, 결국 상수로 쓰겠다는 것입니다. 이런 변수값을 바꾸려고 한다면 에러 메세지를 만나게 됩니다. C 언어의 **const** 형지정자와 비슷한 것으로 보면 됩니다.

getopts

이것은 아주 강력한 도구로서 명령어 줄에서 스크립트로 넘어온 인자를 파싱해 줍니다. C 프로그래머들에게 익숙한 **getopt** 라이브러리 함수의 bash 버전입니다. 스크립트로 넘어오는 여러개의 옵션 [2] 과 그 해당 인자들을 처리해줍니다(예를 들면, **`scriptname -abc -e /usr/local`**).

getopts는 내부적으로 두 개의 변수를 사용합니다. `$OPTIND`(*OPTion INDex*)는 인자 포인터이고 `$OPTARG`(*OPTion ARGument*)는 옵션에 딸려 넘어오는 해당 인자(선택적)입니다. 선언 태그의 옵션 이름뒤에 콜론이 있으면 해당 인자가 있다는 뜻입니다.

getopts는 보통 while 루프와 같이 써서 옵션과 인자를 한 번에 하나씩 처리하고 `$OPTIND` 변수값을 하나씩 줄여서 그 다음을 처리하게 합니다.

참고:

1. 명령어 줄에서 인자앞에는 빼기(-)나 더하기(+)를 적어 줘야 하는데 이 접두사가 있어야 **getopts**가 명령어 줄 인자를 옵션으로 인식할 수 있습니다. 실제로는, -나 +가 빠져 있는 첫번째 인자를 만나면 바로 종료하게 됩니다.
2. **getops**는 표준 **while** 루프와 약간 다른 형태로 조건 대괄호가 빠져 있는 형태입니다.
3. 예전의 **getopt** 대신 **getopts**가 새롭게 쓰입니다.

```
while getopts ":abcde:fg" Option
# 초기 선언.
# a, b, c, d, e, f, g 옵션(플래그)만 지원.
# 'e' 뒤의 :로 'e' 옵션에는 인자가 있어야 된다는 것을 나타냄
do
  case $Option in
    a ) # 'a'일 경우 할 일.
    b ) # 'b'일 경우 할 일.
    ...
    e)  # 'e'일 경우 할 일, $OPTARG 로 'e' 뒤에 따라오는 해당 인자를 처리.
    ...
```

```
    g ) # 'g'일 경우 할 일.
  esac
done
shift $(($OPTIND - 1))
# 인자 포인터를 다음으로 이동.

# 보이는 것처럼 그렇게 복잡하지는 않습니다. <씨익>.
```

예 **11-13. getopts**로 스크립트로 넘어온 옵션과 인자 읽기

```
#!/bin/bash

# 'getopts' 는 스크립트로 넘어온 명령어줄 인자를 처리해 줍니다.
# 인자들은 "옵션"(플래그)과 해당 인자로 파싱됩니다.

# 이렇게 실행시켜 보세요.
# 'scriptname -mn'
# 'scriptname -oq qOption' (qOption 은 아무런 문자열이면 됩니다.)
# 'scriptname -qXXX -r'
#
# 'scriptname -qr'    - 원하던 결과가 안 나오는데, "r"이 "q" 옵션의 추가 인자
로 처리되기 때문입니다.
# 'scriptname -q -r'  - 위와 똑같이 원치 않는 결과.
# 어떤 옵션에 추가 인자("flag:")가 필요하다고 설정이 되면
# 명령어줄에서 자기 바로 다음에 나오는 것을 무조건 자신의 인자로 받아들입니다.

NO_ARGS=0
OPTERROR=65

if [ $# -eq "$NO_ARGS" ]  # 인자 없이 불렸군요.
then
  echo "사용법: `basename $0` options (-mnopqrs)"
  exit $OPTERROR          # 인자가 주어지지 않았다면 사용법을 알려주고 종료.
fi
# 사용법: scriptname -options
# 주의: 대쉬(-)가 필요합니다.


while getopts ":mnopq:rs" Option
do
  case $Option in
    m     ) echo "1번 시나리오: option -m-";;
    n | o ) echo "2번 시나리오: option -$Option-";;
    p     ) echo "3번 시나리오: option -p-";;
    q     ) echo "4번 시나리오: option -q-, \"$OPTARG\"를 줘서";;
    # 'q' 옵션은 추가 인자가 있어야 하는데 없다면 디폴트로 처리됩니다.
    r | s ) echo "5번 시나리오: option -$Option-"";;
```

```
    *     ) echo "구현되지 않은 옵션이 선택됐습니다.";;   # 디폴트
  esac
done

shift $(($OPTIND - 1))
# 인자 포인터를 감소시켜서 다음 인자를 가르키게 합니다.

exit 0
```

스크립트 동작

source
. (점(dot) 명령어)

이 명령어가 명령어 줄에서 불린다면 해당 스크립트를 실행 시킵니다. 스크립트에서 **`source file-name`**이라고 불린다면 `file-name`을 읽어 들일 것입니다. C/C++의 **`#include`** 지시자와 같은 역할을 합니다. 여러개의 스크립트가 공통으로 쓰이는 데이타 파일이나 함수 라이브러리를 써야 할 경우같은 상황에서 유용합니다.

예 **11-14.** 데이타 파일 "포함하기"

```
#!/bin/bash

. data-file    # 데이타 파일 로드.
# "source data-file"이라고 하는 것과 같지만, 이게 좀 더 이식성 있는 방법입니
다.

# "data-file"은 'basename'으로 참조되기 때문에 현재 디렉토리에 꼭 있어야 됩니
다.

# 이제 그 파일에 들어 있는 몇개의 데이타를 참조해 보겠습니다.

echo "variable1 (data-file 에 있는 변수) = $variable1"
echo "variable3 (data-file 에 있는 변수) = $variable3"

let "sum = $variable2 + $variable4"
echo "variable2 + variable4 (data-file 에 있는 변수) = $sum"
echo "message1 (data-file 에 있는 변수) 은 \"$message1\""
# 주의:                                    이스케이프된 쿼우트

print_message 이것은 data-file 의 message-print 함수가 보여주는 메세지입니
다.


exit 0
```

위의 예 11-14에서 `data-file`은 같은 디렉토리에 있어야 합니다.

```
# error free
```

```
# (옮긴이:
# 제일 윗줄에 한글이 들어가 있으면 file data-file 이라고 했을 때
# data-file: International language text
# 라고 나오네요. 이것 때문인지 예제에서 이 파일을 . 하면 에러가 납니다.
# 맨 위의 error free 는 이 에러를 피하기 위한 것입니다.)
#
#
# 이것은 스크립트에 의해 읽히는 데이타 파일입니다.
# 이런 종류의 파일은 보통 변수나 함수등을 담고 있습니다.
# 쉘 스크립트에 의해 'source'나 '.'로 적재될 수 있습니다.

# 변수를 초기화 합시다.

variable1=22
variable2=474
variable3=5
variable4=97

message1="안녕, 잘 지냈어?"
message2="이젠 됐어. 안녕."

print_message ()
{
# 넘어온 어떤 메세지도 다 에코시킴.

  if [ -z "$1" ]
  then
    return 1
    # 인자가 없으면 에러.
  fi

  echo

  until [ -z "$1" ]
  do
    # 함수로 넘어온 인자를 하나씩 모두 처리.
    echo -n "$1"
    # 라인 피드를 없애면서 한 번에 하나씩 에코.
    echo -n " "
    # 낱말 사이에 빈 칸을 집어 넣음.
    shift
    # 다음.
  done

  echo

  return 0
}
```

exit

스크립트를 무조건 끝냄. **exit**는 정수값을 인자로 받아서 쉘에게 스크립트의 종료 상태를 알려줄 수도 있습니다. 아주 간단한 스크립트가 아니라면 스크립트의 마지막에 `exit 0`처럼 성공적인 실행을 알려 주는 것은 아주 좋은 습관입니다.

> 참고: 만약에 **exit**가 인자 없이 쓰인다면 그 스크립트의 종료 상태는 **exit**를 제외하고 가장 마지막에 실행된 명령어의 종료 상태로 됩니다.

exec

이 쉘 내부 명령은 현재의 프로세스를 주어진 명령어로 대치시킵니다. 보통은 쉘이 어떤 명령어를 만나면 그 명령어를 실행하기 위해서 자식 프로세스를 포크[3] 시킵니다. 하지만 **exec** 내장 명령은 포크를 하지 않고 exec된 그 명령어로 쉘 자체를 대치시킵니다. 그렇기 때문에 스크립트에서 이 명령어가 쓰이면 **exec**된 명령어가 종료할 때 스크립트가 강제로 종료됩니다. 이런 이유로, **exec**을 스크립트에서 쓰려면 아마도 제일 마지막 명령어로 써야 할 겁니다.

exec는 또한 파일 디스크립터를 재할당 할 때도 쓰입니다. **`exec <zzz-file`**은 표준입력을 `zzz-file`으로 바꿔줍니다(예 16-1 참고).

예 11-15. exec 효과

```
#!/bin/bash

exec echo "\"$0\" 를 종료합니다."   # 스크립트에서 종료.

# 다음 줄은 절대 실행되지 않습니다.

echo "여기는 절대 에코되지 않습니다."

exit 0  # 역시, 여기서 종료되지도 않고요.
```

> 참고: find 명령어의 `-exec` 옵션은 **exec** 쉘 내장 명령과 *다릅니다*.

shopt

이 명령어는 쉘이 실행중에 옵션을 바꿀 수 있게 해 줍니다(예 24-1과 예 24-2 참고). 이는 종종 Bash 시스템 구동 파일(startup files)에서 쓰이는 데 일반 스크립트에서도 쓰일 수 있습니다. 이 명령어는 bash 버전 2나 그 다음 버전부터 쓸 수 있습니다..

```
shopt -s cdspell
# 'cd'를 쓸 때 디렉토리 이름이 약간 틀린 것 정도는 자동으로 고쳐줍니다.
```

명령어

true

단순히 성공적(0)인 종료 상태를 리턴하는 명령어.

```
# 무한 루프
while true   # ":" 의 별칭(alias)
do
   operation-1
   operation-2
   ...
   operation-n
   # 루프를 빠져 나갈 방법이 필요.
done
```

false

단순히 실패한 종료 상태를 리턴하는 명령어.

```
# 널 루프
while false
do
   # 이 부분은 실행되지 않음.
   operation-1
   operation-2
   ...
   operation-n
   # 아무 일도 안 일어납니다!
done
```

type [cmd]

외부 명령어인 which와 비슷하게 주어진 “cmd”의 완전한 경로명을 보여 줍니다. 하지만, **which**와는 다르게 **type**는 bash 내장 명령입니다. 유용한 옵션인 `-a`를 주면 주어진 “cmd”가 *키워드*인지 *내장 명령*인지를 알려주고 똑같은 이름의 시스템 명령어가 있다면 그 위치도 알려줍니다.

```
bash$ type '['
[ is a shell builtin
bash$ type -a '['
[ is a shell builtin
[ is /usr/bin/[
```

hash [cmds]

주어진 명령어의 경로명을 쉘 해쉬 테이블에 저장해서 그 명령어가 다시 불릴 때 `$PATH`에서 찾지 않도록 해줍니다. **hash**를 인자 없이 쓰면 자신이 해쉬하고 있는 목록을 보여줍니다. `-r` 옵션은 해쉬 테이블을 초기화 합니다.

help

help 명령어는 쉘 내장 명령에 대한 간략한 사용법을 알려줍니다. whatis와 동일하지만 내장 명령에 대해서 쓰인다는 점이 다릅니다.

```
bash$ help exit
exit: exit [n]
    Exit the shell with a status of N.  If N is omitted, the exit status
    is that of the last command executed.
```

11.1. 작업 제어 명령어

다음에 나올 몇몇 작업 제어 명령어들은 "작업 구분자"(job identifier)를 인자로 받습니다. 이 장 맨 끝에 있는 테이블을 참고하세요.

jobs

백그라운드로 돌고 있는 작업들을 작업 번호와 함께 보여줍니다만, **ps**만큼 쓸 만하진 않습니다.

> 참고**:** 작업(job)과 프로세스에 대해서 혼동하기가 쉬운데, **kill**, **disown**, **wait**같은 내장 명령은 작업 번호나 프로세스 번호, 둘 다, 인자로 받아 들입니다. 하지만 **fg**, **bg**, **jobs**는 오직 작업 번호만 받아 들입니다.
>
> ```
> bash$ sleep 100 &
> [1] 1384
>
> bash $ jobs
> [1]+  Running                 sleep 100 &
> ```
>
> "1" 은 작업 번호(현재 쉘에 의해 관리되는 작업들)이고, "1384"는 프로세스 번호(시스템에 의해 관리되는 프로세스들)입니다. 이 작업이나 프로세스를 죽이려면 **kill %1**라고 하거나 **kill 1384**라고 하면 됩니다.
>
> **Thanks, S.C.**

disown

쉘의 활성화 작업 테이블에서 특정 작업을 지워버립니다.

fg
bg

fg 명령어는 백그라운드에서 실행중인 작업을 포그라운드로 돌려 놓습니다. **bg** 명령어는 중지되어 있던 작업을 백그라운드에서 다시 돌게 합니다. **fg**나 **bg**에 작업 번호가 주어지지 않는다면 현재 돌고 있는 작업에 대해서 동작합니다.

wait

백그라운드로 실행중인 모든 작업이나 옵션으로 주어진 특정 작업 번호나 프로세스 아이디가 끝날 때까지 스크립트 실행을 중단 시킵니다. 자신이 기다리고 있던 명령어의 종료 상태를 리턴합니다.

백그라운드 작업이 끝나기 전에 스크립트가 끝나는 것(무서운 고아 프로세스를 만들어 낼 수 있습니다)을 피하기 위해 **wait** 명령어를 쓸 수도 있습니다.

예 **11-16.** 작업을 계속 해 나가기 전에 프로세스가 끝나길 기다리기

```
#!/bin/bash

ROOT_UID=0   # $UID 가 0인 사용자만이 루트 권한을 갖습니다.
E_NOTROOT=65
E_NOPARAMS=66

if [ "$UID" -ne "$ROOT_UID" ]
then
  echo "이 스크립트는 루트만 실행시킬 수 있습니다."
  # "잘 시간이 지난 것 같은데 꺼지지 그래."
  exit $E_NOTROOT
fi

if [ -z "$1" ]
then
  echo "사용법: `basename $0` find-string"
  exit $E_NOPARAMS
fi


echo "'locate' 데이타베이스 업데이트중..."
echo "시간이 걸릴 수 있습니다."
updatedb /usr &      # 루트로 실행시켜야 됩니다.

wait
# 'updatedb' 가 끝나기 전까지 이 다음 부분을 실행 시키지 않습니다.
# 아마도 업데이트된 최신 데이타베이스에
# 여러분의 찾는 파일이 반영돼 있기를 바랄테니까요.

locate $1

# wait 명령어를 쓰면,
# 'updatedb' 가 돌고 있는데 스크립트가 종료되는 최악의 시나리오에서
# 고아 프로세스를 만드는 것을 막아줍니다.

exit 0
```

wait %1 이나 **wait $PPID** 처럼 **wait**에 작업 ID를 인자로 줄 수도 있습니다. 작업 ID 테이블을 참고하세요.

> 작은 정보: 스크립트에서 어떤 명령어를 백그라운드로 돌리려고 & 를 붙여서 실행시키면 스크립트가 **ENTER**를 칠 때까지 멈춰 있을 수 있습니다. 명령어가 표준출력으로 쓰기 때문에 생기는 문제처럼 보이는데, 이것 때문에 아주 성가실 수 있습니다.
>
> ```
> #!/bin/bash
> # test.sh
>
> ls -l &
> echo "Done."
> ```
>
> ```
> bash$ ./test.sh
> Done.
> [bozo@localhost test-scripts]$ total 1
> -rwxr-xr-x    1 bozo     bozo           34 Oct 11 15:09 test.sh
> ```
>
> 백그라운드로 돌릴 명령어 다음에 **wait**를 두면 문제를 해결할 수 있어 보입니다.
>
> ```
> #!/bin/bash
> # test.sh
>
> ls -l &
> echo "Done."
> wait
> ```
>
> ```
> bash$ ./test.sh
> Done.
> [bozo@localhost test-scripts]$ total 1
> -rwxr-xr-x    1 bozo     bozo           34 Oct 11 15:09 test.sh
> ```
>
> 명령어의 출력을 아무 파일이나 `/dev/null`로 재지향하는 것도 이 문제를 해결할 수 있습니다.

suspend

Control-Z 와 비슷한 효과를 갖고 있지만 이것은 쉘을 **suspend** 시킵니다(쉘의 부모 프로세스는 적당한 시간이 지나면 실행을 재개할 것입니다).

logout

로그인 쉘을 빠져나가기. 옵션으로 종료 상태를 지정해 줄 수 있습니다.

times

명령어를 실행하는 데 쓰인 시스템 시간에 대한 통계 정보를 다음 형식으로 보여줍니다.

```
0m0.020s 0m0.020s
```

제한된 범위의 값만을 보여주기 때문에 쉘 스크립트를 프로파일하거나 벤치마크하는데 쓰이지 않습니다.

kill

적당한 종료 시그널을 주어 프로세스를 강제로 끝내게 합니다(예 13-4 참고).

> 참고: **`kill -l`**이라고 하면 가능한 모든 시그널을 볼 수 있습니다. **`kill -9`**은 간단한 **kill**만으로 죽기를 거부하는 지독한 프로세스를 "확실히 죽여줍니다". 가끔은, **`kill -15`**로도 될 때가 있습니다. 부모가 종료된 "좀비 프로세스"는 죽일 수 없지만(이미 죽은 것을 죽일 수는 없습니다), 보통은 **init**이 이런 상태를 금방 청소해 줄 것입니다.

command

command 명령어 지시어는 "명령어"에 대한 별칭이나 함수 찾기를 하지 않습니다.

> 참고: 이는 스크립트의 명령어 처리에 영향을 주는 세 가지 지시어중 하나이고, 나머지 두 개의 지시어는 builtin과 enable입니다.

builtin

builtin BUILTIN_COMMAND라고 치면 "BUILTIN_COMMAND"를 쉘 내장 명령어로 실행 시키면서 잠시 같은 이름을 가진 함수와 외부 시스템 명령어에 대한 기능을 꺼버립니다.

enable

이 명령어는 쉘 내장 명령을 키거나 끄는 역할을 합니다. 예를 들어, **enable -n kill**이라고 하면 쉘 내장 명령인 kill의 기능을 끄고 다음부터 나오는 모든 **kill**에 대해서는 `/bin/kill`을 실행 시킵니다.

`-a` 옵션을 주면 모든 쉘 내장 명령에 대해 각각이 사용 가능한 지를 보여줍니다. `-f filename` 옵션은 **enable** 명령어가 미리 컴파일된 오브젝트 파일에서 공유 라이브러리(DLL) 모듈을 내장 명령으로 로드하도록 해 줍니다. [4].

autoload

이 명령어는 **ksh** autoloader를 Bash로 포팅한 것입니다. **autoload**를 함수 선언시에 같이 쓰면, 그 함수가 처음 불릴 경우에 외부 파일에서 로드합니다. [5] 이렇게 하면 시스템 리소스를 절약해 줍니다.

조심할 것은 **autoload**는 Bash 설치시 기본으로 깔리지 않기 때문에 **enable -f**(위를 참조)로 로드를 해 주어야 합니다.

표 **11-1. 작업 ID(Job Identifiers)**

표시	뜻
`%N`	[N] 작업 숫자
`%S`	**S** 문자로 시작하는 작업을 부름(명령어줄)

표시	뜻
`%?S`	**S** 문자를 포함하는 작업을 부름(명령어줄)
`%%`	“현재” 작업(포그라운드에서 중지된 최근 작업이나 백그라운드로 막 돌기 시작한 작업)
`%+`	“현재” 작업(포그라운드에서 중지된 최근 작업이나 백그라운드로 막 돌기 시작한 작업)
`%-`	마지막 작업
`$!`	최근 백그라운드 프로세스

주석

1. 이렇게 하는 이유는 성능상의 문제(보통은 **fork**를 해야 하는 외부 명령어보다 더 빠르게 실행)이거나 특정 내부 명령의 경우에 쉘 내부 변수로 직접 접근할 필요가 있기 때문입니다.
2. 옵션은 플래그처럼 동작하는 인자로서 스크립트의 행동을 키거나 꺼 줍니다. 특정 옵션과 관련이 있는 인자는 그 옵션(플래그)이 나타내는 행동을 키거나 끄게 합니다.
3. 명령어나 쉘 자신이 어떤 작업을 수행하기 위해 새로운 하위 프로세스를 초기화(혹은 **spawn**)하는 것을 포크(fork)라고 합니다. 이 때, 새롭게 생긴 프로세스를 “자식” 프로세스라고 하고, 그 자식 프로세스를 포크한 프로세스를 “부모” 프로세스라고 합니다. 자식 프로세스가 자신에게 주어진 일을 하는 동안 부모 프로세스도 계속 자신의 일을 해 나갑니다.
4. 로드할 수 있는 내장 명령의 C 소스는 보통 `/usr/share/doc/bash-?.??/functions` 디렉토리에서 찾을 수 있습니다.

   enable 명령어의 `-f` 옵션은 모든 시스템에서 가능하지 않습니다. 주의하세요.
5. typeset -fu 라고 해서 **autoload**와 같은 효과를 가져올 수 있습니다.

12장. 외부 필터, 프로그램, 명령어

유닉스 표준 명령어들은 쉘 스크립트를 좀 더 융통성 있게 만들어 줍니다. 스크립트의 막강한 힘은 시스템 명령어들과 간단한 프로그래밍 구조로 된 쉘 지시자들에서 나옵니다.

12.1. 기본 명령어

명령어 목록

ls

파일 "목록"을 보여주는 기본 명령어로서, 너무 간단해서 과소평가되기 쉬운 명령어지만 그렇지 않습니다. 예를 들면, 하위 디렉토리까지 표시해주는 `-R` 옵션을 쓰면 디렉토리 구조를 트리같은 형태로 보여줍니다. 흥미로운 다른 옵션들로는, 파일 크기대로 정렬해주는 `-S`, 변경 시간으로 정렬해주는 `-t`, 파일의 inode를 보여주는 `-i` 가 있습니다(예 12-3 참고).

예 **12-1. CDR** 디스크를 구울 때 **ls**로 목차 만들기

```
#!/bin/bash

SPEED=2     # 여러분의 CDR 속도에 맞게 수정하세요.
IMAGEFILE=cdimage.iso
CONTENTSFILE=contents
DEFAULTDIR=/opt

# CDR 을 자동으로 구워주는 스크립트.

# Joerg Schilling 의 "cdrecord" 패키지 사용.
# (http://www.fokus.gmd.de/nthp/employees/schilling/cdrecord.html)

# 일반 사용자가 이 스크립트를 실행시키려면 cdrecord 에
# suid 가 걸려 있어야 합니다(루트로 chmod u+s /usr/bin/cdrecord).

if [ -z "$1" ]
then
  IMAGE_DIRECTORY=$DEFAULTDIR
  # 명령어줄에서 지정 안 했을 경우의 기본 디렉토리.
else
    IMAGE_DIRECTORY=$1
fi

ls -lRF $IMAGE_DIRECTORY > $IMAGE_DIRECTORY/$CONTENTSFILE
# "l" 옵션은 "자세한"(long) 파일 목록을 보여줍니다.
# "R" 옵션은 하위 디렉토리까지 보여줍니다.
```

```
# "F" 옵션은 파일 타입을 표시해 줍니다(예를 들면, 디렉토리인 경우에는 이름 끝
에 / 를 붙임.
echo "목차를 만드는 중."

mkisofs -r -o $IMAGFILE $IMAGE_DIRECTORY
echo "ISO9660 파일 시스템 이미지($IMAGEFILE) 만드는 중."

cdrecord -v -isosize speed=$SPEED dev=0,0 $IMAGEFILE
echo "디스크를 굽고 있습니다."
echo "시간이 오래 걸리니까 기다리기 바랍니다."

exit 0
```

cat
tac

cat은 *concatenate* (연속으로 잇다)에서 따온 말로써 파일을 표준출력으로 뿌려 줍니다. 보통 재지향(>이나 >>)과 같이 써 여러 파일을 한 파일로 만들어 줍니다.

```
cat filename cat file.1 file.2 file.3 > file.123
```

cat에 `-n` 옵션을 주면 대상 파일의 모든 줄 앞에 줄번호를 붙여 줍니다. `-b` 옵션은 빈 줄이 아닌 줄에만 줄번호를 붙여 줍니다. `-v` 옵션은 출력할 수 없는 문자들을 ^ 표기법으로 보여줍니다.

예 12-21와 예 12-17를 참고하세요.

tac은 *cat*을 거꾸로 쓴 것인데, 파일 끝에서부터 거꾸로 보여줍니다.

rev

파일의 각 줄을 거꾸로 뒤집어 표준출력으로 내보냅니다. **tac**과 다른 점은 줄의 순서는 그대로 살리고 각 줄을 줄 끝에서 줄 처음으로 뒤집는다는 것입니다.

```
bash$ cat file1.txt
This is line 1.
 This is line 2.


bash$ tac file1.txt
This is line 2.
 This is line 1.


bash$ rev file1.txt
.1 enil si sihT
 .2 enil si sihT
```

cp

파일 복사(copy) 명령어입니다. **`cp file1 file2`**은 `file1`을 `file2`로 복사하는데 `file2`가 이미 존재한다면 덮어 씁니다(예 12-5 참고).

> 작은 정보: 특별히 유용한 옵션은 아카이브 옵션, `-a`(디렉토리 트리 전체를 복사할 때)와 재귀 옵션인 `-r`과 `-R`입니다.

mv

파일 이동(move) 명령어입니다. **cp**와 **rm**을 합친 것과 동일합니다. 여러개의 파일을 한 디렉토리로 옮기거나 디렉토리 이름을 바꿀 때 쓰일 수도 있습니다. 스크립트에서 **mv**가 쓰이는 예제는 예 9-13와 예 A-3을 참고하세요.

rm

파일을 삭제(remove). `-f` 옵션은 읽기 전용 파일도 강제로 지워줍니다.

> 주의
>
> 재귀 옵션인 `-r`과 같이 쓰이면 디렉토리 트리 이하 모든 파일을 지워버립니다.

rmdir

디렉토리를 삭제. 삭제할 디렉토리안에는 안 보이는 "도트 파일"[1] 을 포함해서 아무 파일도 없어야 이 명령어가 제대로 실행됩니다.

mkdir

디렉토리 만들기. 새로운 디렉토리를 생성해 줍니다. **`mkdir -p project/programs/December`** 라고 하면 주어진 디렉토리를 만들어 주는데 *-p* 옵션을 주면 자동으로 필요한 부모 디렉토리가 만들어 집니다.

chmod

파일의 속성을 변경(예 11-8 참고).

```
chmod +x filename
# 모든 사용자가 "filename" 을 실행시킬수 있게 함.

chmod u+s filename
# "filename" 소유권에 "suid" 비트를 걸어줍니다.
# 이렇게 하면 일반 사용자가 "filename"을 실행시켰을 때 그 파일 사용자의
# 권한을 갖습니다.
# (이것은 쉘 스크립트에는 해당되지 않습니다.)


chmod 644 filename
```

```
# "filename"에 대해 파일 소유자는 읽기/쓰기 가능하고 다른 사람들은 읽기만 하게
# 해 줍니다(8진 모드).

chmod 1777 directory-name
# 모든 사람들이 디렉토리안에서 읽고 쓰고 실행할 수 있게 하고 또한
# "스티키 비트"를 설정합니다.
# 이렇게 하면 오직 그 디렉토리의 소유자나 그 디렉토리에 들어 있는 파일의
# 소유자나 루트만(당연하죠?) 특정 파일을 지울 수 있습니다.
```

chattr

파일 속성을 바꿔줍니다. 위에서 설명한 **chmod**와 똑같은 역할을 하지만 문법이 약간 다르고 오직 **ext2** 파일 시스템에 대해서만 동작합니다.

ln

이미 존재하는 파일에 대한 링크를 만들어 줍니다. 심볼릭이나 “소프트” 링크 플래그를 나타내는 `-s` 옵션을 많이 씁니다. 참조되는 원래의 파일이 하나 이상의 이름을 갖도록 해주기 때문에 별칭보다는 더 좋은 방법입니다(예 5-6 참고).

`ln -s oldfile newfile` 라고 하면 이미 존재하는 `oldfile`에 대한 새로운 링크 파일인 `newfile`을 만들어 줍니다.

12.2. 복잡한 명령어

명령어 목록

find

-exec *COMMAND* \;

find가 찾아낸 각각의 파일에 대해 *COMMAND*를 실행합니다. *COMMAND*는 \;으로 끝나야 합니다(**find**로 넘어가는 명령어의 끝을 나타내는 ;를 쉘이 해석하지 않도록 이스케이프 시켜야 합니다). *COMMAND*에 {}이 포함되어 있으면 선택된 파일을 완전한 경로명으로 바꿔 줍니다.

```
bash$ find ~/ -name '*.txt'
/home/bozo/.kde/share/apps/karm/karmdata.txt
/home/bozo/misc/irmeyc.txt
/home/bozo/test-scripts/1.txt
```

```
find /home/bozo/projects -mtime 1
# /home/bozo/projects 디렉토리안에 있는 파일중에서
# 하루 전에 변경된 파일들을 모두 보여 줍니다.
```

```
find /etc -exec grep '[0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-9]*[.][0-
9][0-9]*' {} \;

# /etc 디렉토리에 들어 있는 파일들에 포함된
# 모든 IP 주소(xxx.xxx.xxx.xxx)를 찾아줍니다.
# IP 가 아닌 것도 나오는데 이것들을 어떻게 걸러낼 수 있을까요?

# 이건 어때요?

find /etc -type f -exec cat '{}' \; | tr -c '.[:digit:]' '\n' \
 | grep '^[^.][^.]*\.[^.][^.]*\.[^.][^.]*\.[^.][^.]*$'

# Thanks, S.C.
```

> **경고**
>
> **find**에서 쓰이는 `-exec` 옵션을 쉘 내장 명령인 exec과 헷갈리면 안 됩니다.

예 12-2. Badname, 파일 이름에 일반적이지 않은 문자나 공백 문자를 포함하는 파일을 지우기.

```
#!/bin/bash

# 현재 디렉토리에 들어 있는 파일중, 이름에 정상적이지 않은 글자가 포함된 파일을 지
우기

for filename in *
do
badname=`echo "$filename" | sed -
n /[\+\{\;\"\\\=\?~\(\)\<\>\&\*\|\$]/p`
# 이런 고약한 글자를 포함하는 파일들: + { ; " \ = ? ~ ( ) < > & * | $
rm $badname 2>/dev/null    # 에러 메세지는 무시.
done

# 다음은 모든 종류의 공백 문자를 포함하는 파일들을 처리하겠습니다.
find . -name "* *" -exec rm -f {} \;
# "find"가 찾은 파일이름이 "{}"로 바뀝니다.
# '\'를 써서 ';'가 명령어 끝을 나타낸다는 원래의 의미로 해석되게 합니다.

exit 0

#---------------------------------------------------------------------
# 다음의 명령어들은 위에서 "exit"를 했기 때문에 실행되지 않습니다.

# 위 스크립트의 다른 방법:
find . -name '*[+{;"\\=?~()<>&*|$ ]*' -exec rm -f '{}' \;
```

```
exit 0
# (Thanks, S.C.)
```

예 **12-3. inode** 로 파일을 지우기

```
#!/bin/bash
# idelete.sh: inode 로 파일을 지우기.

#  이 스크립트는 파일이름이 ? 나 - 처럼 부적절한 문자로 시작될 때 유용합니다.

ARGCOUNT=1                      # 인자로 파일이름이 필요함.
E_WRONGARGS=70
E_FILE_NOT_EXIST=71
E_CHANGED_MIND=72

if [ $# -ne "$ARGCOUNT" ]
then
  echo "사용법: `basename $0` filename"
  exit $E_WRONGARGS
fi

if [ ! -e "$1" ]
then
  echo "\""$1"\" 는 존재하지 않는 파일입니다."
  exit $E_FILE_NOT_EXIST
fi

inum=`ls -i | grep "$1" | awk '{print $1}'`
# inum = 파일의 inode (index node)
# 모든 파일은 자신의 물리적 주소 정보를 담고 있는 inode를 갖고 있습니다.

echo; echo -n "\"$1\" 를 진짜로 지우실 겁니까(y/n)? "
read answer
case "$answer" in
[nN]) echo "마음을 바꿨군요, 그렇죠?"
      exit $E_CHANGED_MIND
      ;;
*)    echo "\"$1\" 를 지우는 중.";;
esac

find . -inum $inum -exec rm {} \;
echo ""\"$1"\" 가 지워졌습니다!"

exit 0
```

스크립트에서 **find**를 사용하는 다음 예제들을 참고하세요. 예 12-22, 예 4-3, 예 10-8. 이 복잡하고 강력한 명령어에 대해서 더 알고 싶으면 맨페이지를 살펴보세요.

xargs

명령어에 인자들을 필터링해서 넘겨 주고 그 명령어를 다시 조합하는 데 쓸 수도 있습니다. **xargs**는 입력을 필터용으로 작게 조각내서 명령어가 처리하게 해 줍니다. 역따옴표의 강력한 대용품이라고 생각하면 됩니다. 역따옴표를 써서 too many arguments란 에러가 났을 때, **xargs**로 바꿔 쓰면 성공할 수도 있습니다. 보통은 표준 입력이나 파이프에서 데이터를 읽어 들이지만 파일의 출력에서도 읽을 수 있습니다.

xargs의 기본 명령어는 echo입니다.

`ls | xargs -p -l gzip` 은 현재 디렉토리의 모든 파일에 대해 한번에 한 파일씩 물어보면서 gzips으로 묶어줍니다.

작은 정보: 재밌는 옵션중 하나인 `-n` *XX*을 쓰면 넘길 인자의 갯수를 *XX*로 제한합니다.

`ls | xargs -n 8 echo` 는 현재 디렉토리의 파일들을 한 줄에 8 개씩 끊어서 보여줍니다.

작은 정보: 다른 유용한 옵션으로 **find -print0**나 **grep -lZ**와 함께 쓰는 `-0`이 있습니다. 이 옵션은 공백 문자나 따옴표가 들어간 인자를 처리할 수 있게 해줍니다.

`find / -type f -print0 | xargs -0 grep -liwZ GUI | xargs -0 rm -f`

`grep -rliwZ GUI / | xargs -0 rm -f`

위의 두 가지 모두 "GUI"를 포함하고 있는 어떤 파일도 지워 줍니다. **(Thanks, S.C.)**

예 **12-4.** 시스템 로그 모니터링용 **xargs** 로그 파일

```
#!/bin/bash

# 현재 디렉토리에 /var/log/messages 의 끝 부분을 포함하는 로그 파일을 만들기

# 주의: 일반 사용자도 이 스크립트를 쓰게 하려면
#       루트로 chmod 644 /var/log/messages 라고 해서
#       누구나 /var/log/messages 를 읽을 수 있게 해야 됩니다.

LINES=5

( date; uname -a ) >>logfile
# 시간과 머신 이름
echo ------------------------------------------------------------
---- >>logfile
tail -$LINES /var/log/messages | xargs |  fmt -s >>logfile
echo >>logfile
echo >>logfile

exit 0
```

예 **12-5. copydir. xargs**로 현재 디렉토리를 다른 곳으로 복사하기

```
#!/bin/bash

# 현재 디렉토리의 모든 파일을
# 명령어줄에서 지정한 디렉토리로 복사하기(verbose).

if [ -z "$1" ]   # 인자가 없다면 종료.
then
  echo "사용법: `basename $0` directory-to-copy-to"
  exit 65
fi

ls . | xargs -i -t cp ./{} $1
# 어떤 파일이름에도 "공백문자"가 들어 있지 않다면
#    cp * $1
# 이라고 해도 동일합니다.

exit 0
```

`expr`

다목적 표현식 평가 명령어: 주어진 연산에 따라 자동으로 계산하거나 평가합니다. 이때, 인자는 빈칸으로 분리되어야 합니다. 산술, 비교, 문자열, 논리 연산등이 가능합니다.

`expr 3 + 5`

`8` 리턴

`expr 5 % 3`

`2` 리턴

`` y=`expr $y + 1` ``

변수를 증가. **`let y=y+1`** 이나 **`y=$(($y+1))`** 과 같음. 이것은 산술 확장 예제입니다.

`` z=`expr substr $string $position $length` ``

$string의 $position에서부터 $length만큼의 문자열조각(substring)을 추출해 냄.

예 **12-6. expr** 쓰기

```
#!/bin/bash

# 'expr'의 몇가지 사용법 보여주기
# ===============================

echo
```

```
# 산술 연산자
# ---- ------

echo "산술 연산자"
echo
a=`expr 5 + 3`
echo "5 + 3 = $a"

a=`expr $a + 1`
echo
echo "a + 1 = $a"
echo "(변수 증가)"

a=`expr 5 % 3`
# 나머지(modulo)
echo
echo "5 mod 3 = $a"

echo
echo

# 논리 연산자
# ---- ------

#  참이면 1, 거짓이면 0 리턴.
#  Bash 관례와 반대입니다.

echo "논리 연산자"
echo

x=24
y=25
b=`expr $x = $y`         # 같은 값인지 확인하기.
echo "b = $b"            # 0  ( $x -ne $y )
echo

a=3
b=`expr $a \> 10`
echo 'b=`expr $a \> 10`, 즉...'
echo "If a > 10, b = 0 (거짓)"
echo "b = $b"            # 0  ( 3 ! -gt 10 )
echo

b=`expr $a \< 10`
echo "If a < 10, b = 1 (참)"
echo "b = $b"            # 1  ( 3 -lt 10 )
echo
# 연산자를 이스케이프 시킨것에 주의.
```

```
b=`expr $a \<= 3`
echo "If a <= 3, b = 1 (참)"
echo "b = $b"            # 1  ( 3 -le 3 )
# "\>=" 연산자도 있어요(크거나 같음).


echo
echo

# 비교 연산자
# ---- ------

echo "비교 연산자"
echo
a=zipper
echo "a 는 $a"
if [ `expr $a = snap` ]
# 변수 'a'를 강제로 재평가(re-evaluation)
then
   echo "a 는 zipper 가 아님"
fi

echo
echo



# 문자열 연산자
# ------ ------

echo "문자열 연산자"
echo

a=1234zipper43231
echo "\"$a\" 를 가지고 조작해 보겠습니다."

# length: 문자열 길이
b=`expr length $a`
echo "\"$a\" 의 길이는 $b."

# index: 문자열에서 문자열조각(substring)이 일치하는 첫번째 문자의 위치
b=`expr index $a 23`
echo "\"$a\" 에서 \"2\" 가 첫번째로 나오는 위치는 \"$b\" 입니다."

# substr: 문자열조각 추출, 추출할 시작 위치와 추출할 길이 지정
b=`expr substr $a 2 6`
echo "시작위치는 2이고 길이가 6인 \"$a\" 의 문자열조각은 \"$b\" 입니다."
```

```
# 'match' 연산은 정규표현식을 쓰는 'grep'과 비슷합니다.
b=`expr match "$a" '[0-9]*'`
echo \"$a\" 에서 앞쪽에 나오는 숫자의 갯수는 $b 입니다.
b=`expr match "$a" '\([0-9]*\)'`        # 중괄호가 이스케이프된 것에 주의하
세요.
echo "\"$a\" 에서 앞쪽에 나오는 숫자는 \"$b\" 입니다."

echo

exit 0
```

중요**: match** 대신 : 연산자를 쓸 수 있습니다. 예를 들면, 위의 예제에서 **``b=`expr $a : [0-9]*` ``** 은 **``b=`expr match $a [0-9]*` ``** 과 완전히 동일합니다.

```
#!/bin/bash

echo
echo "\"expr $string :\" 를 쓰는 문자열 연산"
echo "--------------------------------------"
echo

a=1234zipper43231
echo "\"`expr "$a" : '\(.*\)'`\" 를 가지고 문자열 연산을 합니다."
#       이스케이프된 소괄호.
#       정규표현식 파싱.

echo "\"$a\" 의 길이는 `expr "$a" : '.*'` 입니다."   # 문자열 길이

echo "\"$a\" 에서 앞쪽에 나오는 숫자의 갯수는 `expr "$a" : '[0-9]*'` 입니다."

echo "\"$a\" 에서 앞쪽에 나오는 숫자는 `expr "$a" : '\([0-9]*\)'` 입니다."

echo

exit 0
```

펄과 sed가 더 뛰어난 문자열 파싱 능력을 가지고 있기 때문에 스크립트에서 **Perl**이나 **sed**의 간단한 "서브루틴"을 쓰는 것이 **expr**을 쓰는 것 보다 더 좋은 방법입니다.

문자열 연산에 대한 더 자세한 사항은 9.2절을 참고하세요.

12.3. 시간/날짜 명령어

명령어 목록

date

간단하게 **date**라고만 치면 날짜와 시간을 표준 출력으로 보여줍니다. 이 명령어에서 진짜 재밌는 부분은 형식화와 파싱 옵션입니다.

예 **12-7. date** 쓰기

```
#!/bin/bash
# 'date' 명령어 연습

echo "올해가 시작한 뒤로 지금까지 `date +%j` 일이 지났습니다."
# 날짜를 형식화 하려면 포매터 앞에 '+'를 써야 됩니다.
# %j 는 오늘이 연중 몇 번째 날인가를 알려줍니다.

echo "1970/01/01 이후로 지금까지 `date +%s` 초가 지났습니다."
#  %s 는 "UNIX 에폭(epoch)"이 시작한 뒤로 현재까지 몇 초가 지났는지를 알려줍니
다.
#+ 이걸 도대체 어디다 써 먹죠?

prefix=temp
suffix=`eval date +%s`  # 'date'의 "+%s" 옵션은 GNU 전용 옵션입니다.
filename=$prefix.$suffix
echo $filename
#  "유일한" 임시 파일 이름으로 $$ 를 쓰는 것 보다 더 훌륭합니다.

# 더 많은 형식화 옵션을 보려면 'date' 맨 페이지를 읽어 보세요.

exit 0
```

zdump

주어진 타임 존에 해당하는 시간을 에코.

```
bash$ zdump EST
EST  Tue Sep 18 22:09:22 2001 EST
```

time

명령어 실행 시간에 대한 아주 자세한 통계들을 보여줍니다.

`time ls -l /`의 출력은 다음과 같습니다.

```
0.00user 0.01system 0:00.05elapsed 16%CPU (0avgtext+0avg
data 0maxresident)k
 0inputs+0outputs (149major+27minor)pagefaults 0swaps
```

time과 아주 비슷한 바로 앞 장의 times 명령어도 참고하세요.

참고: **time**은 Bash 2.0 버전 이후로 쉘 예약어가 됐기 때문에 파이프라인에서 쓰이면 약간 다른 동작을 보입니다.

touch

파일에 대한 접근/수정 시각을 현재 시각이나 특정한 시각으로 바꿔 주는 유틸리티지만 새 파일을 만들 때 쓸 수도 있습니다. `zzz`란 파일이 없다고 가정하고 **`touch zzz`** 라고 하면 크기가 0인 `zzz`을 새로 만들어 줍니다. 이런 식으로 시간 정보를 갖는 빈 파일을 만들어서 프로젝트의 변경 날짜를 추적하는데 쓰는 등의 사용법이 가능합니다.

touch 명령어는 **`: >> newfile`** 과 동일한 동작을 합니다(보통 파일에 대해서).

at

작업 제어 명령어인 **at**은 주어진 명령어들을 특정 시간에 수행합니다. 이 명령어는 표면상으로는 crond과 닮아 있지만 여러 명령어들을 단지 한 번만 수행하려고 할 때 주로 쓰입니다.

`at 2pm January 15` 라고 하면 그 시간에 실행시킬 명령어들을 물어보는데 이 명령어는 실용적인 목적때문에 쉘 스크립트와 호환되는 명령어면 다 됩니다. 스크립트에서 한 줄에 이것을 타이핑할 것이기 때문입니다. 입력은 Ctl-D 로 끝냅니다.

`-f` 옵션을 쓰거나 입력 재지향(<)을 써서 파일에 저장되어 있는 명령어들을 읽어 들일 수 있습니다. 이 파일은 쉘 스크립트가 될 수도 있지만 당연히 사용자 입력이 필요 없어야 합니다. 실행할 파일에 run-parts 명령어를 넣어 두면 다른 종류의 여러 스크립트들을 돌릴 수 있기 때문에 아주 멋지게 쓸 수 있습니다.

```
bash$ at 2:30 am Friday < at-jobs.list
job 2 at 2000-10-27 02:30
```

batch

batch도 **at**과 비슷한 작업 제어 명령어지만 다른 점은 시스템 부하가 `0.8` 이하인 경우에만 명령어들을 실행합니다. **at**처럼 `-f` 옵션을 써서 실행할 명령어를 파일에서 읽을 수 있습니다.

cal

깔끔한 형태의 월별 달력을 표준출력으로 출력합니다. 현재를 비롯해 아주 먼 과거나 미래도 보여 줍니다.

sleep

쉘의 wait 루프와 같습니다. 주어진 초단위 시간 동안 아무일도 안 하면서 멈추어 있습니다. 타이밍, 혹은 백그라운드로 돌면서 특정한 사건이 일어나는지 주기적으로 확인하는 등의 경우에 유용하게 쓰일 수 있습니다(예 30-5 참고).

```
sleep 3
```

```
# 3 초 대기.
```

참고**: sleep** 명령어의 단위는 기본적으로 초를 쓰지만 분이나 시간, 일 단위도 지정해 줄 수 있습니다.

```
sleep 3 h
# 3 시간 대기!
```

usleep

마이크로슬립(**Microsleep**)("u"는 그리스 문자 "mu"로 읽던가 마이크로(micro) 접두사 그대로 읽으면 됩니다). 이 명령어는 위에서 설명한 **sleep**과 하는 일이 같지만 마이크로초 동안 "잠들어" 있습니다. 아주 짧은 시간 동안의 타이밍에 쓰이거나 굉장히 잦은 간격으로 진행되는 프로세스를 폴링할 때 쓰일 수 있습니다.

```
usleep 30
# 30 마이크로초 대기.
```

> 경고
>
> **usleep**은 특별히 정확한 타이밍을 제공하지 않기 때문에 아주 중요한 타이밍 루프에는 안 맞습니다.

hwclock
clock

hwclock는 시스템의 하드웨어 클럭을 읽거나 조절해 줍니다. 몇몇 옵션은 루트 권한이 필요합니다. `/etc/rc.d/rc.sysinit` 시스템 구동 파일은 부팅시에 **hwclock**을 써서 하드웨어 클럭으로 시스템 시간을 맞춰줍니다.

clock 명령어는 **hwclock**과 동의어입니다.

12.4. 텍스트 처리 명령어

명령어 목록

sort

주로 파이프에서 필터로 쓰여 파일을 정렬할 때 쓰입니다. 다양한 키나 문자 위치에 따라 텍스트 스트림이나 파일 전체를 정렬할 수 있습니다. `-m` 옵션을 쓰면 이미 정렬된 파일을 합쳐줍니다. **info page**에서 많은 기능과 옵션들을 볼 수 있습니다. 예 10-8과 예 10-9를 참고하세요.

tsort

위상 정렬(Topological sort) 명령어로서, 공백문자로 구분되는 문자열의 쌍을 읽어 패턴에 따라 정렬.

diff
patch

diff: 유연한 파일 비교 유틸리티. 대상 파일들을 줄 단위로 차례 차례 비교해 줍니다. 예를 들어 낱말 사전을 비교하는 어플리케이션에서 sort와 **uniq**를 써서 파일을 필터링하고 파이프를 통해 **diff**로 넘겨주는 상황등에서 유용하게 쓰일 수 있습니다. **`diff file-1 file-2`** 는 두 파일에서 서로 다른 줄이 있으면 그 줄이 속해 있는 파일을 캐럿과 함께 보여줍니다.

`--side-by-side` 옵션을 주면 비교할 파일을 서로 구분된 컬럼에 두고 줄 단위로 비교하면서 일치하지 않는 줄을 표시해 줍니다.

diff의 다양한 프론트엔드(frontend)로는 **spiff**, **wdiff**, **xdiff**, **mgdiff**가 있습니다.

> 작은 정보**: diff** 는 비교할 두 파일이 똑같으면 종료 상태 0을 리턴하고 다르면 1을 리턴합니다. 이것 때문에 **diff**를 쉘 스크립트의 테스트문에서 쓸 수 있습니다(아래 참조).

diff는 보통 **patch**에 쓸 다른 파일을 만들어 내는데 쓰입니다. `-e` 옵션은 **diff**가 자신의 출력을 **ed**나 **ex** 스크립트에서 쓸 수 있는 파일로 만들어 내게 합니다.

patch: 유연한 버전 관리 유틸리티. **patch**는 **diff**가 만들어낸 다른 파일이 주어지면 패키지의 이전 버전을 새로운 버전으로 업그레이드 해 줄 수 있습니다. 이렇게 하면 새롭게 발표된 패키지 전체를 배포할 필요 없이 비교적 작은 "diff" 파일만 배포하면 되므로 아주 편리합니다. 따라서 커널 "패치"는 리눅스 커널의 빈번한 배포시 더 선호됩니다.

```
patch -p1 <patch-file
# 'patch-file'에 나열된 모든 변경 사항을
# 그 파일 내부에서 참조하고 있는 파일에 대해 적용 해 줍니다.
# 이렇게 하면 패키지의 새로운 버전으로 업그레이드가 됩니다.
```

커널 패치 하기:

```
cd /usr/src
gzip -cd patchXX.gz | patch -p0
# 'patch'로 커널 소스를 업그레이드 하기.
# 익명의 저자(알란 콕스?)가 쓴 리눅스 커널 문서 "README"에서 인용.
```

> 참고**: diff**는 디렉토리 트리 전체도 비교할 수 있습니다(실제로 존재하는 파일에 대해서).
>
> ```
> bash$ diff ~/notes1 ~/notes2
> Only in /home/bozo/notes1: file02
> Only in /home/bozo/notes1: file03
> Only in /home/bozo/notes2: file04
> ```

작은 정보: **gzip**으로 묶인 파일을 비교하려면 **zdiff** 를 쓰세요.

diff3

diff의 확장 버전으로 동시에 세 개의 파일을 비교해 줍니다. 성공시에는 0을 리턴하지만 불행하게도 비교 결과에 대한 정보를 얻을 수는 없습니다.

```
bash$ diff3 file-1 file-2 file-3
====
 1:1c
   This is line 1 of "file-1".
 2:1c
   This is line 1 of "file-2".
 3:1c
   This is line 1 of "file-3"
```

sdiff

두 개의 파일을 한 파일로 합치기 위해서 비교하거나 편집해 줍니다. 이 명령어는 사용자와 대화(interactive) 해야 하는 특성이 있어서 스크립트에서 쓰기는 어렵습니다.

cmp

cmp 명령어는 위에서 설명한 **diff**의 간단한 버전입니다. **diff**가 두 파일간의 차이점에 대해서 보고해 주는 반면, **cmp**는 단지 두 파일간에 서로 다른 부분만을 보여줍니다.

참고: **cmp**도 **diff**처럼 두 파일이 똑같다면 종료 상태 0을 리턴하고 다르다면 1을 리턴합니다. 따라서 쉘 스크립트의 테스트문에서 **cmp**를 쓸 수 있습니다.

예 **12-8.** 스크립트에서 두 파일을 비교하기 위해 **cmp** 쓰기**.**

```
#!/bin/bash

ARGS=2  # 인자가 두 개 필요합니다.
E_BADARGS=65

if [ $# -ne "$ARGS" ]
then
  echo "사용법: `basename $0` file1 file2"
  exit $E_BADARGS
fi


cmp $1 $2 > /dev/null  # /dev/null 은 "cmp"의 출력을 안 보이게 해줍니다.
# 'diff'에서도 동작합니다. 즉, diff $1 $2 > /dev/null
```

```
if [ $? -eq 0 ]         # "cmp"의 종료 상태를 확인.
then
  echo "\"$1\" 은 \"$2\" 와 똑같은 파일입니다."
else
  echo "\"$1\" 은 \"$2\" 와 다른 파일입니다."
fi

exit 0
```

작은 정보: **gzip**으로 묶인 파일에는 **zcmp**를 쓰세요.

comm

다목적 파일 비교 유틸리티. 제대로 된 결과를 얻으려면 파일 내용이 정렬돼 있어야 합니다.

comm *-options first-file second-file*

`comm file-1 file-2` 출력은 세 칸으로 이루어 집니다:

- column 1 = `file-1`에 유일한 줄
- column 2 = `file-2`에 유일한 줄
- column 3 = 두 파일 양쪽에 공통으로 나타나는 줄

다음 옵션은 하나 이상의 출력 칸을 제거해 줍니다.

- `-1` 은 1번 칸을 제거
- `-2` 는 2번 칸을 제거
- `-3` 은 3번 칸을 제거
- `-12` 는 1번과 2번 칸을 제거, 등등.

uniq

이 필터는 정렬된 파일에서 중복된 줄을 제거합니다. 보통 파이프에서 sort와 같이 쓰입니다.

```
cat list-1 list-2 list-3 | sort | uniq > final.list
# list 파일들을 붙이고,
# 정렬한 다음,
# 중복된 줄을 제거한 후,
# 마지막으로 결과를 출력 파일로 저장.
```

유용한 옵션인 `-c`을 쓰면 입력 파일의 각 줄 앞에 중복된 수를 표시해 줍니다.

```
bash$ cat testfile
```

```
이 줄은 한 번만 나옵니다.
이 줄은 두 번 나옵니다.
이 줄은 두 번 나옵니다.
이 줄은 세 번 나옵니다.
이 줄은 세 번 나옵니다.
이 줄은 세 번 나옵니다.


bash$ uniq -c testfile
      1 이 줄은 한 번만 나옵니다.
      2 이 줄은 두 번 나옵니다.
      3 이 줄은 세 번 나옵니다.


bash$ sort testfile | uniq -c | sort -nr
      3 이 줄은 세 번 나옵니다.
      2 이 줄은 두 번 나옵니다.
      1 이 줄은 한 번만 나옵니다.
```

`sort INPUTFILE | uniq -c | sort -nr` 이라고 하면 `INPUTFILE` 파일의 발생 빈도 목록을 만들어 줍니다(**sort** 명령어의 `-nr`은 숫자를 거꾸로 정렬). 이 템플릿은 로그 파일이나 사전 목록, 구문 분석이 필요한 어떤 것에도 쓰일 수 있습니다.

예 **12-9.** 낱말 빈도수 분석

```
#!/bin/bash
# wf.sh: 조잡한 텍스트 파일의 낱말 빈도 분석.


# 명령어줄에서의 입력 파일을 확인용.
ARGS=1
E_BADARGS=65
E_NOFILE=66

if [ $# -ne $ARGS ]  # 필요한 인자가 스크립트로 맞게 넘어왔는지?
then
  echo "사용법: `basename $0` filename"
  exit $E_BADARGS
fi

if [ -f "$1" ]       # 파일이 존재하는지 확인.
then
  file_name=$1
else
  echo "\"$1\" 는 없는 파일입니다."
  exit $E_NOFILE
fi
```

```
######################################################
# 메인
sed -e ’s/\.//g’  -e ’s/ /\
/g’ "$1" | tr ’A-Z’ ’a-z’ | sort | uniq -c | sort -nr
#                           ========================
#                            발생 빈도

#  점(period)을 걸러내고 낱말 사이의 빈 칸을 라인피드로 바꾼 다음
#+ 모든 문자를 소문자로 변환한 뒤,
#+ 각 낱말의 발생 빈도를 맨 앞에 두고 숫자대로 정렬.
######################################################

# 독자들을 위한 연습문제:
# 1) ’sed’ 명령어가 콤마같은 다른 구둣점도 걸러내도록 해 보세요.
# 2) 여러개의 빈 칸과 다른 공백문자도 처리하도록 고쳐 보세요.
# 3) 다른 정렬용 키를 추가해서 동일한 발생 빈도를 갖는 낱말에 대해서
#+   알파벳 순으로 정렬되도록 해 보세요.

exit 0
```

```
bash$ cat testfile
이 줄은 한 번만 나옵니다.
이 줄은 두 번 나옵니다.
이 줄은 두 번 나옵니다.
이 줄은 세 번 나옵니다.
이 줄은 세 번 나옵니다.
이 줄은 세 번 나옵니다.


bash$ ./wf.sh testfile
     6 이
     6 나옵니다
     6 줄은
     5 번
     3 세
     2 두
     1 한
     1 번만
```

expand
unexpand

탭을 빈 칸으로 만들어 주는 필터로서 파이프에서 주로 쓰입니다.

unexpand 필터는 빈 칸을 탭으로 바꿔주는 필터로 **expand**와 완전히 반대로 동작합니다.

cut

파일에서 필드를 뽑아 내는 툴. awk의 **`print $N`** 명령어 셋과 비슷하지만 약간의 제한 사항을 갖고 있습니다. 스크립트에서 **awk**보다 더 간단하게 쓸 수 있습니다. 특별히 중요한 옵션으로 `-d`(구분자)와 `-f`(필드 지시자)가 있습니다.

cut으로 마운트 된 파일 시스템 목록 알아내기: 쓰기:

```
cat /etc/mtab | cut -d ' ' -f1,2
```

cut으로 OS와 커널 버전 알아내기:

```
uname -a | cut -d" " -f1,3,11,12
```

cut 으로 이 메일 폴더에서 메세지 헤더를 뽑아내기:

```
bash$ grep '^Subject:' read-messages | cut -c10-80
Re: Linux suitable for mission-critical apps?
 MAKE MILLIONS WORKING AT HOME!!!
 Spam complaint
 Re: Spam complaint
```

cut으로 파일을 파싱하기:

```
# /etc/passwd 에 들어있는 모든 사용자 목록.

FILENAME=/etc/passwd

for user in $(cut -d: -f1 $FILENAME)
do
  echo $user
done

# Oleg Philon 이 제공해준 예제.
```

`cut -d ' ' -f2,3 filename` 은 **`awk -F'[ ]' '{ print $2, $3 }' filename`** 과 같은 결과를 보여줍니다.

예 12-29 참고.

colrm

칸 제거 필터. 파일에서 여러 칸(글자들 단위)을 지워주는데, 칸 범위를 지정하지 않으면 원래 파일을 표준출력으로 다시 내보냅니다. **`colrm 2 4 <filename`** 이라고 하면 텍스트 파일인 `filename` 각 줄의 두 번째 칸에서 4 번째 칸의 글자를 지웁니다.

주의

파일에 탭이나 출력 할 수 없는 글자가 포함되어 있다면 예상치 못한 동작을 할 수도 있습니다. 이런 경우에는 **colrm** 앞에 expand와 **unexpand**를 파이프로 걸어서 써 보기 바랍니다.

paste

서로 다른 파일들을 여러 단으로 나뉘어진 하나의 파일로 만들어 주는 툴로서, **cut**과 같이 써서 시스템 로그 파일을 만드는데 유용합니다.

join

특수한 목적을 가진 **paste**류의 명령어라고 보면 됩니다. 두 파일을 의미있는 형태로 묶어서 본질적으로는 간단한 관계 데이터베이스를 만들어 주는 강력한 유틸리티입니다.

join 명령어는 정확히 두 파일에 대해서 동작하지만 두 파일 사이에 공통으로 표시된 필드(tagged field)(보통은 숫자 라벨)가 들어 있는 줄에 대해서만 합쳐서 결과를 표준출력에 씁니다. 제대로 동작하려면 두 파일 모두, 표시 필드가 제대로 정렬되어 있어야 합니다.

```
File: 1.data

100 Shoes
200 Laces
300 Socks

File: 2.data

100 $40.00
200 $1.00
300 $2.00

bash$ join 1.data 2.data
File: 1.data 2.data

 100 Shoes $40.00
 200 Laces $1.00
 300 Socks $2.00
```

참고**:** 표시 필드는 결과에서 한 번만 나옵니다.

head

파일 앞부분을 표준출력으로 보여 줍니다(기본적으로 10줄을 보여주지만 따로 지정해 줄 수도 있습니다). **head**는 재밌는 옵션이 몇 개 있습니다.

예 **12-10. 10**자리 랜덤한 숫자 만들기

```
#!/bin/bash
# rnd.sh: 10 자리 랜덤한 숫자 출력

# Script by Stephane Chazelas.

head -c4 /dev/urandom | od -N4 -tu4 | sed -ne '1s/.* //p'


# =================================================================== #

# 분석
# ----

# head:
# -c4 옵션은 첫 번째 4 바이트만 받아 들입니다.

# od:
# -N4 옵션은 출력을 4 바이트로 제한해 줍니다.
# -tu4 옵션은 출력을 unsigned 10진 포맷으로 해줍니다.

#  sed:
#  -n 옵션은 "s" 명령어의 "p" 플래그와 같이 쓰여서
#+ 오직 일치하는 줄만 출력해 줍니다.



# 다음은 이 스크립트의 저자가 'sed'의 동작에 대해서 설명한 내용입니다.

# head -c4 /dev/urandom | od -N4 -tu4 | sed -ne '1s/.* //p'
# ----------------------------------> |

# "sed"로 들어가는 출력을   --------> |
# 0000000 1198195154\n 라고 가정하면

# sed가 0000000 1198195154\n 을 읽으면,
# 뉴라인(\n) 문자를 발견하기 때문에 첫 번째 줄(0000000 1198195154)에
# 대해서 처리할 준비가 됩니다.
# 그 다음에는 자신의 <주소 범위><동작> 을 참고하는데 여기서는

#   주소 범위  동작
#   1          s/.* //p

# 가 됩니다.
```

```
# 줄 번호가 주소 범위안에 들어가기 때문에 동작을 수행합니다:
# 그 줄에서 공백문자로 끝나는 가장 긴 문자열("0000000 ")을
# 아무것도 아닌 것으로(//) 치환시키고, 성공한다면 그 결과를 출력해 줍니다
# (여기에서, "p"는 "s" 명령어에 대한 플래그지 "p" 명령어가 아닙니다).

# sed 는 이제 다음 입력을 받을 준비를 합니다(여기서 주의할 점은,
# 만약에 -n 옵션이 없다면, 그 줄을 한 번 더 출력할 것이라는 것입니다).

# 그 다음에는 나머지 문자들을 읽어 들이는데, 파일의 끝을 발견하게 됩니다.
# 이제 마지막 줄임을 나타내는 '$'가 매겨지는 두 번째 줄을 처리할 준비가 된거죠.
# 그런데 두 번째 줄은 어떤 <범위> 에도 들지 않기 때문에 동작을 멈춥니다.

# 이 동작들을 간단하게 설명해 보면 다음과 같습니다:
# "첫번째 줄만 읽은 다음 제일 오른쪽에 나오는 빈 칸까지 삭제한 다음 출력"

# 이렇게 하면 더 좋을 수도 있습니다:
#           sed -e 's/.* //;q'

# 두 개의 <주소 범위><동작> 으로 쓰일 수도 있습니다.
#           sed -e 's/.* //' -e q):

#   주소 범위                 동작
#   일치하는 줄 없음          s/.* //
#   일치하는 줄 없음          q (quit)

# 이렇게 하면, sed 는 오직 첫 번째 줄만 입력으로 받아 들입니다.
# "-n" 옵션이 없기 때문에 종료하기("q" 명령어) 전에 바뀐 줄을 출력해 줍니다.

# ==================================================================== #

# 위의 한 줄짜리 스크립트는 더 간단하게 쓸 수도 있습니다:
#           head -c4 /dev/urandom| od -An -tu4

exit 0
```

예 12-27 참고.

tail

파일의 마지막 부분을 표준출력으로 보여 줍니다(기본적으로10줄). 보통, 시스템 로그 파일의 변경 사항을 추적할 때 쓰는데, 파일 뒷부분에 계속 덧붙여지는 사항을 볼 수 있게 해 주는 `-f` 옵션을 주면 됩니다.

예 **12-11. tail**로 시스템 로그를 모니터하기

```
#!/bin/bash

filename=sys.log
```

```
cat /dev/null > $filename; echo "깨끗한 파일 생성."
# 존재하지 않는 파일이라면 새로 만들고,
# 길이를 0 으로 만듦.
# : > filename   이라고 해도 됩니다.

tail /var/log/messages > $filename
# 이 명령이 제대로 동작하려면 /var/log/messages 에 아무나 읽을 수 있는
# 퍼미션이 걸려 있어야 됩니다.

echo "$filename 은 시스템 로그의 마지막 부분을 보여줍니다."

exit 0
```

예 12-4, 예 12-27, 예 30-5 참고.

grep

정규 표현식을 쓰는 다목적 파일 검색 도구로서, 원래 예전의 라인 에디터인 **ed**의 명령어나 필터였던 **`g/re/p`**에서 따온 것으로 **global - regular expresstion - print**란 뜻입니다.

grep *`pattern`* [*`file`*...]

대상 파일에서 보통 텍스트이거나 정규 표현식인 *`pattern`*을 찾아 줍니다.

```
bash$ grep '[rst]ystem.$' osinfo.txt
The GPL governs the distribution of the Linux operating system.
```

대상 파일이 주어지지 않는다면 파이프에서 쓰여서 다른 명령어의 표준출력에 대한 필터로 동작합니다.

```
bash$ ps ax | grep clock
765 tty1     S      0:00 xclock
901 pts/1    S      0:00 grep clock
```

`-i` 옵션은 대소문자 구분 없이 찾도록 해줍니다.

`-l` 옵션은 일치하는 줄이 아니라 일치하는 줄이 들어 있는 파일만 보여줍니다.

`-n` 옵션은 일치하는 줄과 그 줄번호를 같이 보여 줍니다.

```
bash$ grep -n Linux osinfo.txt
2:This is a file containing information about Linux.
6:The GPL governs the distribution of the Linux operating system.
```

`-v` (혹은 `--invert-match`) 옵션은 일치하는 패턴을 걸러내 줍니다.

```
grep pattern1 *.txt | grep -v pattern2
```

```
# "*.txt"에서 "pattern2"는 제외하고 "pattern1"을 포함하는 모든 줄.
```

`-c` (`--count`) 옵션은 일치하는 패턴을 보여주지 않고 일치한 횟수만 보여줍니다.

```
grep -c txt *.sgml   # ("*.sgml"에서 "txt"가 나오는 횟수)


#   grep -cz .
#            ^ 점이 있어요.
# "."과 일치하고 0으로 구분된(-z) 아이템 갯수(-c)
# 즉, 최소한 1 글자 이상을 포함하는 아이템.
#
printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -
cz .     # 4
printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -
cz '$'   # 5
printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -
cz '^'   # 5
#
printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -
c '$'    # 9
# 기본적으로 뉴라인 문자(\n)가 아이템을 구분해 줍니다.

# -z 옵션은 GNU "grep"에서만 쓰이는 옵션임에 주의하세요.


# Thanks, S.C.
```

대상 파일을 하나 이상 적어주면 일치하는 파일도 같이 보여 줍니다.

```
bash$ grep Linux osinfo.txt misc.txt
osinfo.txt:This is a file containing information about Linux.
osinfo.txt:The GPL governs the distribution of the Linux operating system.
misc.txt:The Linux operating system is steadily gaining in popularity.
```

> 작은 정보: 오직 하나의 파일에서만 찾으려고 할 때 **grep**이 강제로 파일이름을 보여주게 하고 싶으면 두 번째 파일로 `/dev/null`을 주면 됩니다.
>
> ```
> bash$ grep Linux osinfo.txt /dev/null
> osinfo.txt:This is a file containing information about Linux.
>  osinfo.txt:The GPL governs the distribution of the Linux operating system.
> ```

grep은 일치하는 패턴을 찾으면 종료 상태 0을 리턴하는데 출력을 안 하게 해 주는 `-q` 옵션과 같이 스크립트의 테스트 문에서 쓰면 유용하게 쓸 수 있습니다.

```
SUCCESS=0                      # grep 검색이 성공하면
word=Linux
```

```
filename=data.file

grep -q "$word" "$filename"    # "-q" 옵션은 표준출력으로 아무것도 에코하지
않습니다.

if [ $? -eq $SUCCESS ]
then
  echo "$filename 에서 $word 발견"
else
  echo "$filename 에서 $word 발견 못 함"
fi
```

예 30-5 은 시스템 로그 파일에서 **grep**으로 특정 낱말 패턴을 찾는 것을 보여줍니다.

예 **12-12.** 스크립트에서 “**grep**”을 에뮬레이트 하기

```
#!/bin/bash
# grp.sh: ’grep’을 아주 조잡하게 다시 구현.

E_BADARGS=65

if [ -z "$1" ]    # 인자 확인.
then
  echo "사용법: `basename $0` pattern"
  exit $E_BADARGS
fi

echo

for file in *     # $PWD 의 모든 파일을 탐색.
do
  output=$(sed -n /"$1"/p $file)  # 명령어 치환.

  if [ ! -z "$output" ]           # "$output" 을 쿼우트 안 하면 어떻게 될
까요?
  then
    echo -n "$file: "
    echo $output
  fi              #  sed -ne "/$1/s|^|${file}: |p"  라고 해도 똑같습니다.

  echo
done

echo

exit 0

# 독자용 연습문제:
# ---------------
```

```
# 1) 주어진 파일에서 하나 이상 일치한다면 출력에 뉴라인을 추가해 보세요.
# 2) 여러가지 특징들을 추가해 보세요.
```

참고: **egrep** 은 **grep -E**와 같습니다. 좀 더 유연한 검색 능력을 갖는 확장 정규 표현식 셋을 사용합니다.

fgrep 은 **grep -F**와 같습니다. 문자 그대로의 검색(정규 표현식 안 씀)만 하기 때문에 속도가 약간 빠릅니다.

agrep 은 **grep**이 유사(approximate) 매칭을 할 수 있게 확장해 줍니다. 찾을 문자열과 찾은 문자열은 주어진 숫자 만큼의 문자가 다를 수도 있습니다. 이 유틸리티는 리눅스 배포판에서 기본으로 포함되지 않습니다.

작은 정보: 압축된 파일에서 검색을 하려면 **zgrep**, **zegrep**, **zfgrep**을 쓰세요. 압축 안 된 파일에 대해서도 동작하지만 **grep**, **egrep**, **fgrep** 보다는 약간 느립니다. 압축 파일과 비압축 파일이 섞여 있을 때 사용하면 편리합니다.

bzip으로 압축된 파일에서 검색을 하려면 **bzgrep** 을 쓰세요.

look

look은 **grep**과 비슷하게 동작하지만 정렬된 낱말 목록인 "사전"에 들어 있는 낱말에 대해서만 찾습니다. 따로 지정하지 않으면 `/usr/dict/words` 에 들어 있는 낱말만 찾는데 다른 사전 파일을 지정해 줄 수도 있습니다.

예 **12-13.** 목록에 들어 있는 낱말들의 유효성 확인하기

```
#!/bin/bash
# lookup: 데이타 파일에 들어 있는 모든 낱말들에 대해서 사전 검색을 수행.

file=words.data  # 테스트용 낱말들이 들어 있는 데이타 파일.

echo

while [ "$word" != end ]  # 데이타 파일의 마지막 낱말.
do
  read word      # 루프의 끝에서 재지향을 걸었기 때문에 데이타 파일에서 읽음.
  look $word > /dev/null  # 사전 파일의 결과를 표시하지 않음.
  lookup=$?      # 'look' 명령어의 종료 상태.

  if [ "$lookup" -eq 0 ]
  then
    echo "\"$word\" 는 유효함."
  else
    echo "\"$word\" 는 유효하지 않음."
  fi

done <"$file"    # 표준입력을 $file로 재지향 하기 때문에 $file에서 "읽음".
```

```
echo

exit 0

# ------------------------------------------------------------------
# 위의 "exit" 명령어 때문에 다음 코드는 실행되지 않습니다.


# Stephane Chazelas 가 더 간결한 다음 코드를 제안해 주었습니다:

while read word && [[ $word != end ]]
do if look "$word" > /dev/null
   then echo "\"$word\" 는 유효함."
   else echo "\"$word\" 는 유효하지 않음."
   fi
done <"$file"

exit 0
```

sed
awk

텍스트 파일이나 명령어 출력을 파싱하는데 특히 알맞은 스크립트 언어입니다. 홀로 쓰일 수도 있고 파이프 중간이나 쉘 스크립트에서 쓰일 수도 있습니다.

sed

많은 **ex** 명령어들을 배치 모드에서 쓸 수 있게 해주는 비대화형(non-interactive) "스트림 에디터"입니다. 쉘 스크립트에서 아주 자주 쓰입니다.

awk

프로그램 가능한 파일 분석및 형식화 명령어로서, 구조화된 텍스트 파일의 필드나 컬럼을 뽑아내고 조작하는데 아주 적당하며, 문법은 C와 비슷합니다.

wc

wc는 파일이나 I/O 스트림에 나타나는 "낱말 갯수"(word count)를 알려줍니다:

```
bash $ wc /usr/doc/sed-3.02/README
20     127     838 /usr/doc/sed-3.02/README
[20 lines  127 words  838 characters]
```

wc -w 는 낱말 갯수만 알려줍니다.

wc -l 은 줄 수만 알려줍니다.

wc -c 는 글자 수만 알려줍니다.

wc -L 을 가장 긴 줄의 길이만 알려줍니다.

wc 로 현재 디렉토리에 **.txt** 파일이 몇 개 있는지 알아내기:

```
$ ls *.txt | wc -l
# "*.txt" 중, 파일명에 라인피드가 들어 있지 않는 한, 잘 동작합니다.

# 다른 방법:
#      find . -maxdepth 1 -name \*.txt -print0 | grep -cz .
#      (shopt -s nullglob; set -- *.txt; echo $#)

# Thanks, S.C.
```

wc 로 d 에서 h 사이의 문자로 시작되는 파일들 크기의 전체 합을 구하기.

```
bash$ wc [d-h]* | grep total | awk '{print $3}'
71832
```

wc 로 이 책의 메인 소스 파일에서 “Linux”가 몇 번이나 나오는지 알아보기.

```
bash$ grep Linux abs-book.sgml | wc -l
50
```

예 12-27 와 예 16-5 도 참고.

몇몇 명령어는 자신의 옵션으로 **wc**의 일부 기능을 갖고 있기도 합니다.

```
... | grep foo | wc -l
# 자주 쓰던거죠? 하지만 좀 더 간단하게 쓸 수 있습니다.

... | grep -c foo
# grep의 "-c"(나 "--count") 옵션을 쓰세요.

# Thanks, S.C.
```

tr

문자 변환 필터.

> **경고**
>
> 적절하게 쿼우팅이나 대괄호로 묶어줘야 합니다. 쿼우팅은 **tr** 명령어에서 쓰이는 특수한 문자들이 쉘에 의해 재해석 되지 않도록 막아줍니다. 대괄호는 쉘이 확장을 못 하도록 쿼우트 되어야 합니다.

`tr "A-Z" "*" <filename` 이나 **`tr A-Z \* <filename`** 은 `filename`에 들어 있는 모든 대문자를 별표로 변환해서 표준출력으로 내 보냅니다. 몇몇 시스템에서는 이렇게 하면 안 되고 **`tr A-Z '[**]'`** 이라고 해야 제대로 동작할 수도 있습니다.

`-d` 옵션은 지정된 범위에 해당하는 문자들을 지워 줍니다.

```
tr -d 0-9 <filename
```

```
# "filename" 에 들어 있는 모든 숫자를 지워 줍니다.
```

`--squeeze-repeats`(나 `-s`) 옵션은 연속적인 문자들 중에서 첫번째만 남기고 나머지 문자들은 지워 줍니다. 이 옵션은 과도한 공백 문자를 지울 때 유용합니다.

```
bash$ echo "XXXXX" | tr --squeeze-repeats 'X'
X
```

예 12-14. toupper: 파일 내용을 모두 대문자로 바꿈.

```
#!/bin/bash
# 파일 내용을 모두 대문자로 바꿈.

E_BADARGS=65

if [ -z "$1" ]  # 명령어줄 인자 여부의 표준 확인 작업.
then
  echo "사용법: `basename $0` filename"
  exit $E_BADARGS
fi

tr a-z A-Z <"$1"

# 위와 같지만 POSIX 문자셋 표기법을 쓰는 방법:
#        tr '[:lower:]' '[:upper:]' <"$1"
# Thanks, S.C.

exit 0
```

예 12-15. lowercase: 현재 디렉토리의 모든 파일명을 소문자로 바꿈.

```
#! /bin/bash
#
# 현재 디렉토리의 모든 파일 이름을 다 소문자로 바꿈
#
# 원래 John Dubois의 스크립트를 Chet Ramey가 bash용으로 수정한 것에서
# 영감을 얻어 본 문서의 저자인 Mendel Cooper가 상당히 간단하게 수정했음.


for filename in *                # 현재 디렉토리의 모든 파일을 탐색.
do
   fname=`basename $filename`
   n=`echo $fname | tr A-Z a-z`  # 이름을 소문자로 바꾸고,
   if [ "$fname" != "$n" ]       # 원래 소문자가 아닌 파일만 소문자로 바꿈.
   then
     mv $fname $n
   fi
done
```

```
exit 0


# "exit" 때문에 다음 코드는 실행되지 않습니다.
#---------------------------------------------#
# 이 부분을 실행시키려면 위 스크립트를 모두 지우세요.

# 위 스크립트는 파일이름에 공백이나 뉴라인이 들어 있을 때에는 제대로 동작하지 않습
니다.

# 그래서 Stephane Chazelas 가 다음 스크립트를 제안해 주었습니다.


for filename in *    # "*"는 "/"가 들어 있는 파일은 리턴하지 않기 때문에
                     # basename 을 쓸 필요가 없습니다.
do n=`echo "$filename/" | tr '[:upper:]' '[:lower:]'`
#                             POSIX 문자셋 표기법.
#                    명령어 치환을 해도 꼬리부분(trailing)의 뉴라인이
#                    지워지지 않기 때문에 / 를 붙였습니다.
   # 변수 치환:
   n=${n%/}          # 파일이름에서 아까 붙인 / 를 제거.
   [[ $filename == $n ]] || mv "$filename" "$n"
                     # 파일이름이 이미 소문자인지 확인.
done

exit 0
```

예 12-16. du: 도스용 텍스트 파일을 UNIX용으로 변환.

```
#!/bin/bash
# du.sh: DOS 텍스트 파일을 UNIX 텍스트 파일로 변환.

E_WRONGARGS=65

if [ -z "$1" ]
then
  echo "사용법: `basename $0` filename-to-convert"
  exit $E_WRONGARGS
fi

NEWFILENAME=$1.unx

CR='\015'  # 캐리지 리턴.
# DOS 텍스트 파일에서는 줄 끝에 CR-LF 가 붙습니다.

tr -d $CR < $1 > $NEWFILENAME
# CR 을 지우고 새 파일로 쓰기.
```

```
echo "원래 DOS 텍스트 파일은 \"$1\" 이고,"
echo "변환된 UNIX 텍스트 파일은 \"$NEWFILENAME\" 입니다."

exit 0
```

예 12-17. rot13: 초허접(ultra-weak) 암호화, rot13.

```
#!/bin/bash
# rot13.sh: 아주 유치한 고전적인 rot13 알고리즘

# 사용법: ./rot13.sh filename
# or      ./rot13.sh <filename
# or      ./rot13.sh 라고 한 다음 키보드에서 입력(stdin)

cat "$@" | tr 'a-zA-Z' 'n-za-mN-ZA-M'   # "a"는 "n"이 되
고, "b"는 "o"가 되는등..
# 'cat "$@"' 라고 하면 표준입력이나 파일에서 입력을 받을 수 있게 해 줍니다.

exit 0
```

예 12-18. "Crypto-Quote" 퍼즐 만들기

```
#!/bin/bash
# crypto-quote.sh: 인용문을 암호화

# 이 스크립트는 유명한 인용문을 간단한 1:1 알파벳 치환을 통해 암호화 시켜줍니다.
#  결과는 일요 신문의 Op Ed(옮긴이: Opposite Editorial, 신문의 서명 기사나
#+ 논평이 실리는 페이지)에서 볼 수 있는 "Crypto Quote" 퍼즐과 비슷합니다.


key=ETAOINSHRDLUBCFGJMQPVWZYXK
# "key" 는 단순히 알파벳을 뒤섞어 놓은 것입니다.
# 이 "key"를 바꾸면 암호화가 바뀌게 됩니다.

# 'cat "$@"' 는 표준입력이나 파일에서 입력을 받아 들입니다.
# 표준입력에서 입력을 받는 다면 Control-D 로 끝내면 되고,
# 파일이라면 명령어줄 매개변수로 파일이름을 지정해 주면 됩니다.

cat "$@" | tr "a-z" "A-Z" | tr "A-Z" "$key"
#        |  대문자로      |     암호화
# 소문자, 대문자, 대소문자가 섞인 인용문에 대해서 동작합니다.
# 알파벳이 아닌 문자들은 그대로 둡니다.


# 이 스크립트에 이런 입력을 넣는다면,
# "Nothing so needs reforming as other people's habits."
# --Mark Twain
```

```
#
# 결과는 다음과 같을 겁니다:
# "CFPHRCS QF CIIOQ MINFMBRCS EQ FPHIM GIFGUI'Q HETRPQ."
# --BEML PZERC

# 복호화를 하려면 이렇게 하면 됩니다:
# cat "$@" | tr "$key" "A-Z"


#  이 간단한 암호화는 연필과 종이를 가지고
#+ 평균 12 년 정도 계산하면 금방 깨집니다.

exit 0
```

tr 변종들

tr 유틸리티는 역사적으로 두 개의 변종을 갖고 있습니다. BSD 버전은 대괄호를 쓰지 않지만(**`tr a-z A-Z`**), SysV 버전은 씁니다(**`tr '[a-z]' '[A-Z]'`**). GNU 버전은 BSD 버전을 닮았기 때문에 대괄호 속의 문자 범위를 꼭 쿼우팅 해 줘야 합니다.

fold

입력 줄을 주어진 넓이로 접어주는(wrap) 필터. 특별히 유용한 -s 옵션을 쓰면 낱말 사이의 빈 칸에서 줄을 나눠줍니다(예 12-19와 예 A-2 참고).

fmt

간단한 파일 형식화 명령어로 파이프 중간에 필터로 쓰여 긴 줄을 "접기"(wrap) 위해 쓰입니다.

예 **12-19.** 파일 목록 형식화**.**

```
#!/bin/bash

WIDTH=40                      # 넓이는 40 행.

b=`ls /usr/local/bin`         # 파일 목록을 얻은 다음...

echo $b | fmt -w $WIDTH

#  echo $b | fold - -s -w $WIDTH
#  라고 해도 됩니다.

exit 0
```

예 12-4 참고.

> 작은 정보**: fmt**의 강력한 대체품인 Kamil Toman의 **par** 유틸리티는 http://www.cs.berkeley.edu/ ~amc/Par/ (http://www.cs.berkeley.edu/~amc/Par/) 에서 구할 수 있습니다.

ptx

ptx [targetfile] 명령어는 targetfile의 permuted index(상호 참조 리스트 - cross-reference list)를 출력해 줍니다. 필요하다면 여기서 나온 결과를 나중에 필터링이나 형식화해서 쓸 수도 있습니다.

column

컬럼 형식화 명령어. 목록 형태의 텍스트 출력의 적당한 곳에 탭을 넣어서 "예쁜 출력" 테이블을 얻게 해 주는 필터입니다.

예 **12-20. column** 으로 디렉토리 목록을 형식화 하기

```
#!/bin/bash
# "column" 맨 페이지에 있는 에제를 약간 수정했습니다.


(printf "PERMISSIONS LINKS OWNER GROUP SIZE MONTH DAY HH:MM PROG-
NAME\n" \
; ls -l | sed 1d) | column -t

# "sed 1d" 는 "total        N" 이라고 나오는 첫번째 줄을 지워줍니다.
# 여기서 "N"은 "ls -l"이라고 했을 때의 전체 파일 수를 나타냅니다.

# "column"의 -t 는 표를 예쁘게 찍기(pretty-print) 옵션입니다.

exit 0
```

nl

줄 번호 매기기 필터. **`nl filename`** 이라고 하면 `filename`의 빈 칸을 제외한 각 줄에 연속적인 번호를 붙여서 표준출력으로 보여 줍니다. `filename`을 지정해 주지 않으면 표준입력에 대해서 동작합니다.

예 **12-21. nl:** 자기 자신에게 번호를 붙이는 스크립트**.**

```
#!/bin/bash

# 이 스크립트는 이 스크립트 파일에 줄 번호를 붙여서 표준출력으로 두 번씩 출력해 줍
니다.

# 'nl' 명령어는 빈 줄을 세지 않기 때문에 지금 이 줄을 3 번째 줄로 봅니다.
# 'cat -n' 은 바로 윗줄을 5 번째 줄로 봅니다.
```

```
nl `basename $0`

echo; echo  # 이제 'cat -n'으로 해 볼까요.

cat -n `basename $0`
# 'cat -n' 이 다른 점은 빈 줄에도 번호를 붙인다는 겁니다.
# 주의할 점은 'nl -ba' 라고 하면 'cat -n' 과 똑같이 동작한다는 것입니다.

exit 0
```

pr

출력 형식화 필터. 파일이나 표준출력을 프린터로 찍거나 출판용이나 스크린으로 보기 좋은 섹션 형태로 페이지를 매겨 줍니다. 행과 열을 조작하기, 여러 줄을 합치기, 마진을 넣어 주기, 줄에 번호를 매겨 주기, 페이지 헤더를 붙여 주기, 여러 파일을 합치기등을 비롯한 다양한 옵션이 가능합니다. **pr**은 **nl**, **paste**, **fold**, **column**, **expand**의 기능을 합친 것보다 더한 능력을 가진 명령어입니다.

`pr -o 5 --width=65 fileZZZ | more` 라고 하면 `fileZZZ`를 마진을 5로 하고 전체 폭을 65로 하고 멋있게 페이지를 매겨서 스크린에 뿌려 줍니다.

특별히 유용한 옵션인 `-d`는 한 줄마다 강제로 빈 줄을 넣어줍니다(double-spacing, **sed -G**와 같습니다).

gettext

프로그램의 출력을 다른 언어로 번역해서 보여주는 GNU 지역화(localization) 유틸리티입니다. 처음엔 C 프로그램을 위해서 쓰였지만 쉘 스크립트에서도 쓰입니다. *`info page`*를 참고하기 바랍니다.

iconv

주로 지역화에서 쓰이는 명령어로 파일을 다른 인코딩(문자셋)으로 변환해 줍니다.

recode

이 명령어는 **iconv**의 개선판이라고 보면 됩니다. 파일을 다른 인코딩으로 변환해주는 이 다목적 유틸리티는 표준 리눅스 설치시에는 포함되지 않습니다.

groff
gs
TeX

Groff, TeX, 포스트스크립트(postscript)는 출판용 원고나 형식화된 비디오 디스플레이용 텍스트 마크업 언어(text markup language)들입니다.

맨 페이지가 **groff**을 씁니다(예 A-1 참고). 고스트스크립트(ghostscript, **gs**)는 포스트스크립트 해석기의 GPL 버전입니다. **TeX**는 Donald Knuth의 정교한 조판 시스템입니다. 흔히, 이 마크업 언어들에 넘길 인자나 옵션들을 쉘 스크립트로 처리를 해서 편하게 씁니다.

lex
yacc

구문 분석기(lexical analyzer)인 **lex**는 패턴 매칭을 위한 프로그램을 만들어 냅니다. 리눅스 시스템에서는 이 명령어의 비특허 버전인 **flex**로 바뀌었습니다.

yacc 유틸리티는 스펙셋에 의거한 파서를 만들어 냅니다. 리눅스 시스템에서는 이 명령어의 비특허 버전인 **bison**으로 바뀌었습니다.

12.5. 파일, 아카이브(archive) 명령어

아카이빙

tar

유닉스의 표준 아카이브(archive) 유틸리티. 원래는 *Tape ARchiving* 프로그램에서 왔는데, 이 프로그램은 테입 드라이브부터 보통 파일, 심지어는 표준출력(예 4-3 참고)까지 포함하는 모든 종류의 디바이스에 대해서 모든 종류의 아카이브를 다룰 수 있도록 만들어 졌습니다. GNU tar는 오래전부터 gzip 압축을 다룰 수 있는 옵션이 패치되어 있었는데, **tar czvf archive-name.tar.gz *** 라고 하면 하위 디렉토리를 포함한 모든 파일을 묶어서 압축하라는 뜻입니다(도트파일은 제외).

유용한 **tar** 옵션 몇 가지:

1. `-c` 만들기(새 아카이브)
2. `--delete` 지우기(아카이브에 들어 있는 파일)
3. `-r` 덧붙이기(파일을 아카이브로)
4. `-t` 목록(아카이브 내용)
5. `-u` 아카이브 업데이트
6. `-x` 뽑아내기(아카이브에 들어 있는 파일)
7. `-z` 아카이브를 gzip 으로 압축

> **경고**
>
> **gzip**으로 묶인채 손상된 tar 아카이브는 복구하기가 매우 힘들기 때문에 중요한 파일을 아카이브로 만들때는 여러 군데에 백업을 해 놓기 바랍니다.

shar

쉘 아카이브 유틸리티. 쉘 아카이브 파일은 실제로는 #!/bin/sh 헤더와 아카이브를 풀기 위한 명령어들로 이루어진 쉘 스크립트로써, 압축되지 않은 파일들이 죽 붙어 있는 파일

입니다. shar 아카이브는 아직도 인터넷 뉴스 그룹에서 볼 수 있는데 여기 말고 다른 곳에서는 **tar/gzip** 때문에 거의 안 씁니다. **shar** 아카이브는 **unshar** 명령어로 풀어 줍니다.

ar

주로 바이너리 오브젝트 파일 라이브러리에서 쓰이는 아카이브를 위한 생성, 조작 유틸리티.

cpio

이 특화된 아카이브 복사 명령어(**co**p**y** **i**nput and **o**utput)는 **tar/gzip** 때문에 이제 거의 안 쓰이지만 디렉토리 트리를 옮기려는 경우등의 쓰임새가 아직 남아 있습니다.

예 **12-22. cpio**로 디렉토리 트리 옮기기

```
#!/bin/bash

# cpio 로 디렉토리 트리를 복사하기.

ARGS=2
E_BADARGS=65

if [ $# -ne "$ARGS" ]
then
  echo "사용법: `basename $0` source destination"
  exit $E_BADARGS
fi

source=$1
destination=$2

find "$source" -depth | cpio -admvp "$destination"
# 여기서 쓰인 cpio 옵션이 무슨 뜻인지 알려면 맨 페이지를 읽어 보세요.

exit 0
```

예 **12-23. rpm** 아카이브 풀기

```
#!/bin/bash
# de-rpm.sh: 'rpm' 아카이브 풀기

E_NO_ARGS=65
TEMPFILE=$$.cpio                                 # "유일한" 임시 파일.
                                                 # $$ 는 스크립트의 프로세스 ID.

if [ -z "$1" ]
then
  echo "사용법: `basename $0` filename"
exit $E_NO_ARGS
```

```
fi


rpm2cpio < $1 > $TEMPFILE                  # rpm 아카이브를 cpio 아카이브
로 변환.
cpio --make-directories -F $TEMPFILE -i   # cpio 아카이브 풀기.
rm -f $TEMPFILE                           # cpio 아카이브 지우기.

exit 0
```

압축

gzip

표준 GNU/UNIX 압축 유틸리티로서, 성능이 떨어지고 특허가 걸려 있는 **compress**를 대신합니다. 압축 풀기 명령어는 **gunzip**으로써, **gzip -d**와 같습니다.

zcat 필터는 **gzip**으로 묶인 파일의 압축을 풀어 표준출력으로 내보내기 때문에 파이프의 입력이나 재지향에서 쓰일 수 있습니다. 즉, 실제로는 입축 파일에 대해서 동작하는 **cat**이라고 보면 됩니다(옛날 **compress** 로 묶인 파일도 포함). **zcat**은 **gzip -dc**와 같습니다.

> 경고
>
> 몇몇 상업용 유닉스 시스템에서는 **zcat**이 **uncompress -c**와 동의어로 쓰이지만 **gzip**으로 묶인 파일에 대해서는 동작하지 않습니다.

예 7-6 참고.

bzip2

또 다른 압축 유틸리티로써, 특별히 크기가 큰 파일에 대해서는 **gzip**보다 더 효율적입니다. **bzip2**에 대한 압축 풀기 명령어는 **bunzip2**입니다.

compress
uncompress

상용 유닉스 배포판에서 찾을 수 있는 오래되고 특허가 걸려있는 유틸리티이고, 더 효율적인 **gzip**으로 거의 다 바뀌었습니다. **gunzip**이 **compress**로 묶인 파일들을 풀 수 있지만, 리눅스 배포판들은 호환성을 위해서 **compress**를 닮은 명령어를 포함시킵니다.

> 작은 정보: **znew** 명령어는 **compress**로 압축된 파일을 **gzip**으로 압축된 파일로 변환해 줍니다.

sq

또 다른 압축 유틸리티로써 오직 정렬된 아스키 낱말 목록에 대해서만 동작하는 필터입니다. **sq < input-file > output-file** 처럼 표준 필터를 쓰듯이 쓰면 됩니다. 속도는 빠르지만 gzip만큼 효과적이지는 않습니다. 이 명령어에 해당하는 압축 풀기 필터는 **unsq**이고 사용법은 **sq**와 같습니다.

작은 정보**: sq**의 출력을 **gzip**에 파이프로 걸어서 더 압축 시킬 수도 있습니다.

zip
unzip

도스의 **PKZIP**과 호환되는 크로스 플랫폼 파일 아카이빙및 압축 유틸리티. 인터넷에서 "Zip"으로 묶인 아카이브들이 "타르볼"보다 더 많이 쓰입니다.

파일 정보

file

파일 종류를 구분지어 주는 유틸리티. **`file file-name`** 이라고 치면 `ascii text`이 `data`같은 `file-name`에 대한 스펙을 알려줍니다. 이 명령어는 Linux/UNIX 배포판에 따라 `/usr/share/magic`이나 `/etc/magic`, `/usr/lib/magic`등에서 매직 넘버를 참고해서 파일 스펙을 알려줍니다.

`-f` 옵션을 쓰면 파일이름의 목록이 들어 있는 지정된 파일을 분석하면서 배치 모드로 동작합니다. `-z` 옵션은 대상 파일이 압축된 파일일 경우 강제로 압축이 풀린 상태의 파일 타입을 분석해 줍니다.

```
bash$ file test.tar.gz
test.tar.gz: gzip compressed data, deflated, last modi-
fied: Sun Sep 16 13:34:51 2001, os: Unix

bash file -z test.tar.gz
test.tar.gz: GNU tar archive (gzip compressed data, deflated, last modi-
fied: Sun Sep 16 13:34:51 2001, os: Unix)
```

예 **12-24. C** 소스에서 주석을 제거하기

```
#!/bin/bash
# strip-comment.sh: C 소스에서 주석(/* 주석 */)을 제거해 줍니다.

E_NOARGS=65
E_ARGERROR=66
E_WRONG_FILE_TYPE=67

if [ $# -eq "$E_NOARGS" ]
```

```
then
  echo "사용법: `basename $0` C-program-file" >&2 # 에러 메세지는 표준출력
으로.
  exit $E_ARGERROR
fi

# 파일 타입이 맞는지 확인.
type=`eval file $1 | awk '{ print $2, $3, $4, $5 }'`
# "file $1" 이 파일 타입을 에코해 준 다음...
# awk 가 첫 번째 필드인 파일이름을 지워주고...
# 그 결과가 "type" 변수로 들어갑니다.
correct_type="ASCII C program text"

if [ "$type" != "$correct_type" ]
then
  echo
  echo "이 스크립트는 오직 C 소스 파일에 대해서만 동작합니다."
  echo
  exit $E_WRONG_FILE_TYPE
fi


# 약간은 신비스러워 보이는 sed 스크립트:
#--------
sed '
/^\/\*/d
/.*\/\*/d
' $1
#--------
# sed 의 기본에 대해서 몇 시간만 투자를 하면 이해하기 쉽습니다.


# 주석이 코드와 같은 줄에 있는 경우를 처리하기 위해서는
# 추가적인 sed 스크립트가 필요합니다.
# 약간은 어려울 수도 있지만 독자들을 위해서 연습문제로 남겨 놓습니다.

# 또, 위 코드는 우리가 바라지 않던 결과로서
# "*/" 이나 "/*" 인 줄도 지워버립니다.

exit 0


# ----------------------------------------------------------------------
# 다음의 코드들은 위에서 'exit 0'이라고 했기 때문에 실행되지 않습니다.

# Stephane Chazelas 가 제안한 다른 방법:

usage() {
  echo "사용법: `basename $0` C-program-file" >&2
```

```
  exit 1
}

WEIRD=`echo -n -e ’\377’`   # 혹은 WEIRD=$’\377’
[[ $# -eq 1 ]] || usage
case `file "$1"` in
  *"C program text"*) sed -
e "s%/\*%${WEIRD}%g;s%\*/%${WEIRD}%g" "$1" \
     | tr ’\377\n’ ’\n\377’ \
     | sed -ne ’p;n’ \
     | tr -d ’\n’ | tr ’\377’ ’\n’;;
  *) usage;;
esac

# 이것 역시 다음과 같은 경우에는 오동작을 합니다:
# printf("/*");
# 나
# /*  /* 주석에 주석이 들어감 */
#
# 특별한 경우들(문자열에 들어 있는 주석,
# \"나 \\" 를 포함한 문자열에 들어 있는 주석...)을 모두 처리할 수 있는
# 유일한 방법은 C 파서(아마도 lex 나 yacc?)를 작성하는 것입니다.

exit 0
```

which

which command-xxx 라고 하면 "command-xxx"의 전체 경로명을 알려 줍니다. 시스템에 특정 명령어나 유틸리티가 설치되어 있는지 알아내려고 할 때 유용합니다.

```
$bash which rm
/usr/bin/rm
```

whereis

위의 **which**와 비슷하지만 "command-xxx" 맨 페이지의 전체 경로명도 같이 알려줍니다.

```
$bash whereis rm
rm: /bin/rm /usr/share/man/man1/rm.1.bz2
```

whatis

whatis filexxx 은 `whatis` 데이타베이스에서 "filexxx"를 찾아줍니다. 시스템 명령어와 중요한 설정 파일을 확인하고 싶을 때 유용합니다. 간단한 **man**이라고 생각하면 됩니다.

```
$bash whatis whatis
whatis               (1)  - search the whatis database for complete words
```

예 12-25. `/usr/X11R6/bin` 둘러보기

```
#!/bin/bash

# 도대체 /usr/X11R6/bin 에 들어있는 이상야릇한 실행파일들의 정체가 뭘까요?

DIRECTORY="/usr/X11R6/bin"
# "/bin", "/usr/bin", "/usr/local/bin" 같은 디렉토리에 대해서도 해보세요.

for file in $DIRECTORY/*
do
  whatis `basename $file`   # 실행파일에 대한 정보를 에코.
done

exit 0
# 이 스크립트의 결과를 재지향 하고 싶다면,
# ./what.sh >>whatis.db
# 혹은 표준출력에서 한 번에 한 쪽씩 보려면,
# ./what.sh | less
```

예 10-3 참고.

vdir

자세한 디렉토리 목록을 보여줍니다. ls -l 이라고 하는 것과 비슷합니다.

GNU **fileutils** 에 속하는 명령어입니다.

```
bash$ vdir
total 10
-rw-r--r--    1 bozo  bozo      4034 Jul 18 22:04 data1.xrolo
-rw-r--r--    1 bozo  bozo      4602 May 25 13:58 data1.xrolo.bak
-rw-r--r--    1 bozo  bozo       877 Dec 17  2000 employment.xrolo

bash ls -l
total 10
-rw-r--r--    1 bozo  bozo      4034 Jul 18 22:04 data1.xrolo
-rw-r--r--    1 bozo  bozo      4602 May 25 13:58 data1.xrolo.bak
-rw-r--r--    1 bozo  bozo       877 Dec 17  2000 employment.xrolo
```

shred

파일을 지우기 전에 랜덤 비트 패턴을 여러번 덮어 써서 보안상 안전하게 지워줍니다. 예 12-33 과 동일한 결과를 가져오지만 더 빈틈없고 멋진 방법입니다.

GNU **fileutils** 에 속하는 명령어입니다.

> **경고**
>
> **shred**를 써서 파일을 지운다고 해도 최첨단 복구 기술(advanced forensic technology)을 이용해 그 내용의 일부나 전체를 복구해 내는 것을 막지는 못합니다.

locate
slocate

locate는 자체 데이타베이스에서 파일을 찾아 줍니다. **slocate**는 **locate**(**slocate**로 별칭이 지정 되어 있을)의 보안 강화 버전입니다.

```
 $bash locate hickson
/usr/lib/xephem/catalogs/hickson.edb
```

strings

strings를 쓰면 바이너리나 데이타 파일안에서 사람이 읽을 수 있는(출력 가능한) 문자를 찾을 수 있습니다. 해당 파일에서 출력 가능한 문자들을 순서대로 보여줍니다. 이 명령어를 쓰면 코어 덤프 파일을 간단하게 점검해 볼 수 있고 알 수 없는 그래픽 파일의 타입을 확인해 볼 수 있습니다(`strings image-file | more` 이라고 쳐서 `JFIF`같은 문자를 보여준다면 그 파일은 **jpeg** 파일이라고 보면 됩니다). 스크립트에서는 **strings**의 출력을 grep이나 sed로 파싱해서 쓸 수 있을 것입니다. 예 10-7과 예 10-8을 참고하세요.

유틸리티

basename

파일명에서 경로 정보를 떼어내고 오직 파일 이름만 보여 줍니다. `basename $0` 이라고 하면 스크립트는 자기가 쉘에서 불린 자기 이름을 알 수 있습니다. 스크립트에 필요한 인자가 없이 실행되는 경우에 "사용법" 메세지를 찍을 때 쓸 수 있습니다:

```
echo "사용법: `basename $0` arg1 arg2 ... argn"
```

dirname

파일명에서 **basename**을 떼어내고 오직 경로 정보만 보여줍니다.

> 참고: **basename**과 **dirname**은 어떤 문자열에 대해서도 동작합니다. 이 명령어들에 넘겨줄 인자는 꼭 실제로 존재하는 파일이 아니여도 됩니다(예 A-6 참고).

예 12-26. basename과 dirname

```
#!/bin/bash

a=/home/bozo/daily-journal.txt
```

```
echo "/home/bozo/daily-journal.txt 의 basename = `basename $a`"
echo "/home/bozo/daily-journal.txt 의 dirname = `dirname $a`"
echo
echo "내 홈 디렉토리는 `basename ~/`."              # 그냥 ~ 도 됩니다.
echo "내 홈 디렉토리의 홈은 `dirname ~/`."          # 그냥 ~ 도 됩니다.

exit 0
```

split

한 파일을 작은 조각으로 나눠주는 유틸리티로서 플로피에 백업을 하려고 하거나, 이메일의 첨부 파일로 쓰려고 할 때, 업로드를 하려고 할 때 주로 쓰입니다.

sum
cksum
md5sum

체크썸(checksum)을 생성해 주는 유틸리티입니다. 체크썸이란 파일의 실제 내용에 대해 산술적인 계산을 해 특정한 숫자를 뽑아낸 것입니다. 이를 이용해 파일의 무결성을 확인할 수 있습니다. 보안에 관련된 목적으로 아주 중요한 시스템 파일 내용이 변경되거나 손상됐는지 여부등을 체크썸 목록으로 관리하고 이를 참조하는 스크립트를 만들어 쓸 수도 있습니다. **md5sum**은 이런 보안 어플리케이션에 제일 적합한 명령어입니다.

인코딩과 암호화

uuencode

바이너리 파일을 아스키 문자로 인코드해서 이메일을 보내거나 뉴스 그룹에 포스팅 할 때 제대로 전송될 수 있게 해줍니다.

uudecode

uuencode된 파일을 디코드해서 원래 바이너리 파일로 만들어 줍니다.

예 **12-27.** 인코드된 파일을 **uudecode**하기

```
#!/bin/bash

lines=35        # 헤더용으로 여유있게 35 줄을 잡습니다.

for File in *   # 현재 디렉토리의 모든 파일을 확인...
do
  search1=`head -$lines $File | grep begin | wc -w`
  search2=`tail -$lines $File | grep end | wc -w`
  #  uuencode 된 파일은 앞 부분에 "begin"이란 말이 들어 있고,
  #+ 끝 부분에 "end"란 말이 들어 있습니다.
  if [ "$search1" -gt 0 ]
  then
```

```
    if [ "$search2" -gt 0 ]
    then
      echo "uudecode 중 - $File -"
      uudecode $File
    fi
  fi
done

#  이 스크립트의 인자로 이 스크립트 자체를 주게 되면 오동작을 할 텐데,
#+ 왜냐하면 이 스크립트는 "begin"과 "end"를 모두 포함하고 있기 때문입니다.

# 연습문제:
# 뉴스 그룹 헤더를 확인하도록 수정해 보세요.

exit 0
```

> 작은 정보: 유즈넷 뉴스 그룹에서 받은 아주 긴 메세지를 uudecode할 때 fold -s 명령어를 파이프등에 걸어서 쓰면 아주 유용합니다.

crypt

예전에는 이 명령어가 유닉스의 파일 암호화 표준 유틸리티였습니다.[2] 정치적인 이유때문에 암호화 소프트웨어의 수출을 금지하는 정부의 규제로 인해 많은 유닉스 세계에서 **crypt**가 사라지게 됐고 아직도 거의 대부분의 리눅스 배포판에서 빠져 있습니다. 다행히도 많은 프로그래머들이 **crypt**를 적절히 대신할 프로그램을 많이 만들어 놨고 이들중에는 저자가 직접 만든 cruft (ftp://metalab.unc.edu/pub/Linux/utils/file/cruft-0.2.tar.gz) 도 있습니다(예 A-4 참고).

잡동사니 명령어(Miscellaneous)

make

바이너리 패키지를 빌드및 컴파일해주는 유틸리티. 소스 파일에서 추가 변경 사항이 발생하면 정해진 동작을 하도록 하는데 쓰입니다.

make 명령어는 파일 의존성과 수행할 동작이 들어 있는 `Makefile`을 바탕으로 동작합니다.

install

특별한 목적을 갖고 있는 파일 복사 명령어로서 **cp**와 비슷하지만 복사될 파일의 소유권과 속성을 설정해 줄 수 있습니다. 이 명령어는 소프트웨어 패키지를 설치할 때에 딱 맞는 명령어처럼 보이는데, 실제로 `Makefiles`(*`make install :`* 섹션)에서 자주 등장합니다. 또한 설치 스크립트에서 찾아 볼 수도 있습니다.

more

less

텍스트 파일이나 스트림을 표준출력으로 한 번에 한 쪽씩 표시해 주는 페이저입니다. 스크립트의 출력을 위한 필터로 쓸 수 있습니다.

12.6. 통신 명령어

정보및 통계

host

DNS를 써서 호스트 이름이나 IP로 그 호스트에 대한 정보를 찾습니다.

vrfy

인터넷 이메일 주소를 검증해(verify) 줍니다.

nslookup

호스트에 대해서 IP를 이용해 인터넷 “네임 서버 룩업”을 합니다. 대화형 모드로 돌거나 스크립트에서 비대화형 모드로 돌 수 있습니다.

dig

nslookup와 비슷하게, 호스트에 대해 IP를 이용해 인터넷 “네임 서버 룩업”을 합니다. 대화형 모드로 돌거나 스크립트에서 비대화형 모드로 돌 수 있습니다.

traceroute

리모트 호스트에 보내는 패킷의 경로를 추적합니다. 이 명령어는 LAN, WAN, 인터넷을 통해서 동작합니다. 리모트 호스트는 IP 주소로 지정되어야 합니다. 파이프에서 grep이나 sed를 이용해 이 명령어의 출력을 필터링 할 수도 있습니다.

ping

지역 네트워크나 외부 네트워크에 있는 다른 머신에게 “ICMP ECHO_REQUEST” 패킷을 브로드캐스트합니다. 네트워크 연결을 테스트 하는데 쓰이는 진단 도구이고, 조심해서 써야 됩니다.

ping이 성공하면 0 번 종료 상태를 리턴하기 때문에 스크립트에서 테스트를 걸 수 있습니다.

```
bash$ ping localhost
PING localhost.localdomain (127.0.0.1) from 127.0.0.1 : 56(84) bytes of data.
Warning: time of day goes back, taking countermeasures.
64 bytes from local-
host.localdomain (127.0.0.1): icmp_seq=0 ttl=255 time=709 usec
64 bytes from local-
host.localdomain (127.0.0.1): icmp_seq=1 ttl=255 time=286 usec
```

```
--- localhost.localdomain ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max/mdev = 0.286/0.497/0.709/0.212 ms
```

whois

DNS(Domain Name System) 룩업을 수행. `-h` 옵션을 쓰면 어떤 서버로 쿼리를 날릴지를 지정할 수 있습니다. 예 5-6을 참고하세요.

finger

네트워크상에 있는 특정 사용자에 대한 정보를 알려줍니다. 사용자의 `~/.plan`, `~/.project`, `~/.forward` 파일이 존재한다면 그 정보들을 표시해 줄 수도 있습니다.

```
bash$ finger bozo
Login: bozo                             Name: Bozo Bozeman
Directory: /home/bozo                   Shell: /bin/bash
On since Fri Aug 31 20:13 (MST) on tty1    1 hour 38 minutes idle
On since Fri Aug 31 20:13 (MST) on pts/0   12 seconds idle
On since Fri Aug 31 20:13 (MST) on pts/1
On since Fri Aug 31 20:31 (MST) on pts/2   1 hour 16 minutes idle
No mail.
No Plan.
```

보안때문에 많은 네트워크들은 **finger** 관련 데몬들을 꺼 놓습니다. [3]

리모트 호스트 접근

sx
rx

sx와 **rx** 명령어는 **xmodem** 프로토콜을 써서 리모트 호스트와 파일을 주고 받을 수 있게 해 줍니다. 이 명령어들은 보통 **minicom** 같은 통신 패키지의 일부분입니다.

sz
rz

sz와 **rz** 명령어는 **zmodem** 프로토콜을 써서 리모트 호스트와 파일을 주고 받을 수 있게 해 줍니다. **zmodem** 프로토콜은 **xmodem** 프로토콜보다 더 나은 전송률을 갖고, 전송이 중지된 파일을 다시 재전송하는 등의 이점이 있습니다. **sx**와 **rx**처럼 이들도 보통은 통신 패키지의 일부분입니다.

ftp

리모트 호스트와 파일을 업로드/다운로드 할 수 있게 해 주는 유틸리및 프로토콜입니다. ftp 세션은 스크립트에서 자동화 될 수 있습니다(예 17-7, 예 A-4, 예 A-8 참고).

cu

리모트 시스템을 호출(**C**all **U**p)해서 간단한 터미널로 연결. 일종의 간단한 telnet 버전입니다.

uucp

유닉스간 복사(UNIX to UNIX copy). 이 명령어는 유닉스 서버간에 파일을 전송하는데 쓰이는 통신 패키지입니다. 쉘 스크립트는 **uucp** 명령어 순서를 효과적으로 처리할 수 있습니다.

인터넷 이메일의 등장으로 **uucp**는 거의 쓰이지 않지만 여전히 존재하고 있고, 인터넷 연결이 불가능하거나 잘 안 되는 경우에도 이 명령어는 완벽하게 동작합니다.

telnet

리모트 호스트로 연결해 주는 프로토콜및 유틸리티.

> **경고**
>
> telnet 프로토콜은 보안 구멍이 있기 때문에 쓰지 말아야 합니다.

rlogin

`Remote login`, 리모트 호스트로 세션을 초기화 해줍니다. 이 명령어는 보안 이슈가 걸려 있기 때문에 ssh을 쓰세요.

rsh

`Remote shell`, 리모트 호스트에서 명령어를 실행시킵니다. 이 명령어는 보안 이슈가 걸려 있기 때문에 **ssh**을 쓰세요.

rcp

리모트 복사(`Remote copy`), 네크워크로 물린 서로 다른 머신끼리 복사를 할 수 있게 해줍니다. 쉘 스크립트에서 **rcp**나 비슷한 종류의 유틸리티들을 쓰는 것은 보안 문제와 관련이 있기 때문에 추천하지 않습니다. 대신 **ssh** 이나 **expect** 스크립트를 고려해 보기 바랍니다.

ssh

`Secure shell`, 리모트 호스트로 로그인 해서 명령어를 실행 시킴. 신원 인증및 암호화를 써서 **telnet**, **rlogin**, **rcp**, **rsh**을 대치하는 명령어입니다. 자세한 사항은 맨 페이지를 참고하세요.

지역 네트워크(Local Network)

write

터미널간 통신을 위한 유틸리티입니다. 자신의 터미널(콘솔이나 hanterm)에서 다른 사용자 터미널로 메세지를 보낼 수 있게 해 줍니다. mesg 명령어는 자신의 터미널로 들어오는 write 메세지를 막을 수 있게 해 줍니다.

write은 대화형 모드로 동작하기 때문에 보통은 스크립트에서 쓰이지 않습니다.

Mail

vacation

메일 수신자가 휴가중(vacation)이거나 잠시 메일을 받을 수 없을 때 자동으로 이메일에 답장을 보내주는 유틸리티입니다. 이 명령어는 **sendmail**과 같이 네트워크에서 동작하기 때문에 다이얼업 POP 메일 계정에서는 쓸 수 없습니다.

12.7. 터미널 제어 명령어

명령어 목록

tput

터미널을 초기화하거나 `terminfo` 파일에 들어 있는 정보를 패치해 줍니다. 몇몇 터미널 동작은 다양한 옵션들로 가능합니다. **tput clear**는 밑에서 설명할 **clear**와 동일한 명령이고 **tput reset**는 역시 밑에서 설명할 **reset**와 동일한 명령입니다.

```
bash$ tput longname
xterm terminal emulator (XFree86 4.0 Window System)
```

터미널을 제어하는데는 stty 명령어 세트가 더 강력한 기능을 제공하는것에 주의하세요.

reset

터미널 파라미터를 리셋하고 텍스트 스크린을 깨끗하게 해 줍니다. **clear**와 마찬가지로 터미널의 제일 위 제일 왼쪽에 커서와 프롬프트가 나타납니다.

clear

clear 명령어는 콘솔이나 한텀의 텍스트 화면을 간단히 청소해 줍니다. 프롬프트와 커서는 스크린이나 한텀 윈도우 왼쪽 최상단에 다시 나타나게 됩니다. 이 명령어는 명령어 줄에서 쓸 수도 있고 스크립트 안에서 쓸 수도 있습니다. 예 10-23을 참고하세요.

script

사용자가 콘솔의 명령어줄이나 한텀 윈도우에서 입력한 모든 키값들을 파일로 저장해서 기억합니다. 실제로는 세션을 기억해 주는 것입니다.

12.8. 수학용 명령어

명령어 목록

factor

정수를 소수로 인수분해합니다.

```
bash$ factor 27417
27417: 3 13 19 37
```

bc
dc

무한 정밀도(arbitrary precision)의 계산을 가능케 해 주는 유연한 유틸리티.

bc는 약간 애매한 C 문법을 따릅니다.

dc는 RPN("역폴란드식 표기법, Reverse Polish Notation")을 씁니다.

둘 중에서 스크립트에서 쓰기에는 **bc**가 더 유용해 보입니다. **bc**는 꽤 잘 동작하는 유닉스 유틸리티이기 때문에 파이프에서 쓰일 수도 있습니다.

bash는 부동소수점 연산을 할 수 없고 몇몇 중요한 산술적인 기능이 빠져 있지만 다행히, **bc**가 이런 부분을 메꿔줍니다.

다음은 **bc**로 스크립트 변수를 계산하기 위한 간단한 틀입니다. 여기서는 명령어 치환을 사용합니다.

```
      variable=$(echo "OPTIONS; OPERATIONS" | bc)
```

예 **12-28.** 저당에 대한 월 상환액**(Monthly Payment on a Mortgage)**

```
#!/bin/bash
# monthlypmt.sh: 저당에 대한 월 상환액을 계산.


# 이 스크립트는 Jeff Schmidt 와 Mendel Cooper(이 문서의 저자인 본인)이 작성한
# "mcalc"(저당액 계산기, mortgage calculator) 패키지의 수정본입니다.
#   http://www.ibiblio.org/pub/Linux/apps/financial/mcalc-
1.6.tar.gz  [15k]

echo
echo "원금, 이자율, 저당 기간을 입력하면 월 상환액을 계산합니다."

bottom=1.0

echo
```

```
echo -n "원금을 넣으세요(콤마 없이) "
read principal
echo -n "이자율을 넣으세요(퍼센트로) "  # 만약에 12% 라면 ".12"가 아니
고 "12"라고 입력.
read interest_r
echo -n "기간을 넣으세요(월 단위) "
read term


 interest_r=$(echo "scale=9; $interest_r/100.0" | bc) # 소수로 변환.
                 # "scale" 로 소수점 이하 몇 자리까지 표현할 것인지를 결정.


 interest_rate=$(echo "scale=9; $interest_r/12 + 1.0" | bc)


 top=$(echo "scale=9; $principal*$interest_rate^$term" | bc)

 echo; echo "시간이 좀 걸리므로, 느긋하게 기다리기 바랍니다."

 let "months = $term - 1"
 for ((x=$months; x > 0; x--))
 do
   bot=$(echo "scale=9; $interest_rate^$x" | bc)
   bottom=$(echo "scale=9; $bottom+$bot" | bc)
#  bottom = $(($bottom + $bot"))
 done

 # let "payment = $top/$bottom"
 payment=$(echo "scale=2; $top/$bottom" | bc)
 # 달러와 센트 표시를 위해 소수점 이하 두 자리만 표시.

 echo
 echo "월 상환액 = \$$payment"  # 결과 앞에 달러 표시를 에코.
 echo


 exit 0

 # 연습문제:
 #   1) 원금 입력시 콤마를 쓸 수 있게 해 보세요.
 #   2) 이자 입력시 퍼센트 표시나 소수점 표시 둘 다 쓸 수 있게 해 보세요.
 #   3) 더 멋있는 스크립트로 만들려면,
 #      완전한 부채 상환표(amortization tables)를 출력하도록 고쳐 보세요.
```

예 **12-29.** 진법 변환**(Base Conversion)**

```
:
```

```
##########################################################################
# Shellscript: base.sh - 숫자를 다른 진법으로 보여줌(Bourne Shell)
# Author     : Heiner Steven (heiner.steven@odn.de)
# Date       : 07-03-95
# Category   : Desktop
# $Id: base.sh,v 1.2 2000/02/06 19:55:35 heiner Exp $
##########################################################################
# Description
#
# Changes
# 21-03-95 stv 입력이 0xb 일 경우 생기는 에러 고침(0.2)
##########################################################################

# ==> 스크립트 저자의 허락하에 사용함.
# ==> 이 표시는 이 문서의 저자가 주석을 붙인 것임.

NOARGS=65
PN=`basename "$0"`        # 프로그램 이름
VER=`echo '$Revision: 1.2 $' | cut -d' ' -f2`  # ==> VER=1.2

Usage () {
    echo "$PN - 숫자를 다른 진법으로 보여줌, $VER (stv '95)
사용법: $PN [숫자 ...]

숫자를 알려주지 않으면 표준 입력에서 읽습니다.
가능한 숫자로는
    2 진수 0b 로 시작(즉, 0b1100)
    8 진수 0 으로 시작(즉, 014)
    16 진수 0x 로 시작(즉, 0xc)
    10 진수 그 외(즉, 12)" >&2
    exit $NOARGS
}   # ==> 사용법을 알려 주는 함수.

Msg () {
    for i   # ==> in [list] 가 빠져 있습니다.
    do echo "$PN: $i" >&2
    done
}

Fatal () { Msg "$@"; exit 66; }

PrintBases () {
    # 숫자의 진법을 결정.
    for i      # ==> in [list] 가 빠져 있어서...
    do         # ==> 명령어줄 인자(들)에 대해서 동작.
case "$i" in
    0b*) ibase=2;; # 2진수
    0x*|[a-f]*|[A-F]*) ibase=16;; # 16진수
    0*) ibase=8;; # 8진수
```

```
    [1-9]*) ibase=10;; # 10진수
    *)
Msg "부적절한 숫자($i)라 무시합니다."
continue;;
esac

# 접두사를 제거하고, 16진수를 대문자로 변환(bc에서 필요함).
number=`echo "$i" | sed -e ’s:^0[bBxX]::’ | tr ’[a-f]’ ’[A-F]’`
# ==> sed 구분자로 "/"가 아닌 ":"를 썼네요.

# 일단 10진수로 변환
dec=`echo "ibase=$ibase; $number" | bc`  # ==> ’bc’ 는 계산용 유틸리티죠.
case "$dec" in
    [0-9]*) ;;  # 됐군요.
    *) continue;;  # 에러네요. 무시합니다.
esac

# 변환된 숫자들을 한 줄로 출력합니다.
# ==> ’here document’ 는 명령어 목록을 ’bc’로 입력시켜 줍니다.
echo `bc <<!
    obase=16; "hex="; $dec
    obase=10; "dec="; $dec
    obase=8;  "oct="; $dec
    obase=2;  "bin="; $dec
!
    ` | sed -e ’s: : :g’

    done
}

while [ $# -gt 0 ]
do
    case "$1" in
--) shift; break;;
-h) Usage;;                 # ==> 도움말.
-*) Usage;;
*) break;; # 첫번째 숫자.
    esac   # ==> 입력에 대해서 에러 체크를 더 하면 좀 더 유용하게 쓸 수 있을 겁니다.
    shift
done

if [ $# -gt 0 ]
then
    PrintBases "$@"
else # 표준입력에서 읽기.
    while read line
    do
PrintBases $line
```

```
    done
fi
```

bc를 실행시키는 다른 방법으로는 명령어 치환 블럭 안에서 here document를 쓰는 것입니다. 이 방법은 스크립트에서 **bc**로 옵션과 명령어를 넘기려고 할 때 특히 적당합니다.

```
variable=`bc << LIMIT_STRING
options
statements
operations
LIMIT_STRING
`
```

```
...아니면...
```

```
variable=$(bc << LIMIT_STRING
options
statements
operations
LIMIT_STRING
)
```

예 12-30. 다른 방법으로 bc 실행

```
#!/bin/bash
# 'bc'를 'here document'와 같이 쓴 것을 명령어 치환을 써서 실행.


var1=`bc << EOF
18.33 * 19.78
EOF
`
echo $var1       # 362.56


#  $( ... ) 표기법도 역시 됩니다.
v1=23.53
v2=17.881
v3=83.501
v4=171.63

var2=$(bc << EOF
scale = 4
a = ( $v1 + $v2 )
b = ( $v3 * $v4 )
a * b + 15.35
EOF
```

```
)
echo $var2       # 593487.8452


var3=$(bc -l << EOF
scale = 9
s ( 1.7 )
EOF
)
# 1.7 라디안의 사인값을 리턴.
# "-l" 옵션은 'bc'의 수학 라이브러리를 호출합니다.
echo $var3       # .991664810


# 이제, 함수에서 해 보죠...
hyp=             # 전역 변수 선언.
hypotenuse ()    # 정삼각형의 빗변을 계산.
{
hyp=$(bc -l << EOF
scale = 9
sqrt ( $1 * $1 + $2 * $2 )
EOF
)
# 불행하게도, Bash 함수에서는 부동형 값을 리턴할 수 없습니다.
}

hypotenuse 3.68 7.31
echo "hypotenuse = $hyp"    # 8.184039344


exit 0
```

12.9. 기타 명령어

명령어 목록

jot
seq

이 유틸리티들은 사용자가 정한 간격대로 정수값을 차례차례 만들어 냅니다. for 루프에서 아주 유용하게 쓰일 수 있습니다.

예 **12-31. seq**로 루프에 인자를 만들어 넣기

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

```
for a in `seq 80`  # 혹은      for a in $( seq 80 )
# for a in 1 2 3 4 5 ... 80   과 똑같습니다(과도한 타이핑을 안 해도 되죠!).
# 시스템에 'jot'이 있다면 대신 쓸 수도 있습니다.
do
  echo -n "$a "
done
# "for" 루프의 [list] 를 만들기 위해서 명령어의 출력을 사용하는 예제

echo; echo


COUNT=80  # 'seq'에 변경할 수 있는 매개변수를 줘도 되는군요.

for a in `seq $COUNT`  # 혹은   for a in $( seq $COUNT )
do
  echo -n "$a "
done

echo

exit 0
```

run-parts

run-parts 명령어[4]는 대상 디렉토리에 들어 있는 모든 스크립트를 파일명을 아스키 순서대로 차례로 실행시켜 줍니다. 당연한 이야기지만 그 스크립트들은 실행 퍼미션이 걸려 있어야 됩니다.

crond 데몬이 `/etc/cron.*` 디렉토리에 들어 있는 스크립트를 실행 시키기 위해 **run-parts** 를 실행시킵니다.

yes

yes의 기본 동작은 `y`문자와 라인 피드를 표준출력으로 계속 뿌리는데 **control-c**로 멈출 수 있습니다. **`yes`** 다른 문자열처럼 하면 다른 문자열을 계속 뿌려 줍니다. 그러면 왜 이런 명령어가 필요할까요? 명령어 줄이나 스크립트 상에서 사용자 입력을 원하는 프로그램에게 **yes**의 출력을 재지향 시키거나 파이프를 통해 전달할 수 있습니다. 사실, 이 명령어는 **expect**의 간단한 버전입니다.

`yes | fsck /dev/hda1` 라고 하면 **fsck**를 한 방에 실행(non-interactive)시킵니다(조심하세요!).

`yes | rm -r dirname` 는 **`rm -rf dirname`**라고 한 것과 같은 효과를 갖습니다(조심하세요!).

주의

yes를 fsck 나 fdisk 처럼 어쩌면 위험할 수도 있는 시스템 명령어에 파이프를 걸 때는 아주 조심해야 됩니다.

banner

주어진 인자를 아스키 문자로 이루어진(기본은 '#') 큰 수직 배너로 만들어 표준출력으로 출력해 줍니다. 프린터로 재지향시켜 실제로 출력해 볼 수도 있습니다.

printenv

특정 사용자의 모든 환경 변수들을 보여줍니다.

```
bash$ printenv | grep HOME
HOME=/home/bozo
```

lp

lp와 **lpr**은 파일을 프린트 큐로 보내서 프린터로 출력하기 위해 쓰이는 명령어입니다. [5] 이 명령어들의 이름은 옛날 라인 프린터(line printer)를 쓰던 시절에서 유래 됐습니다.

`bash$ lp file1.txt` 라고 하거나 `bash$ lp <file1.txt`

형식화된 출력을 얻기 위해 **pr**에서 **lp**로 파이프를 연결해서 자주 쓰입니다.

```
bash$ pr -options file1.txt | lp
```

groff이나 고스트스크립트(ghostscript)같은 형식화 패키지들은 자신의 출력을 **lp**로 직접 보내기도 합니다.

```
bash$ groff -Tascii file.tr | lp
```

```
bash$ gs -options | lp file.ps
```

관련 명령어로는 프린트 큐를 보기 위한 **lpq**나 프린트 큐에서 특정 작업을 지우기 위한 **lprm**이 있습니다.

tee

[유닉스는 수도업계에서 이 아이디어를 얻었습니다.]

이 명령어는 재지향 연산자이긴 하지만 약간 다릅니다. 수도 배관공의 **tee**(T자형 배관 파이프)처럼 단일 명령어나 파이프의 일부분으로 동작하는 명령어의 출력을 "빨아 들여" 다른 곳으로 동일한 결과를 복사해 내지만 결과에는 아무 영향을 미치지 않습니다. 진행중인 프로세스의 상황을 파일이나 종이로 출력할 때 유용한데, 보통은 디버깅 용도로 쓰여 중간 결과값을 추적하는데 쓰입니다.

```
               tee
              |------> 파일로
              |
```

```
===============|===============
명령어 --->----|-연산자--->---> 명령어의 결과
==============================
```

```
cat listfile* | sort | tee check.file | uniq > result.file
```

(`check.file`은 uniq가 중복된 줄을 지우기 전의 상태인 **`cat listfile* | sort`** 까지의 상태를 갖고 있습니다.)

mkfifo

이름이 약간 애매한 이 명령어는 네임드 파이프(named pipe)를 만들어 냅니다. 네임드 파이프란 프로세스끼리 데이타를 주고 받을 수 있도록 하는 임시 **first-in-first-out** 버퍼를 가르키는 말입니다. [6] 전형적인 시나리오는 한 프로세스가 FIFO에 데이타를 쓰고 다른 프로세스는 그 데이타를 읽어 가는 것입니다. 예 A-10를 참고하세요.

pathchk

파일명이 올바른지 아닌지를 확인해 줍니다. 파일명이 최대 허용 가능 길이(255 글자)를 넘는다거나 자기 이름에 들어 있는 경로중 하나 이상의 디렉토리가 찾을 수 없다거나 할 경우에는 에러 메세지를 발생시킵니다. 불행하게도 **pathchk**는 알아 볼 수 있는 에러 코드를 리턴시키지 않기 때문에 스크립트 상에서는 전혀 쓸모가 없습니다.

dd

어딘지 모르게 불명확하고 약간은 쓰기 꺼려하는 "데이타 복사기"(data duplicator) 명령어입니다. 원래는 마그네틱 테이프를 이용해서 유닉스의 미니 컴퓨터와 IBM의 메인프레임간에 데이타를 교환하는데 쓰이던 유틸리티지만 아직도 그 쓰임새가 남아 있습니다. **dd** 명령어는 변환 과정을 거쳐 파일이나 표준입력/표준출력을 간단히 복사해 줍니다. 아스키/EBCDIC [7] 간 변환, 대소문자간 변환, 입출력간의 바이트 쌍을 바꾸거나 입력 파일의 처음이나 끝을 건너뛰거나 잘라내서 출력 파일을 만들어 내는 등의 변환이 가능합니다. **`dd --help`** 라고 하면 이 강력한 유틸리티가 처리할 수 있는 변환 목록과 다른 옵션을 볼 수 있습니다.

```
# 'dd' 연습하기.

n=3
p=5
input_file=project.txt
output_file=log.txt

dd if=$input_file of=$output_file bs=1 skip=$((n-1)) count=$((p-
n+1)) 2> /dev/null
# $input_file 의 n 에서 p 까지 문자를 추출해 내기.
```

```
echo -n "hello world" | dd cbs=1 conv=unblock 2> /dev/null
# "hello world" 를 수직으로 에코시키기


# Thanks, S.C.
```

dd가 얼마나 다재다능한지를 보여주기 위해서 키보드 입력을 갈무리하는데 써 보겠습니다.

예 **12-32.** 키보드 입력을 갈무리하기

```
#!/bin/bash
# ENTER 없이 키누름을 갈무리하기.


keypresses=4                      # 갈무리할 키누름 수.


old_tty_setting=$(stty -g)        # 현재의 터미널 세팅을 저장.

echo "키를 $keypresses 번 누르세요."
stty -icanon -echo                # 캐노니컬(canonical) 모드 끄기.
                                  # 로컬 에코 끄기.
keys=$(dd bs=1 count=$keypresses 2> /dev/null)
# 'dd' 는 "if" 옵션이 없으면 표준입력을 씁니다.

stty "$old_tty_setting"           # 예전 터미널 세팅으로 복구.

echo "\"$keys\" 키를 눌렀습니다."

# S.C. 가 이 방법을 알려줬습니다.
exit 0
```

dd 명령어는 데이타 스트림에 대해 랜덤한 접근을 할 수 있습니다.

```
echo -n . | dd bs=1 seek=4 of=file conv=notrunc
# "conv=notrunc" 는 출력 파일이 잘리지(truncated) 않을 것이라는 옵션입니다.

# Thanks, S.C.
```

dd는 플로피나 테입 드라이브(예 A-5)같은 디바이스의 raw 데이타나 디스크 이미지에 직접 접근할 수 있기 때문에 보통은 부트 플로피를 만들 때 씁니다.

```
dd if=kernel-image of=/dev/fd0H1440
```

비슷하게, 심지어 "다른" OS 에서 포맷된 디스켓을 하드 디스크의 이미지 파일로 통째로 복사할 수 있습니다.

```
dd if=/dev/fd0 of=/home/bozo/projects/floppy.img
```

dd는 임시 스왑 파일을 초기화 하거나(예 29-2) 램디스크를 초기화하는데(예 29-3) 쓸 수 있습니다. 추천할만한 방법은 아니지만 전체 하드 드라이브 파티션을 저레벨로 복사해 낼 수도 있습니다.

아마도 시간이 남아도는 사람들은 **dd**를 어떻게 재미있게 쓸 수 있을지 계속 생각할 겁니다.

예 **12-33.** 파일을 안전하게 지우기

```
#!/bin/bash
# blotout.sh: 파일의 모든 기록 지우기.

#  이 스크립트는 대상 파일을 지우기 전에
#+ 임의의 바이트들로 덮어쓰고, 0으로 덮어쓰기를 반복합니다.
#  이렇게 하고 나면, 디스크 섹터를 물리적으로 검사해도
#+ 원래의 파일 데이타를 찾아 낼 수 없습니다.

PASSES=7         # 파일 조각(file-shredding) 단계.
BLOCKSIZE=1      #  /dev/urandom 으로 I/O 를 할 때 필요한 유닛 블럭 크기.
                 #+ 이 크기가 지정되지 않으면 이상한 결과가 나옵니다.
E_BADARGS=70
E_NOT_FOUND=71
E_CHANGED_MIND=72

if [ -z "$1" ]   # 파일이 지정되지 않았음.
then
  echo "사용법: `basename $0` filename"
  exit $E_BADARGS
fi

file=$1

if [ ! -e "$file" ]
then
  echo "\"$file\" 파일을 찾을 수 없음."
  exit $E_NOT_FOUND
fi

echo; echo -n "\"$file\" 파일을 정말로 완전히 지워 버리겠습니까(y/n)? "
read answer
case "$answer" in
[nN]) echo "하기 싫다구요?"
      exit $E_CHANGED_MIND
      ;;
*)    echo "\"$file\" 파일을 완전히 지우는 중.";;
esac


flength=$(ls -l "$file" | awk '{print $5}')  # 5 번째 필드가 파일 길이.
```

```
pass_count=1

echo

while [ "$pass_count" -le "$PASSES" ]
do
  echo "$pass_count 번째 단계"
  sync         # 버퍼 플러쉬.
  dd if=/dev/urandom of=$file bs=$BLOCKSIZE count=$flength
               # 파일을 임의의 바이트들로 덮어씀.
  sync         # 다시 버퍼 플러쉬.
  dd if=/dev/zero of=$file bs=$BLOCKSIZE count=$flength
               # 파일을 0으로 덮어씀.
  sync         # 또 다시 버퍼 플러쉬.
  let "pass_count += 1"
  echo
done


rm -f $file    # 마지막으로, 온통 뒤섞이고 조각나 버린 파일을 삭제.
sync           # 마지막 버퍼 플러쉬.

echo "\"$file\" 파일이 완전히 삭제되었습니다."; echo


#  이 스크립트는 파일을 완전히 "조각내는데"(shredding) 비효율적이고 느린 방법을
#+ 쓴 것만 빼면 정말 안전합니다. GNU "fileutils" 패키지중의 하나인
#+ "shred" 명령어도 똑같은 일을 하지만 좀 더 효율적입니다.

#  이 스크립트로 지워진 파일은 일반적인 방법으로는 "undelete"되거나
#+ 복구해 낼 수 없습니다.
#  하지만...
#+ 이런 간단한 방법은 과학적인 분석(forensic analysis)까지 막아내지는 못합니다.


#  Tom Vier의 "wipe" 파일 삭제 패키지는 이 간단한 스크립트보다 좀 더
#+ 완전하게 파일을 지워 줍니다.
#     http://www.ibiblio.org/pub/Linux/utils/file/wipe-2.0.0.tar.bz2

#  파일 삭제와 보안에 대해서 좀 더 자세하게 알고 싶다면,
#+ Peter Gutmann의 논문을 참고하기 바랍니다.
#+     "Secure Deletion of Data From Magnetic and Solid-State Memory".
#           http://www.cs.auckland.ac.nz/~pgut001/secure_del.html


exit 0
```

od

od(**8**진 덤프, octal dump) 명령어는 입력(혹은 파일)을 8진수나 다른 진수로 변환해 줍니다. **od**는 바이너리 데이타 파일이나 `/dev/urandom` 같은 읽을 수 없는 시스템 디바이스 파일을 읽거나 처리하려고 할 때 필터로 쓸 수 있는 유용한 명령어입니다. 예 9-21과 예 12-10를 참고하세요.

hexdump

이진 파일을 16진수나 8진수, 10진수, 아스키로 덤프를 뜸. 이 명령어는 위에서 설명했던 **od**와 거의 비슷하지만 그렇게 쓸모가 있지는 않습니다.

m4

숨겨진 보물인 **m4**는 강력한 매크로 프로세서[8] 유틸리티이나 거의 완전한 언어에 가깝습니다. **m4**는 사실 eval, tr, awk의 몇가지 기능들을 묶어 놓은 것입니다.

예 **12-34. m4** 쓰기

```
#!/bin/bash
# m4.sh: m4 매크로 프로세서 쓰기.

# 문자열
string=abcdA01
echo "len($string)" | m4                           # 7
echo "substr($string,4)" | m4                      # A01
echo "regexp($string,[0-1][0-1],\&Z)" | m4      # 01Z

# 산술식
echo "incr(22)" | m4                               # 23
echo "eval(99 / 3)" | m4                           # 33

exit 0
```

주석

1. `~/.Xdefaults`처럼 점으로 시작하는 파일들이 있는데 그런 파일들은 보통의 **ls**만으로는 나타나지 않고, 실수로 **rm -rf** *라고 해도 지워지지 않습니다. 도트 파일은 보통 셋업이나 설정을 위해서 사용자의 홈디렉토리에서 놓여집니다.
2. 이 명령어는 단일 시스템이나 지역 네트워크에 있는 파일을 암호화 하는데 쓰이는 대칭형 블럭 암호화(symmetric block cipher)명령어입니다. 이와 반대 개념인 “공개키” 암호화란 것도 있는 데 이 암호화의 유명한 예가 바로 **pgp**입니다.
3.

데몬(daemon)은 터미널 세션과 연결되어 있지 않는 백그라운드 프로세스입니다. 데몬은 주어진 시간이나 어떤 이벤트가 발생했을 때 지정된 서비스를 수행합니다.

“데몬”이란 말은 신비스럽거나 거의 초자연적인 무언가가 있는 유령이란 뜻의 그리스어인데, 유닉스 데몬이 남몰래 조용히 자신에게 주어진 일을 수행하는 방법을 나타냅니다.

4. 사실 이 명령어는 데비안 리눅스 배포판에서 쓰던 것입니다.
5. 프린트 큐란 프린트를 위해 “순서대로 대기”하고 있는 작업 그룹을 말합니다.
6. Andy Vaught가 리눅스 저널 (http://www.linuxjournal.com)에 1997년 9월에 기고한 Introduction to Named Pipes (http://www2.linuxjournal.com/lj-issues/issue41/2156.html)란 아주 훌륭한 문서를 참고하세요.
7. EBCDIC (발음은 “엡시딕”, ebb-sid-ic) 은 Extended Binary Coded Decimal Interchange Code 의 첫 글자만 딴 두문자어입니다. IBM 의 데이타 포맷으로 더 이상 쓰이지 않습니다. `conv=ebcdic` 옵션의 이상한 응용으로는 아주 빠르고 간단하지만 그렇게 안전하지는 않은 텍스트 파일 인코더로 쓸 수 있습니다.

   ```
   cat $file | dd conv=swab,ebcdic > $file_encrypted
   # 인코드(언뜻 봐서는 알아볼 수가 없습니다).
   # 더 이해하기 힘들게 해려면 바이트를 스왑(swab)시킬 수도 있습니다.

   cat $file_encrypted | dd conv=swab,ascii > $file_plaintext
   # 디코드.
   ```

8. 매크로는 명령어 문자열이나 매개변수에 따라 여러가지 연산으로 확장되는 심볼릭 상수입니다.

13장. 시스템과 관리자용 명령어

시스템과 관리자용 명령어의 좋은 예는 `/etc/rc.d` 에 있는 시작, 종료 스크립트들입니다. 이 명령어들은 보통 시스템 관리나 파일시스템을 긴급하게 고치려고 할 때 루트가 사용합니다. 이들 몇몇은 잘못 쓰면 시스템을 망가트릴 수 있기 때문에 사용에 주의를 요합니다.

사용자와 그룹

chown
chgrp

chown 명령어는 파일의 소유권을 바꿔줍니다. *root*가 특정 사용자가 소유한 파일을 다른 사용자용으로 바꾸려고 할 때 유용하게 쓰입니다. 하지만, 일반 사용자는 자신이 소유한 파일조차도 소유권을 바꿀 수 없습니다. [1]

```
root# chown bozo *.txt
```

chgrp 명령어는 파일의 그룹 소유권을 바꿔줍니다. 이 명령어를 쓰려면 그 파일의 소유자이고 바꾸려는 그룹의 멤버여야 합니다(혹은 *root*이거나).

```
chgrp --recursive dunderheads *.data
# $PWD 디렉토리의 모든 하위 디렉토리("recursive"에 의해)의
# 모든 "*.data" 파일들은 "dunderheads" 그룹이 그 소유권을 갖습니다.
```

useradd
userdel

관리자용 명령어인 **useradd**는 시스템에 사용자 계정을 추가해 주고 그 사용자용으로 지정된 홈 디렉토리를 만들어 줍니다. **useradd**와 쌍을 이루는 **userdel**는 시스템에서 사용자 계정을 삭제해 주고 [2] 해당 파일들도 삭제해 줍니다.

> 참고: **adduser** 명령어는 **useradd**의 동의어로서, 보통 **useradd**를 가르키는 심볼릭 링크 파일입니다.

id

id 명령어는 현재 사용자의 실제 ID와 유효 사용자 ID, 그룹 ID를 보여줍니다. 내부 bash 변수인 $UID, $EUID, $GROUPS와 짝을 이룹니다.

```
bash$ id
uid=501(bozo) gid=501(bozo) groups=501(bozo),22(cdrom),80(cdwriter),81(audio)

bash$ echo $UID
501
```

예 9-4 참고.

who

시스템에 현재 로그인해 있는 모든 사용자를 보여줍니다.

```
bash$ who
bozo  tty1     Apr 27 17:45
bozo  pts/0    Apr 27 17:46
bozo  pts/1    Apr 27 17:47
bozo  pts/2    Apr 27 17:49
```

`-m`을 주면 오직 현재 사용자에 대한 자세한 정보만을 보여줍니다. **who am i**나 **who The Man**처럼 **who**에 아무 인자나 두 개 넘겨주면 **who -m** 이라고 한 것과 같습니다.

```
bash$ who -m
localhost.localdomain!bozo  pts/2    Apr 27 17:49
```

whoami는 **who -m** 과 비슷하지만 사용자 이름만 보여줍니다.

```
bash$ whoami
bozo
```

w

로그인 되어 있는 사용자와 그 사용자와 관련된 모든 프로세스를 보여 줍니다. 이는 **who**의 확장 버전인데, **w**의 출력에 **grep**으로 파이프를 걸어서 특정한 사용자나 프로세스를 찾을 수 있습니다.

```
bash$ w | grep startx
bozo  tty1     -                 4:22pm  6:41   4.47s  0.45s  startx
```

logname

현재 사용자의 로그인 이름을 `/var/run/utmp`에서 찾아서 보여줍니다. 위에서 설명한 whoami와 거의 동일한 명령어입니다.

```
bash$ logname
bozo

bash$ whoami
bozo
```

그렇지만...

```
bash$ su
Password: ......
```

```
bash# whoami
root
bash# logname
bozo
```

su

다른 사용자(substitute **u**ser)로 프로그램이나 스크립트를 실행 시킵니다. **rjones**란 사용자로 쉘을 새롭게 시작하고 싶으면 **su rjones**라고 하면 됩니다. 옵션 없이 **su**만 실행시키면 기본적으로 **root** 로 받아들입니다. 예 A-10를 참고하세요.

users

로그인 하고 있는 모든 사용자를 보여줍니다. 이 명령어는 **who -q** 와 거의 비슷한 명령어입니다.

ac

사용자가 로그인 해 있던 시간을 `/var/log/wtmp` 에서 읽어서 보여줍니다. 이 명령어는 GNU 계정 유틸리티(accounting utility) 중 하나입니다.

```
bash$ ac
        total       68.08
```

last

사용자가 마지막으로 로그인 한 시간을 `/var/log/wtmp`에서 읽어서 보여줍니다. 이 명령어는 외부에서 로그인 한 정보도 보여줄 수 있습니다.

groups

현재 사용자가 속해 있는 그룹을 보여줍니다. 내부 변수인 $GROUPS에 해당하는 명령어이지만 숫자가 아닌 그룹 이름으로 보여줍니다.

```
bash$ groups
bozita cdrom cdwriter audio xgrp

bash$ echo $GROUPS
501
```

newgrp

로그아웃 없이 사용자의 그룹 ID를 변경하기. 이 명령어를 쓰면 새 그룹의 파일에 접근할 수 있게 됩니다. 사용자는 보통 동시에 여러 그룹의 멤버이기 때문에 이 명령어를 쓸 일은 별로 없습니다.

터미널

tty

현재 사용자의 터미널 이름을 보여줍니다. 서로 다른 한텀, 엑스텀 윈도우는 서로 다른 터미널로 인식되는것에 주의하세요.

```
bash$ tty
/dev/pts/1
```

stty

터미널 세팅을 보여주거나 변경해 줍니다. 이 복잡한 명령어는 스크립트에서 쓰여 터미널 동작이나 출력하는 방법을 제어할 수 있습니다. info 페이지를 참고하고, 조심해서 공부하세요.

예 **13-1.** 지움 글자**(erase character)** 세팅하기

```
#!/bin/bash
# erase.sh: "stty"로 입력시의 지움 글자(erase character)를 세트하기.

echo -n "이름이 뭐에요? "
read name                      # 아무 글자나 치고 지우려고 해보세요.
                               # 안 될 겁니다.
echo "이름이 $name 군요."

stty erase '#'                 # "hashmark" (#) 를 지움 글자로 세트.
echo -n "이름이 뭐죠? "
read name                      # 마지막에 친 글자를 # 으로 지워보세요.
echo "$name 가 당신 이름이군요."

exit 0
```

예 **13-2.** 비밀스런 비밀번호**:** 터미널 에코 끄기

```
#!/bin/bash

echo
echo -n "비밀번호를 넣으세요 "
read passwd
echo "비밀번호는 $passwd 입니다."
echo -n "누군가가 어깨 너머로 당신을 보고 있었다면, "
echo "당신의 비밀번호를 알아냈을 수도 있습니다."

echo && echo  # "and list"로 묶인 라인 피드 두 줄"

stty -echo    # 화면 에코를 끕니다.

echo -n "비밀번호를 다시 넣으세요 "
read passwd
echo
```

```
echo "비밀번호는 $passwd 입니다."
echo

stty echo     # 화면 에코를 킵니다.

exit 0
```

stty를 창조적으로 써서 사용자가 **ENTER**를 안 눌러도 어떤 키를 눌렀는지를 알아낼 수 있습니다.

예 **13-3.** 키누름 알아내기

```
#!/bin/bash
# keypress.sh: 키누름 알아내기("hot keyboard").

echo

old_tty_settings=$(stty -g)   # 현재 세팅을 저장.
stty -icanon
Keypress=$(head -c1)          # GNU 시스템이 아니라면
                              # $(dd bs=1 count=1 2> /dev/null)

echo
echo "\""$Keypress"\" 키가 눌렸습니다."
echo

stty "$old_tty_settings"      # 원래 세팅으로 복구.

# Thanks, Stephane Chazelas.

exit 0
```

예 9-3 참고.

터미널과 모드

터미널은 보통 캐노니컬(canonical) 모드로 동작합니다. 사용자가 키를 누르면 그 터미널에서 실행중인 프로그램에게 즉시 전달되지 않고, 터미널 지역 버퍼가 키 누름을 저장하고 있다가 **ENTER**를 눌렀을 때에 저장하고 있던 값들을 모두 전달하게 됩니다. 터미널에 기본적인 줄단위 편집기가 들어 있는 것과 비슷합니다.

```
bash$ stty -a
speed 9600 baud; rows 36; columns 96; line = 0;
intr = ^C; quit = ^\; erase = ^H; kill = ^U; eof = ^D; eol = <undef>; eol2 = <undef
start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V; flush = ^O;
...
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -
tostop -echoprt
```

캐노니컬 모드에서는 지역 터미널 줄단위 편집기용 특수키의 재정의가 가능합니다.

```
bash$ cat > filexxx
wha<ctl-W>I<ctl-H>foo bar<ctl-U>hello world<ENTER>
<ctl-D>
bash$ cat filexxx
hello world
bash$ bash$ wc -c < file
13
```

터미널을 제어하는 프로세스는 사용자가 26번의 키를 눌렀음에도 오직 13개의 문자(알파벳 12개, 뉴라인 한 개)만 받았습니다.

넌캐노니컬("raw") 모드에서는 누르는 모든 키(**ctl-H**같은 편집용 특수키도 포함)가 제어 프로세스에게 즉시 전달됩니다.

bash 프롬프트는 `icanon` 옵션과 `echo` 옵션을 둘 다 꺼서 기본적인 줄단위 편집을 더 정교하게 제어할 수 있게 합니다. 예를 들어, bash 프롬프트 상에서 **ctl-A**을 누르면 **^A**가 터미널에 찍히지 않고 bash가 **\1** 문자를 읽고 해석해서 커서를 줄 제일 처음으로 이동시켜 줍니다.

Stephane Chazelas

tset

터미널 세팅을 보여주거나 초기화 함. **stty**보다 기능이 떨어집니다.

```
bash$ tset -r
Terminal type is xterm-xfree86.
Kill is control-U (^U).
```

```
Interrupt is control-C (^C).
```

setserial

시리얼 포트 매개변수를 세트하거나 보여줍니다. 루트로 실행시켜야 하고 보통은 시스템 셋업 스크립트에서 찾을 수 있습니다.

```
# /etc/pcmcia/serial 스크립트에서 발췌:

IRQ=`setserial /dev/$DEVICE | sed -e 's/.*IRQ: //'`
setserial /dev/$DEVICE irq 0 ; setserial /dev/$DEVICE irq $IRQ
```

getty
agetty

터미널용 초기화 프로세스가 **getty**나 **agetty**를 써서 사용자의 로그인을 설정해 줍니다. 이 명령어들은 사용자의 쉘 스크립트에서 쓰이지 않기 때문에 쉘 스크립트에서 이런 기능을 쓰려면 **stty**를 쓰기 바랍니다.

mesg

현재 사용자의 터미널에 대한 쓰기 접근을 제어합니다. 접근을 못 하게 설정되면 네트워크에 있는 다른 사용자가 현재 터미널로 write를 하지 못하게 해 줍니다.

> 작은 정보: 여러분이 텍스트 파일을 편집하고 있는데 갑자기 피자 주문 메세지가 뜨면 아주 짜증날 것입니다. 다중 사용자 네트워크에서는, 방해받기 싫을 때 여러분의 터미널에 대한 쓰기 접근을 막고 싶은 경우가 생길겁니다.

wall

"write all"의 앞글자를 따서 **wall**이 된 이 명령어는 현재 로그인 되어 있는 모든 사용자에게 메세지를 날립니다. 원래는 유용한 시스템 관리자용 도구입니다. 예를 들어, 시스템에 문제가 생겨서 잠깐 동안 다운 시켜야 할 필요가 생겼을 때 모든 사용자들에게 경고를 할 수 있게 해 줍니다(예 17-2 참고).

```
bash$ wall System going down for maintenance in 5 minutes!
Broadcast message from bozo (pts/1) Sun Jul  8 13:53:27 2001...

System going down for maintenance in 5 minutes!
```

> 참고: **mesg**로 쓰기가 막혀있는 터미널은 **wall** 메세지를 받을 수 없습니다.

dmesg

시스템 부팅 메세지를 표준출력으로 보여 줍니다. 디버깅 할 때, 어떤 디바이스 드라이버가 설치됐는지 확인할 때, 사용중인 시스템 인터럽트가 무엇인지 확인할 때 편하게 쓸 수 있습니다. 스크립트에서 **dmesg**의 출력을 grep이나 sed, awk로 파싱해서 쓸 수 있습니다.

정보및 통계

uname

시스템 사양(OS, 커널 버전등)을 표준출력으로 보여줍니다. `-a` 옵션을 주면 시스템 정보를 아주 자세하게 보여주고(예 12-4 참고), `-s` 옵션을 주면 OS 종류만 보여줍니다.

```
bash$ uname -a
Linux localhost.localdomain 2.2.15-
2.5.0 #1 Sat Feb 5 00:13:43 EST 2000 i686 unknown

bash$ uname -s
Linux
```

arch

시스템 아키텍쳐를 보여줍니다. **uname -m** 과 동일한 명령어입니다. 예 10-24를 참고하세요.

```
bash$ arch
i686

bash$ uname -m
i686
```

lastcomm

`/var/account/pacct` 파일에 저장돼 있는 이전 명령어들에 대한 정보를 알려줍니다. 옵션으로 명령어와 사용자 이름을 지정해 줄 수 있습니다. 이 명령어는 GNU 계정 유틸리티(accounting utility)중의 하나입니다.

lastlog

시스템의 모든 사용자가 마지막으로 로그인한 시간을 보여줍니다. 이 명령어는 `/var/log/lastlog` 파일을 참조합니다.

```
bash$ lastlog
root          tty1                      Fri Dec  7 18:43:21 -0700 2001
bin                                     **Never logged in**
daemon                                  **Never logged in**
...
bozo          tty1                      Sat Dec  8 21:14:29 -0700 2001
```

```
bash$ lastlog | grep root
root          tty1                      Fri Dec  7 18:43:21 -0700 2001
```

> **경고**
>
> `/var/log/lastlog` 파일에 읽기 퍼미션이 없는 사용자가 이 명령어를 실행시키면 실패합니다.

lsof

현재 열려 있는 파일들을 보여줍니다. 이 명령어는 현재 열려 있는 모든 파일들에 대한 자세한 표와 각각의 파일에 대한 소유자, 크기, 관련 프로세스등의 정보를 보여 줍니다. 당연히, **lsof**의 출력은 파이프를 통해 grep나 awk로 넘겨서 파싱해서 분석할 수 있습니다.

```
bash$ lsof
COMMAND     PID    USER   FD    TYPE      DEVICE    SIZE     NODE NAME
init          1    root  mem     REG         3,5   30748    30303 /sbin/init
init          1    root  mem     REG         3,5   73120     8069 /lib/ld-
2.1.3.so
init          1    root  mem     REG         3,5  931668     8075 /lib/libc-
2.1.3.so
cardmgr     213    root  mem     REG         3,5   36956    30357 /sbin/cardmgr
...
```

strace

시스템 콜과 시그널을 추적해서 진단하고 디버깅해 주는 도구입니다. 가장 간단하게 실행시키는 방법은 **strace COMMAND**라고 치는 것입니다.

```
bash$ strace df
execve("/bin/df", ["df"], [/* 45 vars */]) = 0
 uname({sys="Linux", node="bozo.localdomain", ...}) = 0
 brk(0)                                  = 0x804f5e4
 ...
```

이 명령어는 리눅스에서의 **truss** 입니다.

free

메모리와 캐쉬 사용량을 탭이 들어간 형태로 보여줍니다. 이 명령어의 출력은 grep이나, awk, **Perl**을 써서 파싱하기에 알맞은 형태입니다. **procinfo** 명령어는 **free**가 보여주는 정보 이외에 더 많은 정보도 보여줍니다.

```
bash$ free
                total       used       free     shared    buffers     cached
```

```
  Mem:          30504       28624        1880       15820        1608       16376
  -/+ buffers/cache:        10640       19864
  Swap:         68540        3128       65412
```

사용하지 않는 램 용량을 보려면:

```
bash$ free | grep Mem | awk '{ print $4 }'
1880
```

procinfo

`/proc` 가상 파일시스템에서 여러 정보와 통계를 뽑아내서 광범위하고 자세하게 보여 줍니다.

```
bash$ procinfo | grep Bootup
Bootup: Wed Mar 21 15:15:50 2001    Load average: 0.04 0.21 0.34 3/47 6829
```

lsdev

설치된 하드웨어 디바이스의 목록을 보여줍니다.

```
bash$ lsdev
Device            DMA   IRQ  I/O Ports
------------------------------------------------
cascade             4     2
dma                          0080-008f
dma1                         0000-001f
dma2                         00c0-00df
fpu                          00f0-00ff
ide0                     14  01f0-01f7 03f6-03f6
...
```

du

디스크의 파일 사용량을 재귀적으로 보여줍니다. 특별히 지정하지 않으면 현재 디렉토리에 대해서 동작합니다.

```
bash$ du -ach
1.0k    ./wi.sh
1.0k    ./tst.sh
1.0k    ./random.file
6.0k    .
6.0k    total
```

df

파일시스템 사용량을 탭이 들어간 형태로 보여 줍니다.

```
bash$ df
Filesystem           1k-blocks      Used Available Use% Mounted on
/dev/hda5               273262     92607    166547  36% /
```

```
/dev/hda8              222525    123951     87085  59% /home
/dev/hda7             1408796   1075744    261488  80% /usr
```

stat

주어진 파일(디렉토리나 디바이스 파일도)에 대해서 자세한 통계(**stat**istics)를 알려줍니다.

```
bash$ stat test.cru
  File: "test.cru"
  Size: 49970        Allocated Blocks: 100          Filetype: Regular File
  Mode: (0664/-rw-rw-r--)         Uid: (  501/ bozo)  Gid: (  501/ bozo)
Device:  3,8   Inode: 18185     Links: 1
Access: Sat Jun  2 16:40:24 2001
Modify: Sat Jun  2 16:40:24 2001
Change: Sat Jun  2 16:40:24 2001
```

존재하지 않는 파일에 대해서 **stat**을 실행시키면 에러 메세지를 냅니다.

```
bash$ stat nonexistent-file
nonexistent-file: No such file or directory
```

vmstat

가상 메모리(virtual memory) 통계(statistics)를 보여줌.

```
bash$ vmstat
   procs                      memory    swap          io system         cpu
r  b  w   swpd   free   buff  cache  si  so    bi    bo   in    cs  us  sy id
0  0  0      0  11040   2636  38952   0   0    33     7  271    88   8   3 89
```

netstat

라우팅 테이블이나 활성화되어 있는 네트워크 연결같은 네트워크 통계와 정보를 보여줍니다. 이 유틸리티는 `/proc/net`(28장)에서 정보를 얻어 옵니다. 예 28-2을 참고하세요.

uptime

시스템이 얼마나 오랫동안 돌고 있었는지 관련 통계와 함께 보여줍니다.

```
bash$ uptime
10:28pm  up  1:57,  3 users,  load average: 0.17, 0.34, 0.27
```

hostname

시스템의 호스트명을 보여줍니다. 이 명령어는 `/etc/rc.d` 에 들어 있는 셋업 스크립트에서 호스트명을 설정해 줍니다(`/etc/rc.d/rc.sysinit`이나 비슷한 스크립트). **uname -n**과 동일한 명령어이고 내부 변수인 $HOSTNAME과 연관이 있습니다.

```
bash$ hostname
localhost.localdomain

bash$ echo $HOSTNAME
localhost.localdomain
```

hostid

호스트 머신에 대한 32비트 16진수 구분자를 에코해 줍니다.

```
bash$ hostid
7f0100
```

> 참고: 이 명령어는 특정 시스템에 대해 "유일한"(unique) 시리얼 숫자를 구해줍니다. 몇몇 상업용 제품의 등록 과정에서 이 숫자를 이용해 사용자 라이센스를 만들어 냅니다. 하지만 불행하게도 **hostid**는 오직 네트워크 주소를 두 바이트 단위로 뒤집어 16진수로 리턴해 줍니다.
>
> 네트워크에 물리지 않은 리눅스 머신의 전형적인 네트워크 주소는 `/etc/hosts`에서 알아낼 수 있습니다.
>
> ```
> bash$ cat /etc/hosts
> 127.0.0.1               localhost.localdomain  localhost
> ```
>
> 공교롭게도 **`127.0.0.1`**을 두 바이트 단위로 뒤집으면 **`0.127.1.0`**이 되고 이를 16진수로 변환하면 **`007f0100`**이 되는데 이는 위에서 살펴본 **hostid**가 리턴하는 값과 정확히 일치합니다. 결국 동일한 **hostid**를 갖는 리눅스 머신이 수 백만 개가 존재하게 되는 것입니다.

시스템 로그

logger

사용자가 만들어낸 메세지를 시스템 로그(`/var/log/messages`)에 추가 시킵니다. 이 명령어는 일반 사용자도 쓸 수 있습니다.

```
logger Experiencing instability in network connection at 23:10, 05/21.
# 자, 이제 'tail /var/log/messages' 라고 해 보세요.
```

스크립트에 **logger** 명령어를 넣어서 디버깅 정보를 `/var/log/messages`에 쓸 수 있습니다.

```
logger -t $0 -i Logging at line "$LINENO".
# "-t" 옵션은 logger 엔트리용 태그를 지정합니다.
# "-i" 옵션은 프로세스 ID를 지정합니다.
```

```
# tail /var/log/message
# ...
# Jul  7 20:48:58 localhost ./test.sh[1712]: Logging at line 3.
```

logrotate

이 유틸리티는 시스템 로그 파일들을 적당하게 로테이트 시키고, 압축하고, 지우고, 메일을 보내는 일들을 처리해 줍니다. 보통 crond은 **logrotate**를 가장 기본적인 하루 일과로 삼습니다.

`/etc/logrotate.conf`에 적당한 내용을 적어주면 시스템 전체 로그뿐만 아니라 개인용 로그 파일을 관리할 수 있습니다.

작업 제어

ps

프로세스 통계(*P*rocess *S*tatistics): 현재 실행중인 프로세스들을 사용자와 PID(프로세스 아이디)에 의해서 보여줌. 보통은 `ax` 옵션을 줘서 부르고, grep이나 sed로 파이프를 걸어서 특정 프로세스를 찾습니다(예 11-8와 예 28-1 참고).

```
bash$ ps ax | grep sendmail
295 ?    S   0:00 sendmail: accepting connections on port 25
```

pstree

현재 실행중인 프로세스를 "나무"(tree) 형태로 보여 줍니다. `-p` 옵션을 주면 프로세스 이름뿐만 아니라 PID까지 보여 줍니다.

top

cpu를 집중적으로 사용하는 프로세스를 중심으로 최신 정보를 계속 보여줍니다. `-b` 옵션은 결과를 텍스트 모드로 보여주기 때문에 파싱을 하거나 스크립트에서 접근할 수가 있습니다.

```
bash$ top -b
  8:30pm  up 3 min,  3 users,  load average: 0.49, 0.32, 0.13
45 processes: 44 sleeping, 1 running, 0 zombie, 0 stopped
CPU states: 13.6% user,  7.3% system,  0.0% nice, 78.9% idle
Mem:    78396K av,   65468K used,   12928K free,       0K shrd,    2352K buff
Swap:  157208K av,       0K used,  157208K free                   37244K cached

  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
  848 bozo      17   0   996  996   800 R     5.6  1.2   0:00 top
    1 root       8   0   512  512   444 S     0.0  0.6   0:04 init
    2 root       9   0     0    0     0 SW    0.0  0.0   0:00 keventd
  ...
```

nice

백그라운드 작업의 우선순위를 바꿔줍니다. 우선순위는 19(제일 낮음)에서 -20(제일 높음)까지 인데, 오직 **root**만이 음수(높은) 우선순위를 줄 수 있습니다. 관련 명령어로는 **renice**, **snice**, **skill**이 있습니다.

nohup

사용자가 로그 아웃을 하더라고 명령어가 계속 돌게 해 줍니다. 명령어에 &를 붙여 실행하지 않으면 포그라운드로 실행이 될 것입니다. **nohup**을 스크립트에서 쓸 때는, 고아 프로세스나 좀비 프로세스가 생기지 않도록 wait과 같이 써야 합니다.

pidof

실행중인 작업의 프로세스 **ID**(pid)를 식별해 줍니다. kill이나 **renice**같은 작업 제어 명령어들은 프로세스 이름이 아니라 **pid**에 대해 동작하기 때문에 종종 **pid**로 구분할 필요가 생깁니다. **pidof** 명령어는 내부 변수인 $PPID와 거의 쌍을 이룹니다.

```
bash$ pidof xclock
880
```

예 **13-4. pidof** 로 프로세스를 죽이기

```
#!/bin/bash
# kill-process.sh

NOPROCESS=2

process=xxxyyyzzz  # 존재하지 않는 프로세스를 가지고,
# 그냥 데모용임...
# ... 실제로 돌고 있는 어떤 프로세스도 죽이려고 하는게 아니니까.
#
#  하지만, 예를 들어 인터넷에서 로그오프하고 싶다면
#+      process=pppd
#+ 됩니다.

t=`pidof $process`       # $process의 pid(프로세스 ID)를 찾고,
# 'kill'은 프로그램 이름이 아니라 pid를 쓰기 때문에

if [ -z "$t" ]           # 해당 프로세스가 없다면 'pidof'는 널을 리턴함.
then
  echo "$process 는 현재 실행중이 아니므로 그냥 종료합니다."
  exit $NOPROCESS
fi

kill $t                  # 잘 죽지 않는 프로세스라면 'kill -9'라고 해야 할지
도 모릅니다.
```

```
# 죽지 않게 돼 있는 프로세스일수도 있기 때문에
# 다시 한 번 " t=`pidof $process` " 로 확인해 볼 필요가 있습니다.


# 위 전체 스크립트는
#    kill $(pidof -x process_name)
# 이라고 할 수도 있겠으나 그러면 교육적이지는 않은것 같군요.

exit 0
```

fuser

어떤 파일이나, 파일 집합, 디렉토리에 접근하고 있는 프로세스를 PID로 식별해 줍니다. `-k` 옵션을 쓰면 해당 프로세스를 죽일 수 있습니다. 이 명령어는 시스템 보안 차원에서 아주 흥미로운 구현인데 주로 스크립트에서 쓰여 시스템 서비스에 대해 허가 받지 않은 사용자의 접근을 막는 용도로 쓰입니다.

crond

시스템 관리용 스케쥴러 프로그램으로서, 시스템 로그 파일을 정리하고 지운다거나 slocate 데이타 베이스를 업데이트 하는 등의 일을 해 줍니다. at의 루트 사용자 버전용 명령어입니다(물론, 각 사용자는 **crontab** 명령어를 써서 자신만의 `crontab` 파일을 가질수도 있습니다). 데몬으로 돌면서 `/etc/crontab`의 내용들을 스케쥴에 따라 실행시켜 줍니다.

프로세스 제어및 부팅

init

init 명령어는 모든 프로세스의 부모 프로세스로서, 시스템 부팅 과정의 제일 마지막에 불리면서 `/etc/inittab`을 읽어서 시스템의 런레벨을 결정합니다. 오직 루트만이 별명인 **telinit**으로 부를 수 있습니다.

telinit

init를 가르키는 심볼릭링크로서, 시스템 런레벨을 바꿀 때 쓰는데 보통은 시스템 관리나 긴급하게 파일시스템을 수리해야 할 때 씁니다. 오직 루트만 이 명령어를 쓸 수 있습니다. 이 명령어는 아주 위험하기 때문에 쓰기 전에 이 명령어를 잘 이해하고 있어야 합니다!

runlevel

현재와 바로 전의 런레벨을, 시스템이 정지 상태인지(런레벨 0), 단일 사용자 모드인지(1), 다중 사용자 모드인지(2나 3), X 윈도우 모드인지(5), 리부팅 중인지(6)등으로 보여줍니다. 이 명령어는 `/var/run/utmp` 파일을 통해 정보를 얻어 옵니다.

halt
shutdown

reboot

보통 시스템 전원을 끄기 전에 시스템을 정지시키는 명령어들.

네트워크

ifconfig

네트워크 인터페이스 설정및 튜닝 유틸리티. 이 명령어는 부팅시 인터페이스를 설정할 때나 리부팅때 인터페이스를 내리기 위해 쓰입니다.

```
# /etc/rc.d/init.d/network 의 일부분

# ...

# 네트워킹이 가능한지 확인.
[ ${NETWORKING} = "no" ] && exit 0

[ -x /sbin/ifconfig ] || exit 0

# ...

for i in $interfaces ; do
  if ifconfig $i 2>/dev/null | grep -q "UP" >/dev/null 2>&1 ; then
    action "Shutting down interface $i: " ./ifdown $i boot
  fi
# "grep"의 GNU 전용인 "-q" 옵션은 "quiet"를 뜻하고, 어떤 출력도 하지 않게 합니
다.
# 따라서 출력을 /dev/null 로 재지향 하는 것이 꼭 필요하지 않습니다.

# ...

echo "현재 동작중인 디바이스:"
echo `/sbin/ifconfig | grep ^[a-z] | awk '{print $1}'`
#                            ^^^^^  globbing 을 막기 위해 쿼우트 시켜야 합
니다.
#  다음도 역시 동작합니다.
#    echo $(/sbin/ifconfig | awk '/^[a-z]/ { print $1 })'
#    echo $(/sbin/ifconfig | sed -e 's/ .*//')
#  S.C.가 주석을 더 넣어 줬습니다. 고마워요.
```

예 30-5 참고.

route

커널 라우팅 테이블 정보를 보거나 바꿀 수 있게 해 줍니다.

```
bash$ route
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
pm3-67.bozosisp *               255.255.255.255 UH       40 0          0 ppp0
127.0.0.0       *               255.0.0.0       U        40 0          0 lo
```

```
default         pm3-67.bozosisp 0.0.0.0         UG       40 0          0 ppp0
```

chkconfig

네트워크 설정을 체크해줌. 이 명령어는 `/etc/rc?.d` 디렉토리에 들어있고 부팅시 시작되는 네트워크 서비스들을 보여주고 관리해 줍니다.

원래는 IRIX에 있던 것을 레드햇 리눅스가 포팅한 것으로 다른 리눅스 배포판에서는 기본 설치에 속하지 않을 수도 있습니다.

```
bash$ chkconfig --list
atd             0:off   1:off   2:off   3:on    4:on    5:on    6:off
rwhod           0:off   1:off   2:off   3:off   4:off   5:off   6:off
...
```

tcpdump

네트워크 패킷 "스니퍼". 주어진 기준에 맞는 패킷 헤더의 덤프를 떠서 네트워크 트래픽을 분석하고 문제점을 해결할 수 있게 해 줍니다.

bozoville 와 **caduceus** 두 호스트간의 IP 패킷 트래픽을 덤프:

```
bash$ tcpdump ip host bozoville and caduceus
```

당연히 **tcpdump**의 출력은 앞에서 논의했던 텍스트 처리 유틸리티들을 이용해서 파싱할 수가 있습니다.

파일시스템

mount

파일시스템을 마운트해 줍니다. 보통은 플로피나 시디롬 같은 외부 디바이스에 대해서 쓰입니다. `/etc/fstab`에 가능한 파일시스템이나 파티션, 디바이스, 옵션등을 적어 놓으면 자동이나 수동으로 마운트를 편하게 할 수 있습니다. `/etc/mtab` 파일은 `/proc` 같은 가상 파일시스템도 포함해서 현재 마운트 되어 있는 파일 시스템을 보여 줍니다.

mount -a 는 `/etc/fstab`에 들어 있는 파일 시스템과 파티션중에 `noauto` 옵션이 있는 항목만 빼고 모두 마운트 해 줍니다. 부팅될 때, 모든 파티션이 마운트 되도록 `/etc/rc.d` 디렉토리에 들어 있는 시스템 구동 스크립트(`rc.sysinit`이나 비슷한 것)에서 이 명령어를 부릅니다.

```
mount -t iso9660 /dev/cdrom /mnt/cdrom
# CDROM 마운트
mount /mnt/cdrom
# /mnt/cdrom 이 /etc/fstab 에 들어 있을 경우 짧게 부르기
```

이 다재다능한 명령어는 보통 파일을 블럭 디바이스에 존재하는 파일 시스템처럼 마운트 할 수도 있습니다. 이런 능력은 루프백 디바이스(loopback device)라고 하는 파일을 이용해서 가능해 집니다. 이 루프백 디바이스를 적용한 예로서, ISO9660 이미지를 CDR로 굽기 전에 마운트해서 테스트 해보는 것이 있습니다. [3]

예 **13-5. CD** 이미지 확인하기

```
# 루트로...

mkdir /mnt/cdtest  # 마운트 포인트가 없다면 준비함.

mount -r -t iso9660 -o loop cd-image.iso /mnt/cdtest   # 이미지 마운트.
#                  "-o loop" 옵션은 "losetup /dev/loop0" 와 같음.
cd /mnt/cdtest     # 이제 이미지를 확인.
ls -alR            # 이미지에 들어있는 디렉토리 트리에 들어 있는 파일들을 나열.
                   # 기타 등등...
```

umount

현재 마운트 되어 있는 파일 시스템을 언마운트 해 줍니다. 이미 마운트 되어 있는 플로피나 시디롬 디스크를 빼기 전에 꼭 **umount**를 해 줘야 합니다. 안 그러면 파일 시스템이 깨질 수도 있습니다.

```
umount /mnt/cdrom
# 이제 이젝트 버튼을 눌러 디스크를 안전하게 뺄 수 있습니다.
```

> 참고**: automount** 유틸리티가 적절하게 설치되어 있다면 플로피나 시디롬 디스크에 접근시나 제거시에 자동으로 마운트와 언마운트를 할 수 있습니다. 플로피나 시디롬 드라이브를 꼈다 뺐다 할 수 있는 랩탑에서는 문제를 일으킬 수도 있습니다.

sync

버퍼에 들어 있는 최신 데이타를 하드 드라이브로 즉시 쓰게 합니다(버퍼와 드라이브를 동기화). 이 명령어가 꼭 필요한 것은 아니지만 시스템 관리자나 사용자에게 자신들이 변경한 데이타가 갑작스런 전원 이상에도 살아남을 수 있게 해 줍니다. 예전에는 **`sync; sync`**(아주 확실히 하기 위해서 두 번 내림)라고 해서 시스템을 리부팅하기 전의 유용한 예방책으로 쓰였습니다.

파일을 안전하게 지우거나(예 12-33) 천장의 전등이 깜빡이기 시작했을 때 버퍼를 즉시 플러쉬시키고 싶을 때가 있을지도 모릅니다.

losetup

루프백 디바이스를 설정해 줍니다.

예 **13-6.** 한 파일에서 한번에 파일 시스템 만들기

```
SIZE=1000000  # 1 meg

head -c $SIZE < /dev/zero > file  # 지정된 크기로 파일을 설정.
losetup /dev/loop0 file           # 루프백 디바이스로 설정.
mke2fs /dev/loop0                 # 파일 시스템 만들기.
mount -o loop /dev/loop0 /mnt     # 마운트.

# Thanks, S.C.
```

mkswap

스왑 파티션이나 스왑 파일을 만들어 줍니다. 이 명령어 다음에는 꼭 **swapon**으로 활성화를 시켜줘야 합니다.

swapon
swapoff

스왑 파티션이나 스왑 파일을 활성화/비활성화 시켜 줍니다. 이 명령어는 보통 부팅시나 셧다운시에 효력을 갖습니다.

mke2fs

리눅스 ext2 파일시스템을 만들어 줍니다. 이 명령어는 루트로 실행 시켜야 합니다.

예 **13-7.** 새 하드 드라이브 추가하기

```
#!/bin/bash

# 시스템에 두 번째 하드 드라이브 추가하기.
# 소프트웨어 설정. 하드웨어가 이미 마운트돼 있다고 가정함.
# 본 문서의 저자가 "Linux Gazett", http://www.linuxgazette.com, 38호에
# 쓴 기사에서 발췌.

ROOT_UID=0     # 이 스크립트는 루트로 실행 시켜야 됩니다.
E_NOTROOT=67   # root 가 아닌 경우의 종료 에러.

if [ "$UID" -ne "$ROOT_UID" ]
then
  echo "이 스크립트는 루트만 실행시킬 수 있습니다."
  exit $E_NOTROOT
fi

# 이 스크립트는 정말 주의해서 쓰기 바랍니다!
# 만약 무언가가 잘못된다면 여러분의 파일 시스템을 홀라당 날려먹을 수 있습니다.


NEWDISK=/dev/hdb         # /dev/hdb 가 비어 있다고 가정함. 꼭 확인해 볼 것!
MOUNTPOINT=/mnt/newdisk  # 아니면 다른 마운트 포인트 지정.
```

```
fdisk $NEWDISK
mke2fs -cv $NEWDISK1   # 배드 블럭 확인및 자세한 출력.
#  주의:    /dev/hdb 가 *아니라* /dev/hdb1 입니다!
mkdir $MOUNTPOINT
chmod 777 $MOUNTPOINT  # 새 드라이브는 모든 사용자가 접근할 수 있도록 함.


# 자, 테스트를 해 보죠.
# mount -t ext2 /dev/hdb1 /mnt/newdisk
# 디렉토리를 만들어 보고 잘 된다면 umount 한 다음 하던 일을 계속하면 됩니다.

# 마지막 단계:
# 다음을 /etc/fstab 에 추가해 주세요.
# /dev/hdb1  /mnt/newdisk  ext2  defaults  1 1

exit 0
```

예 13-6 와 예 29-3 도 참고하세요.

tune2fs

ext2 파일 시스템을 튜닝해 줍니다. 최대 마운트 숫자같은 파일 시스템 매개변수를 바꾸는데 쓰일 수 있습니다. 루트로 실행해야 됩니다.

> 주의
>
> 이 명령어는 굉장히 위험합니다. 부주의하게 쓴다면 여러분 파일 시스템을 박살낼 수도 있기 때문에 여러분 스스로 책임을 지고 써야 합니다.

dumpe2fs

아주 자세한 파일 시스템 정보를 표준출력으로 덤프해 줍니다. 루트로 실행되야 합니다.

```
root# dumpe2fs /dev/hda7 | grep 'ount count'
dumpe2fs 1.19, 13-Jul-2000 for EXT2 FS 0.5b, 95/08/09
 Mount count:              6
 Maximum mount count:      20
```

hdparm

하드 디스크 매개변수를 보여주거나 바꿀 수 있습니다. 루트로 실행시켜야 되고 잘못 쓸 경우 위험할 수 있습니다.

fdisk

보통은 하드 드라이브일, 저장용 디바이스에 대해서 파티션 테이블을 만들고 변경할 수 있게 해 줍니다. 루트로 실행해야 됩니다.

주의

이 명령어는 아주 조심해서 써야 됩니다. 만약에 뭔가가 잘못되면 여러분의 파일 시스템을 망가트릴 수도 있습니다.

fsck
e2fsck
debugfs

파일 시스템 체크, 치료, 디버그용 명령어들.

fsck: 유닉스 파일 시스템을 체크해 주는 프론트 엔드(front end)로서 다른 유틸리티가 이 명령어를 부름. 실제로 체크할 파일 시스템 타입은 ext2가 기본으로 잡혀 있습니다.

e2fsck: ext2 파일 시스템 체커.

debugfs: ext2 파일 시스템 디버거.

경고

이 명령어들은 루트로 실행시켜야 합니다. 잘못 쓰면 파일 시스템이 손상되거나 망가질 수도 있습니다.

badblocks

저장용 디바이스에 대해 배드 블럭(미디어의 물리적인 결함)을 체크해 줍니다. 새 하드 드라이브를 설치하고 포맷할 때나 백업 미디어의 무결성을 테스트 해보려고 할 때 쓸 수 있습니다. [4] 예를 들어 **badblocks /dev/fd0** 라고 하면 플로피 디스크를 테스트해 줍니다.

badblocks 명령어는 모든 데이타를 다 덮어써버리도록 불릴 수도 있고 읽기 전용 모드로 안전하게 불릴 수도 있습니다. 루트 사용자가 테스트할 디바이스를 소유하고 있다면, 보통 그런 상황일테지만, 루트가 이 명령어를 실행시켜야 합니다.

mkbootdisk

예를 들어 MBR(master boot record)이 깨진 상황등에서 시스템을 다시 살릴 수 있도록 부팅 디스켓을 만들어 줍니다. **mkbootdisk**는 실제로는 Erik Troan이 만든 Bash 스크립트로 `/sbin` 디렉토리에 들어 있습니다.

chroot

루트 디렉토리를 바꿔줍니다(CHange ROOT directory). 명령어들은 보통 기본 루트 디렉토리인 `/`를 기준으로 $PATH에 따라 해석됩니다. 이 명령어는 이 루트 디렉토리를 다른 곳으로 바꾼다음 작업 디렉토리도 그 쪽으로 바꿔줍니다. 보안용으로 아주 유용한데, 예를 들면 시스템 관리자가 텔넷으로 접속중인 사용자를 보안과 관련된 파일 시스템에 접근시키지 않으려 할 때에 사용할 수 있습니다(가끔 guest 사용자를 "chroot 감옥"(chroot jail)에 가둔다라고도 합니다). **chroot**후에는 시스템 바이너리에 대한 실행 경로가 더 이상 맞지 않는 것에 주의하기 바랍니다.

`chroot /opt` 라고 하면 `/usr/bin`을 `/opt/usr/bin`이라고 해석합니다. 비슷하게, 보통의 상황에서는 `/`를 기본 디렉토리로 삼지만 **`chroot /aaa/bbb /bin/ls`** 라고 하면 그 후로 실행되는 **ls**는 `/aaa/bbb`을 기본 디렉토리로 인식합니다. 사용자의 `~/.bashrc`에 **alias XX 'chroot /aaa/bbb ls'** 라는 줄을 넣으면 그 사용자가 "XX"를 실행 시키는 파일 시스템을 효과적으로 제한할 수 있습니다.

chroot는 비상용 부트 플로피로 부팅했을 때 편하게 쓸 수 있고(`/dev/fd0`에 **chroot** 걸기), 시스템이 박살나서 고치려고 할 때 **lilo**에 옵션으로 줄 수도 있습니다. 또한 다른 파일 시스템을 통해서 설치를 할 경우나(**rpm** 옵션), CDROM 같은 읽기 전용 파일 시스템에서 실행할 때 쓸 수 있습니다. 오직 루트 사용자로 실행할 수 있으며, 조심해서 써야 합니다.

> ### 경고
>
> 몇 개의 시스템 파일들은 **chroot**된 디렉토리에 복사해 놓아야 `$PATH`가 그들을 제대로 인식할 수 있습니다.

lockfile

이 유틸리티는 **procmail** 패키지 중의 하나입니다(www.procmail.org (http://www.procmail.org)). 이 명령어는 잠금 파일을 만들어 주는데, 잠금 파일이란 특정 파일이나 디바이스, 리소스에 대해서 접근 제어를 해주는 세마포어 파일입니다. 잠금 파일은 이 특정한 파일, 디바이스, 리소스를 특정 프로세스가 쓰고 있다("busy")는 플래그로 쓰여, 다른 프로세스에게 제한된 접근만을 허용하거나 아예 접근을 못하도록 만들어 줍니다.

잠금 파일은 여러 사용자에 의해 시스템 메일 폴더가 동시에 변경되는 것을 막아주고, 모뎀 포트가 사용중이라는 것을 알려주며, 넷스케이프가 캐쉬를 사용중이라는 것을 보여주는데 쓰일 수 있습니다. 스크립트에서 특정 프로세스가 만들어 놓은 잠금 파일이 있는지 확인해서 그 프로세스가 이미 떠 있는지 알아낼 수도 있습니다. 만약에 스크립트에서 이미 존재하는 잠금 파일을 다시 만들려고 한다면 그 스크립트는 멈춰 버릴 수도 있으니 조심하기 바랍니다.

보통의 어플리케이션들은 잠금 파일을 만들고 체크하는 기본 디렉토리를 `/var/lock`으로 삼습니다. 다음 스크립트처럼 해서 잠금 파일이 있는지 없는지를 확인할 수 있습니다.

```
appname=xyzip
# "xyzip" 어플리케이션은 "/var/lock/xyzip.lock" 란 잠금 파일을 만듭니다.

if [ -e "/var/lock/$appname.lock ]
then
  ...
```

mknod

블럭 디바이스나 문자 디바이스 파일을 만들어 줍니다(예를 들면 하드웨어를 새로 설치할 경우에 필요하겠죠?).

tmpwatch

특정 기간동안 접근이 없는 파일을 자동으로 지워줍니다. 보통은 오래된 로그 파일을 지우기 위해 crond에 걸어 놓고 씁니다.

MAKEDEV

디바이스 파일을 만들어 주는 유틸리티로서, 루트로 실행시켜야 합니다. 이 명령어는 `/dev` 디렉토리에 있습니다.

```
root# ./MAKEDEV
```

이 명령어는 일종의 **mknod**의 향상된 버전입니다.

백업

dump
restore

dump 명령어는 복잡한 파일 시스템 백업 유틸리티로서 보통은 규모가 큰 설치와 네트워크에서 쓰입니다.[5] 디스크 파티션을 있는 그대로(raw) 읽고 바이너리 형태로 백업 파일을 만들어 냅니다. 백업 되는 파일들은 디스크나 테입 드라이브같은 다양한 저장 미디어로 저장됩니다. **restore** 명령어는 **dump**로 백업된 파일들을 복구시켜 줍니다.

fdformat

플로피 디스크에 대해서 로우레벨 포맷을 해 줍니다.

시스템 리소스

ulimit

시스템 리소스에 대해서 최대 한계(upper limit)를 지정해 줍니다. 보통 `-f` 옵션을 써서 쉘이 만들 수 있는 파일 크기를 제한 시킵니다(**ulimit -f 1000** 이라고 하면 파일 크기를 1 메가로 잡아 줍니다). `-t` 옵션은 코어덤프 파일의 크기를 제한 시킵니다(**ulimit -c 0** 이라고 하면 코어덤프를 생성시키지 않습니다). **ulimit** 값은 보통, `/etc/profile`이나 `~/.bash_profile`에서 지정해 줍니다(27장 참고).

umask

사용자(User) 파일 생성 마스크(MASK). 사용자 각자의 기본 파일 속성을 제한해 줍니다. 사용자가 생성하는 모든 파일은 **umask**로 지정된 속성의 영향을 받습니다. **umask**로 넘겨준 값은 해당 파일 소유권을 꺼버립니다. 예를 들어, **umask 022** 는 새로 만들어 지는 파일이 최소한 755 소유권을 갖도록 해 줍니다(777 NAND 022).[6] 사용자는 당연히 나중에 그 파일의 속성을 chmod로 바꿀 수 있습니다. 보통 **umask** 값을 설정할 때는 `/etc/profile`이나 `~/.bash_profile`에서 합니다(27장 참고).

rdev

루트 디바이스, 스왑 영역, 비디오 모드에 대한 정보를 얻거나 변경. **rdev**의 기능은 **lilo**로 넘어갔지만 아직 램 디스크를 설정할 때는 유용합니다. 역시 잘못 쓰면 위험한 명령어입니다.

모듈

lsmod

설치된 커널 모듈을 보여줍니다.

```
bash$ lsmod
Module                  Size  Used by
autofs                  9456   2 (autoclean)
opl3                   11376   0
serial_cs               5456   0 (unused)
sb                     34752   0
uart401                 6384   0 [sb]
sound                  58368   0 [opl3 sb uart401]
soundlow                 464   0 [sound]
soundcore               2800   6 [sb sound]
ds                      6448   2 [serial_cs]
i82365                 22928   2
pcmcia_core            45984   0 [serial_cs ds i82365]
```

insmod

커널 모듈을 강제로 올립니다. 루트로 실행해야 합니다.

modprobe

보통 시스템 구동 스크립트에서 자동으로 불리는 모듈 로더.

depmod

모듈간 의존 파일을 만들어 줍니다. 보통 시스템 구동 스크립트에서 불립니다.

기타 명령어들

env

현재의 환경 변수, 혹은 그 값을 바꿔 프로그램이나 스크립트를 실행 시킵니다(시스템 전체 환경은 건드리지 않습니다). `[varname=xxx]` 옵션을 주면 스크립트가 실행될 동안에만 환경 변수 `varname`의 값을 바꿔줍니다. 아무 옵션도 안 주면 현재 세팅되어 있는 모든 환경 변수를 보여줍니다.

참고: Bash나 본쉘에서 파생된 다른 쉘들에서는 단일 명령어 환경에서 변수를 설정하는 것이 가능합니다.

```
var1=value1 var2=value2 commandXXX
# 'commandXXX' 의 환경에서만 $var1 과 $var2 를 설정.
```

작은 정보: 스크립트의 첫번째 줄("#!"가 있는 줄)에 **env**를 써서 경로를 모르는 쉘이나 명령어 해석기를 지정해 줄 수도 있습니다.

```
#! /usr/bin/env perl

print "펄이 어디에 있는지 몰라도\n";
print "이 펄 스크립트는 잘 동작할 것입니다.\n";

# 펄 실행 파일이 원하는 곳에 없을 수도 있기 때문에
# 이식성 있는 크로스 플랫폼용 스크립트에 아주 좋습니다.
# Thanks, S.C.
```

ldd

실행 파일에 필요한 공유 라이브러리를 보여줍니다.

```
bash$ ldd /bin/ls
libc.so.6 => /lib/libc.so.6 (0x4000c000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x80000000)
```

strip

실행 가능한 바이너리 파일에서 디버깅용 심볼릭 참조 정보를 제거해 줍니다. 이 명령어를 쓰면 실행 파일의 크기는 작아지지만 디버깅은 할 수가 없습니다.

주로 Makefile에서나 나오고 쉘 스크립트에서는 잘 쓰이지 않습니다.

nm

strip 되지 않은 컴파일된 바이너리에 들어 있는 심볼들을 보여줍니다.

rdist

리모트 분산 클라이언트: 리모트 서버에 있는 파일 시스템으로 동기화, 복사, 백업을 해줍니다.

지금까지 배운 관리자용 명령어들에 대한 지식을 가지고 시스템 스크립트를 살펴 보도록 하겠습니다. **killall**은 시스템 셧다운시에, 돌고 있는 프로세스를 멈추게 해 주는 프로세스로서, 짧으면서도 이해하기 쉬운 스크립트중의 하나입니다.

예 13-8. killall, `/etc/rc .d/init.d` 에서 인용

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

```
# --> "# -->" 표시는 본 문서의 저자가 붙인 주석입니다.

# --> 여기서 소개하는 부분은
# --> Miquel van Smoorenburg(<miquels@drinkel.nl.mugnet.org>)의
# --> 'rc' 스크립트 패키지중의 일부입니다.

# --> 이 스크립트는 레드햇 전용 스크립트인 것처럼 보이기 때문에
# --> 다른 배포판에는 없을 수도 있습니다.

# 쓰이지 않으면서 실행중인 모든 서비스를 죽입니다(실제로 실행중인지를
# 확실히 확인하기 때문에 아무 서비스도 안 죽일 수 있습니다).

for i in /var/lock/subsys/*; do
        # --> 표준 for/in 루프이나 "do" 가 같은 줄에 있기 때문에
        # --> ";" 를 붙여줘야 됩니다.
# 스크립트가 실제로 있는지 확인.
[ ! -f $i ] && continue
# --> "and list"를 아주 잘 썼습니다. 다음처럼 해도 똑같습니다.
# --> if [ ! -f "$i" ]; then continue

# 서브시스템 이름을 알아냅니다.
subsys=${i#/var/lock/subsys/}
# --> 변수 이름 매칭인데 여기서는 파일 이름이 되겠죠.
# --> subsys=`basename $i` 와 완전히 똑같은 표현입니다.

# --> 잠금 파일의 이름을 알아내는데, 잠금 파일이 있다면
# --> 해당 프로세스가 실행중이라는 증거입니다.
# --> 앞에서 설명했던 "lockfile" 을 참고하세요.


# 그 서브시스템을 내립니다.
if [ -f /etc/rc.d/init.d/$subsys.init ]; then
    /etc/rc.d/init.d/$subsys.init stop
else
    /etc/rc.d/init.d/$subsys stop
# --> 쉘 내장명령인 'stop'을 써서 돌고 있는 작업과 데몬을 중지시킵니다.
fi
done
```

별로 어려워 보이지 않습니다. 변수 매칭에 대한 부분만 빼고 앞에서 다 배운 것들입니다.

연습문제 **1.** `/etc/rc.d/init.d` 에 있는 **halt** 스크립트를 분석해 보세요. **killall**보다 약간 더 길지만 개념은 같습니다. 자신의 홈 디렉토리 어디쯤에 복사해 놓고 이것 저것 실험해 보세요.(루트로 실행시키면 안 됩니다). `-vn` 옵션을 써서 가상 모드로 실행시켜 보세요(**`sh -vn scriptname`**). 주석을 꼼꼼하게 달아 보세요. “action” 명령어를 “echo” 명령어로 바꿔 보세요.

연습문제 **2.** 자 이제, `/etc/rc.d/init.d` 에서 더 복잡한 스크립트들을 살펴 보세요. 거기에 있는 스크립트들을 이해할 수 있나요? 방금 위에서 했던 방법을 써서 분석해 보세요. "initscripts" 문서의 일부분인 `/usr/share/doc/initscripts-?.??` 에 있는 `sysvinitfiles`을 분석해서 통찰력을 기르세요.

주석

1. 디스크 쿼타가 걸려 있는 리눅스 머신이나 유닉스 시스템인 경우에 적용됩니다.
2. **userdel**은 지울 사용자가 로그인중일 때는 실패합니다.
3. CDR 굽기에 대한 자세한 정보는 Alex Wither 가 리눅스 저널 (http://www.linuxjournal.com)(Linux Journal)에 1999년 10월에 쓴 CD 만들기 (http://www2.linuxjournal.com/lj-issues/issue66/3335.html)(Creating CDs) 기사를 읽어 보기 바랍니다.
4. mke2fs에 `-c` 옵션을 줘도 배드 블럭을 체크해 줍니다.
5. 단일 사용자 리눅스 시스템은 일반적으로 **tar**처럼 간단한 명령어로 백업을 합니다.
6. NAND는 "not-and" 논리 연산자 입니다. 빼기 연산과 비슷한 개념으로 보면 됩니다.

14장. 명령어 치환(Command Substitution)

명령어 치환은 하나나 그 이상의 명령어의 출력을 재할당 해줍니다. 명령어 치환은 말그대로 한 명령어의 출력을 다른 문맥으로 연결해 줍니다.

명령어 치환의 전형적인 형태는 역따옴표(‘...‘)를 쓰는 것입니다. 역따옴표 안에 들어 있는 명령어는 명령어 줄에서 쓸 수 있는 텍스트를 만들어 냅니다.

```
script_name=`basename $0`
echo "이 스크립트의 이름은 $script_name 입니다."
```

명령어의 출력은 다른 명령어의 인자로 쓸 수 있는데, 변수를 설정하거나 **for** 루프에서 인자 리스트로도 쓸 수 있습니다.

```
rm `cat filename`   # “filename” 은 지울 파일 목록을 갖고 있습니다.
#
# S. C. 가 "arg list too long" 이란 에러가 나올 수도 있다고 지적했습니다.
# 더 좋은 방법             xargs rm -- < filename
# ( -- 는 “-”로 시작하는 “filename”도 처리해 줍니다. )

textfile_listing=`ls *.txt`
# 현재 디렉토리의 모든 *.txt 파일의 이름을 담고 있는 변수.
echo $textfile_listing

textfile_listing2=$(ls *.txt)   # 명령어 치환의 다른 형태.
echo $textfile_listing
# 똑같은 결과.

# 파일 목록을 하나의 문자열로 가져가면 뉴라인 문자가 중간에 들어가는
# 문제가 생길 수도 있습니다.
#
# 파일 목록을 인자로 지정하는 안전한 방법은 배열을 사용하는 것입니다.
#      shopt -s nullglob    # 일치하는게 없다면 파일명 확장은 무의미해 집니다.
#      textfile_listing=( *.txt )
#
# Thanks, S.C.
```

경고

명령어 치환은 낱말 조각남(word splitting)이 생길수도 있습니다.

```
COMMAND `echo a b`      # 2개의 인자: a 와 b

COMMAND "`echo a b`"    # 1개의 인자: "a b"

COMMAND `echo`          # 인자 없음

COMMAND "`echo`"        # 하나짜리 빈 인자


# Thanks, S.C.
```

경고

명령어 치환에서 낱말 조각남은 재할당되는 명령어의 출력에서 뉴라인 문자들을 지워 버려서 유쾌하지 않은 결과를 가져 올 수 있습니다.

```
dir_listing=`ls -l`
echo $dirlisting

# 다음처럼 멋지게 정렬된 디렉토리 목록을 바라겠지만
# -rw-rw-r--
1 bozo       30 May 13 17:15 1.txt
# -rw-rw-r--
1 bozo       51 May 15 20:57 t2.sh
# -rwxr-xr-
x    1 bozo      217 Mar  5 21:13 wi.sh

# 실제로 여러분은 이런 결과를 보게 됩니다.:
# total 3 -rw-rw-r--
1 bozo bozo 30 May 13 17:15 1.txt -rw-rw-r--
1 bozo
# bozo 51 May 15 20:57 t2.sh -rwxr-xr-
x 1 bozo bozo 217 Mar 5 21:13 wi.sh

# 뉴라인 문자가 사라져 버렸습니다.
```

명령어 치환에서 낱말 조각남이 안 생긴다 하더라도 뉴라인 문자를 지워버릴 수 있습니다.

```
# cd "`pwd`"  # 이렇게 하면 항상 동작할 겁니다.
# 하지만...

mkdir ’제일 끝에 뉴라인이 있는 디렉토리
’

cd ’제일 끝에 뉴라인이 있는 디렉토리
’

cd "`pwd`"  # 에러 메세지:
# bash: cd: /tmp/file with trailing new-
line: No such file or directory

cd "$PWD"   # 잘 됩니다.
```

```
old_tty_setting=$(stty -g)   # 현재의 터미널 세팅
을 저장.
echo "키를 누르세요 "
stty -icanon -echo           # 터미널의 "캐노니
컬"(canonical) 모드를 끄고,
                             # "로컬" 에코도 끔.
key=$(dd bs=1 count=1 2> /dev/null)   # 키누름을 얻
기 위해 ’dd’를 씀.
stty "$old_tty_setting"      # 저장해 놓았던 세팅
을 복구.
echo "${#key} 개의 키를 눌렀습니
다."  # ${#variable} = $variable 에 들어 있는 문자수
#
# RETURN 말고 다른 키를 눌렀을 때의 출력은 "1 개의 키
를 눌렀습니다."
# RETURN 을 눌렀을 때의 출력은 "0 개의 출력을 눌렀습니
다."
# 명령어 치환이 뉴라인 문자를 먹어 버렸습니다.
```

작은 정보: 명령어 치환은 재지향을 써서 파일의 내용을 변수로 세팅하거나 cat 명령어를 써서 할 수 있게 해줍니다.

```
variable1=`<file1`      # "variable1" 을 "file1"의 내용으로 세트.
variable2=`cat file2`   # "variable2" 를 "file2"의 내용으로 세트.

#  변수에 공백문자가 포함된 값이 들어갈수도 있고
#+ 심지어는 제어 문자가 들어갈수도 있기 때문에 주의해야 합니다.
```

명령어 치환은 Bash 에서 쓸 수 있는 툴셋을 확장시켜 줄 수 있습니다. 표준출력으로 결과를 출력해 주는(잘 동작하는 유닉스 툴이 그래야 하는 것처럼) 프로그램이나 스크립트를 짜고 그 결과를 변수로 할당하면 됩니다.

```
#include <stdio.h>

/*  "Hello, world." C program  */

int main()
{
  printf( "Hello, world." );
  return (0);
}

bash$ gcc -o hello hello.c
```

```
#!/bin/bash
# hello.sh

greeting=`./hello`
echo $greeting

bash$ sh hello.sh
Hello, world.
```

참고: 이제는 역따옴표 대신 **$(COMMAND)** 형태의 명령어 치환이 쓰입니다.

```
output=$(sed -n /"$1"/p $file)
# "grp.sh" 예제에서.
```

쉘 스크립트에서 명령어 치환이 쓰이는 예제들:

1. 예 10-7
2. 예 10-24
3. 예 9-21
4. 예 12-2
5. 예 12-15
6. 예 12-12
7. 예 12-31
8. 예 10-12
9. 예 10-9
10. 예 12-24
11. 예 16-5
12. 예 A-12
13. 예 28-1
14. 예 12-28
15. 예 12-29
16. 예 12-30

15장. 산술 확장(Arithmetic Expansion)

산술 확장은 스크립트에서 산술 연산을 수행할 때 강력한 기능을 제공해 줍니다. 문자열을 산술식으로 변환하는 것은 역따옴표나 이중 소괄호, let을 써서 비교적 간단하게 수행할 수 있습니다.

산술 확장의 다양한 구현(Variations)

역따옴표로 산술 확장 하기(종종 expr과 같이 쓰입니다)

```
z=`expr $z + 3`          # 'expr' 이 확장을 해 줍니다.
```

let
과 이중 소괄호로 산술 확장 하기

이제는 산술 확장에서 역따옴표를 쓰지 않고 이중 소괄호(**`$((...))`**)나 아주 편한 **let**을 씁니다.

```
z=$(($z+3))
# $((EXPRESSION)) 는 산술 확장입니다.
# 명령어 치환과 헷갈리면 안 됩니다.

let z=z+3
let "z += 3"  # 쿼우트를 해 주면, 빈 칸를 비롯해서 특수한 연산자의 사용이 가능
해 집니다.
# 'let' 은 실제로는 산술 확장을 하지 않고 산술 평
가(arithmetic evaluation)를 합니다.
```

이상의 모든 것들은 동일합니다. "입맛에 맞게" 골라 쓰면 됩니다.

스크립트에서 산술 확장이 쓰이는 예제들:

1. 예 12-6
2. 예 10-13
3. 예 26-1
4. 예 26-4
5. 예 A-12

16장. I/O 재지향

쉘은 항상 기본적으로 표준입력(`stdin`, 키보드), 표준출력(`stdout`, 스크린), 표준에러(`stderr`, 스크린에 뿌려질 에러 메세지) “파일들”을 열어 놓습니다. 이 파일들을 포함해서 열려 있는 어떤 파일이라도 재지향 될 수 있습니다. 재지향이란 간단히 말해서 파일, 명령어, 프로그램, 스크립트, 심지어는 스크립트 속의 코드 블럭(예 4-1, 예 4-2 참고)의 출력을 낚아 채서 다른 파일, 명령어, 프로그램, 스크립트의 입력으로 보내는 것입니다.

열려 있는 파일 각각은 파일 디스크립터(file descriptor)를 할당 받습니다. [1] 표준입력, 표준출력, 표준에러에 해당하는 파일 디스크립터는 각각 0, 1, 2 입니다. 추가적으로 열리는 파일을 위해서 3부터 9까지의 파일 디스크립터가 남겨져 있습니다. 종종, 이 추가적인 파일 디스크립터들중의 하나를 표준입력, 표준출력, 표준에러로 할당해서 임시적인 중복된 링크로 쓰는 것이 유용할 때가 있습니다. [2] 이런 방법을 쓰면 아주 복잡한 재지향이나 파일 디스크립터를 뒤죽 박죽 사용했을 때, 아주 간단하게 원래대로 복구시켜 줍니다(예 16-1 참고).

```
>
   # 표준출력을 파일로 재지향.
   # 파일이 없으면 새로 만들고, 있다면 덮어 씁니다.

   ls -lR > dir-tree.list
   # 디렉토리 트리 목록을 파일로 저장해 줍니다.

: > filename
   # > 는 "filename"의 길이가 0 이 되도록 잘라줍니다.
   # : 는 아무 출력도 안 하는 더미 플레이스홀더(placeholder)로 동작합니다.

>>
   # 표준출력을 파일로 재지향.
   # 파일이 없으면 새로 만들고, 있으면 파일 끝에 덧붙입니다.

2>&1
   # 표준에러를 표준출력으로 재지향.
   # 에러 메세지는 표준 출력의 자격으로 스크린에 보내집니다.

i>&j
   # i번 파일 디스크립터를 j번 파일디스크립터로 재지향.
   # i가 가르키는 파일의 모든 출력은 j가 가르키는 파일로 보내집니다.

>&j
   # 기본적으로 1번 파일 디스크립터(표준출력)를 j번 파일 디스크립터로 재지향.
   # 모든 표준출력은 j가 가르키는 파일로 보내집니다.

0<
 <
   # 파일에서 입력을 받도록 해줍니다.
   # “>”와 짝을 이루는 명령어로, 종종 같이 쓰입니다.
   #
```

```
    # grep search-word <filename
```

```
 [j]<>filename
    # "filename"을 읽고 쓰기용으로 열고 "j"번 파일 디스크립터를 할당합니다.
    # "filename"이 없다면 새로 만듭니다.
    # "filename"이 주어지지 않으면 기본적으로 표준입력인 0번이 할당됩니다.
    #
    # 이를 응용하면 파일의 특정한 위치에 쓰기를 할 수 있습니다.
    echo 1234567890 > File    # "File"에 문자열을 씁니다.
    exec 3<> File       # "File"을 열고 3번 파일 디스크립터를 할당합니다.
    read -n 4 <&3             # 문자 4개만 읽은 다음,
    echo -n . >&3             # 소수점을 쓰고,
    exec 3>&-                 # 3번 파일 디스크립터를 닫습니다.
    cat File                  # ==> 1234.67890
    # 어라, 랜덤 억세스네.
```

```
 |
    # 파이프.
    # 프로세스와 명령어를 엮어 주는 일반적인 목적의 툴.
    # “>”와 비슷하지만, 실제로는 좀 더 일반적으로 쓰입니다.
    # 명령어, 스크립트, 파일, 프로그램들을 함께 묶는데 유용하게 쓰입니다.
    cat *.txt | sort | uniq > result-file
    # 모든 *.txt 파일의 출력을 정렬한 다음, 중복되는 줄을 제거하고
    # 마지막으로 그 결과를 “result-file”에 저장.
```

여러개의 입출력 재지향과 파이프를 하나의 명령어 줄에서 같이 쓸 수 있습니다.

```
command < input-file > output-file
command1 | command2 | command3 > output-file
```

예 12-23 와 예 A-10 를 참고.

여러개의 출력 스트림이 한 파일로 재지향 될 수도 있습니다.

```
ls -yz >> command.log 2>&1
# "ls"의 잘못된 옵션인 "yz"의 결과를 "command.log"로 저장합니다.
# 표준에러가 파일로 재지향 됐기 때문에 어떤 에러 메세지라도 그 파일에 저장됩니다.
```

파일 디스크립터 닫기

n<&-

*n*번 입력 파일 디스크립터를 닫아 줍니다.

0<&-

<&-

표준입력을 닫아 줍니다.

n>&-

*n*번 출력 파일 디스크립터를 닫아 줍니다.

1>&-
>&-

표준출력을 닫아 줍니다.

자식 프로세스는 열려 있는 파일 디스크립터를 상속 받는데 이것 때문에 파이프가 동작합니다. 파일 디스크립터가 상속되길 바라지 않는다면 그 파일 디스크립터를 닫으면 됩니다.

```
# 파이프로 표준에러만 재지향 하기.

exec 3>&1                              # 표준출력의 현재 "값"을 저장.
ls -l 2>&1 >&3 3>&- | grep bad 3>&-    # 'ls'와 'grep'을 위해 3번 파일 디스크
립터를 닫고,
exec 3>&-                              # 이제, 스크립트 나머지 부분을 위해 닫습
니다.

# Thanks, S.C.
```

I/O 재지향에 대한 더 자세한 소개는 부록 D를 참고하세요.

16.1. exec 쓰기

exec <filename 명령어는 표준입력을 파일로 재지향 시켜줍니다. 이때부터는, 주로 키보드에서 받던 모든 표준입력이 그 파일에서 들어 오게 됩니다. 이렇게 하면 파일을 줄 단위로 읽을 수가 있게 되고 sed나 awk를 이용해서 입력되는 각 줄을 파싱할 수 있게 됩니다.

예 **16-1. exec**으로 표준입력을 재지향 하기

```
#!/bin/bash
# 'exec'로 표준입력 재지향 하기.


exec 6<&0          # 표준입력을 6번 파일 디스크립터로 링크.

exec < data-file   # 표준입력을 "data-file"에서.

read a1            # "data-file"의 첫번째 줄을 읽음.
read a2            # "data-file"의 두번째 줄을 읽음.

echo
```

```
echo "다음은 파일에서 읽어 들인 것입니다."
echo "-----------------------------------"
echo $a1
echo $a2

echo; echo; echo

exec 0<&6 6<&-
# 6번 파일 디스크립터에 저장되어 있던 표준입력을 복구시키고,
# 다른 프로세스가 쓸 수 있도록 6번 파일 디스크립터를 프리( 6<&- )시킴.
# <&6 6<&-    라고 해도 됩니다.

echo -n "데이타를 넣으세요  "
read b1  # "read"는 이제 원래 자신의 동작인 표준입력에서 입력을 받습니다.
echo "표준입력에서 읽은 값."
echo "---------------------"
echo "b1 = $b1"

echo

exit 0
```

16.2. 코드 블럭 재지향

while, until, for 루프의 코드 블럭, 심지어는 if/then 테스트문 블럭도 표준입력의 재지향을 받아 들일 수 있습니다. 함수조차도 이런 형태의 재지향을 할 수 있습니다(예 23-7 참고). 이렇게 하려면, 해당 코드 블럭의 제일 끝에 < 연산자를 두면 됩니다.

예 **16-2.** 재지향된 **while** 루프

```
#!/bin/bash

if [ -z "$1" ]
then
  Filename=names.data  # 파일이름이 지정되지 않을 경우의 기본값.
else
  Filename=$1
fi
# Filename=${1:-names.data}
# 라고 해도 됩니다(매개변수 치환).

count=0

echo

while [ "$name" != Smith ]  # $name 을 왜 쿼우트 했을까요?
```

```
do
  read name                  # 표준입력이 아니라 $Filename 에서 읽음.
  echo $name
  let "count += 1"
done <"$Filename"            # 표준입력을 $Filename 파일로 재지향.
#    ^^^^^^^^^^^^

echo; echo "$count 개의 이름을 읽었습니다."; echo

# 몇몇 오래된 쉘 스크립트 언어에서는 재지향된 루프가 서브쉘로 돕니다.
# 그렇기 때문에, $count 가 루프 밖에서 초기화되어 0 을 리턴합니다.
# Bash 와 ksh 은 가능한한 서브쉘을 안 띄우려고 하기 때문에
# 이 스크립트는 제대로 동작합니다.
# Heiner Steven 이 이 점을 지적해 주었습니다.

exit 0
```

예 **16-3.** 다른 형태의 재지향된 **while** 루프

```
#!/bin/bash

# 이 스크립트는 앞서 소개해 드렸던 스크립트의 다른 형태입니다.

# Heiner Steven 이 제공해 준 것으로,
# 재지향 루프가 서브쉘로 돌 경우에 루프 안의 변수값이
# 루프 종료후 그 값을 잃어 버리는 것에 대한 해결책입니다.


if [ -z "$1" ]
then
  Filename=names.data      # 파일이름이 지정되지 않았을 경우의 기본값.
else
  Filename=$1
fi


exec 3<&0                  # 표준입력을 3번 파일 디스크립터로 저장.
exec 0<"$Filename"         # 표준입력을 재지향.

count=0
echo


while [ "$name" != Smith ]
do
  read name                # 재지향된 표준입력($Filename)에서 읽음.
  echo $name
  let "count += 1"
```

```
done <"$Filename"          # 루프는 $Filename 파일에서 입력을 받음.
#    ^^^^^^^^^^^^


exec 0<&3                  # 원래 표준입력을 복구.
exec 3<&-                  # 임시 파일 디스크립터 3번을 닫음.

echo; echo "$count 개의 이름을 읽었습니다."; echo

exit 0
```

예 16-4. 재지향된 **until** 루프

```
#!/bin/bash
# 앞의 예제와 똑같으나 "until" 루트를 사용.

if [ -z "$1" ]
then
  Filename=names.data          # 파일이름이 지정되지 않았을 경우의 기본값.
else
  Filename=$1
fi

# while [ "$name" != Smith ]
until [ "$name" = Smith ]     # != 를 = 로 바꿔주세요.
do
  read name                    # 표준입력이 아니라 $Filename 에서 읽어 들입니다.
  echo $name
done <"$Filename"              # 표준입력을 $Filename 파일로 재지향.
#    ^^^^^^^^^^^^

# 앞 예제의 "while" 루프와 똑같은 결과가 나옵니다.

exit 0
```

예 16-5. 재지향된 **for** 루프

```
#!/bin/bash

if [ -z "$1" ]
then
  Filename=names.data           # 파일이름이 지정되지 않을 경우의 기본값.
else
  Filename=$1
fi

line_count=`wc $Filename | awk '{ print $1 }'`  # 대상 파일의 줄 수.
# "for" 루프에서 표준입력을 재지향 하는게 아주 부자연스러워 보이고
```

```
# 구현하기 까다롭겠지만, 여러분이 충분히 똑똑하다면 가능합니다.
#
# 좀 더 간단하게 하려면        line_count=$(wc < "$Filename")


for name in `seq $line_count`  # "seq"가 연속된 숫자를 출력한다는 거, 기억나시
죠?
# while [ "$name" != Smith ]   --   "while" 루프보다 더 복잡합니다.   --
do
  read name                     # 표준입력이 아닌 $Filename 에서 읽어 들입니다.
  echo $name
  if [ "$name" = Smith ]        # 이런 작업을 추가적으로 덧붙여야 합니다.
  then
    break
  fi
done <"$Filename"               # 표준입력이 $Filename 에서 재지향 됨.
#    ^^^^^^^^^^^^

exit 0
```

바로 앞의 예제를 조금 변경하면 루프의 출력도 재지향 할 수 있습니다.

예 16-6. 재지향된 **for** 루프(표준입력, 표준출력 모두 재지향됨)

```
#!/bin/bash

if [ -z "$1" ]
then
  Filename=names.data           # 파일이름이 지정되지 않을 경우의 기본값.
else
  Filename=$1
fi

Savefile=$Filename.new          # 결과를 저장할 파일이름.
FinalName=Jonah                 # "read" 시에 마지막 입력이 될 이름.

line_count=`wc $Filename | awk '{ print $1 }'`  # 대상 파일의 줄 수.


for name in `seq $line_count`
do
  read name
  echo "$name"
  if [ "$name" = "$FinalName" ]
  then
    break
  fi
done < "$Filename" > "$Savefile"     # 표준입력을 $Filename으로 재지향하고,
#    ^^^^^^^^^^^^^^^^^^^^^^^^^^^       그 결과를 백업 파일로 저장.
```

```
exit 0
```

예 **16-7.** 재지향된 **if/then** 테스트

```
#!/bin/bash

if [ -z "$1" ]
then
  Filename=names.data   # 파일이름이 지정되지 않을 경우의 기본값.
else
  Filename=$1
fi

TRUE=1

if [ "$TRUE" ]          # if true    와   if :   도 동작합니다.
then
 read name
 echo $name
fi <"$Filename"
#  ^^^^^^^^^^^^

# 파일의 첫번째 줄만 읽어 들임.
# "if/then" 테스트문은 루프안에서 쓰이지 않는 한, 반복해서 비교할 방법이 없습니다.

exit 0
```

코드 블럭의 표준출력을 재지향하는 것은 그 출력을 파일로 저장하는 효과를 가져옵니다. 예 4-2를 참고하세요.

> 참고**:** Here documents는 특별한 종류의 재지향된 코드 블럭입니다.

16.3. 응용

I/O 재지향을 솜씨 좋게 쓰면 명령어 출력의 일부분(예 11-4 참고)을 파싱하고 다시 붙일 수 있습니다. 이렇게 하면 리포트와 로그 파일을 만들어 낼 수 있습니다.

예 **16-8.** 이벤트 로깅하기

```
#!/bin/bash
# logevents.sh, by Stephane Chazelas.

# 파일로 이벤트 로깅하기.
# /var/log 에 쓸 권한이 필요하므로 루트로 실행시켜야 됩니다.
```

```
ROOT_UID=0     # $UID 0 인 사용자만이 루트 권한을 갖습니다.
E_NOTROOT=67   # 루트가 아닌 사용자일 경우의 종료 에러.


if [ "$UID" -ne "$ROOT_UID" ]
then
  echo "루트로 실행시켜야 됩니다."
  exit $E_NOTROOT
fi


FD_DEBUG1=3
FD_DEBUG2=4
FD_DEBUG3=5

# 이 스크립트가 동작하도록 하려면 다음 두 줄중에 한 줄의 주석을 풀기 바랍니다.
# LOG_EVENTS=1
# LOG_VARS=1


log()  # 시간과 날짜를 로그 파일로 쓰기.
{
echo "$(date)  $*" >&7     # 날짜를 파일로 *append" 시킵니다.
                              # 아래를 참조.
}



case $LOG_LEVEL in
 1) exec 3>&2         4> /dev/null 5> /dev/null;;
 2) exec 3>&2         4>&2         5> /dev/null;;
 3) exec 3>&2         4>&2         5>&2;;
 *) exec 3> /dev/null 4> /dev/null 5> /dev/null;;
esac

FD_LOGVARS=6
if [[ $LOG_VARS ]]
then exec 6>> /var/log/vars.log
else exec 6> /dev/null               # 출력 숨기기.
fi

FD_LOGEVENTS=7
if [[ $LOG_EVENTS ]]
then
  # then exec 7 >(exec gawk '{print strf-
time(), $0}' >> /var/log/event.log)
  # 윗 줄은 Bash 버전 2.04 에서는 동작하지 않습니다.
  exec 7>> /var/log/event.log        # "event.log" 로 append.
```

```
  log                                        # 시간과 날짜 쓰기.
else exec 7> /dev/null                    # 출력 숨기기.
fi

echo "DEBUG3: beginning" >&${FD_DEBUG3}

ls -l >&5 2>&4                        # command1 >&5 2>&4

echo "Done"                                  # command2

echo "sending mail" >&${FD_LOGEVENTS}   # "sending mail" 을 7번 파일 디스크립
터로 쓰기.


exit 0
```

주석

1. 파일 디스크립터란 운영체제가 계속 추적할 수 있도록 열려 있는 파일에 할당해 주는 간단한 숫자입니다. 파일 포인터의 간단한 버전이라고 생각하면 됩니다. C 의 파일 핸들(file handle)과 유사한 개념입니다.
2. *5번 파일 디스크립터*를 쓰면 문제가 생길 수 있습니다. Bash가 exec으로 자식 프로세스를 만들 때, 그 자식은 5번 파일 디스크립터를 상속받습니다(Chet Ramey의 이메일 아카이브, SUBJECT: RE: File descriptor 5 is held open (http://www.geocrawler.com/archives/3/342/1996/1/0/1939805/)을 참고하세요). 이 특별한 파일 디스크립터는 건드리지 않는게 좋습니다.

17장. Here Documents

*here codument*는 I/O 재지향의 특별한 형태로서, ftp나 telnet, **ex**처럼 사용자와 입력을 주고 받는(interactive) 프로그램에게 명령어 스크립트를 입력시키는데 쓰입니다. 프로그램의 입력으로 쓸 명령어 리스트들로 이루어진 스크립트는 보통 제한 문자열(limit string)로 표현됩니다. 특별한 심볼인 << 뒤에 제한 문자열을 적어 줍니다. "here document"는 `command-file`에

```
command #1
command #2
...
```

등이 포함되어 있다고 했을 때, **`interactive-program < command-file`** 이라고 해서 파일의 출력을 프로그램으로 재지향 시키는 것과 비슷한 결과를 가져 옵니다.

"here document"로는 이렇게 할 수 있습니다:

```
#!/bin/bash
interactive-program <<LimitString
command #1
command #2
...
LimitString
```

어떤 명령어 목록에서도 나타나지 않을 만큼 충분히 일반적이지 않은 제한 문자열을 골라야만 복잡한 상황을 피할 수 있습니다.

here documents는 비대화형 모드(non-interactive)로 도는 유틸리티나 명령어와 쓸 때 더 좋은 효과가 나타난다는 것에 주의하세요.

예 **17-1. dummyfile:** 두 줄짜리 더미 파일 만들기

```
#!/bin/bash

# 파일 편집을 위해 'vi'를 비대화 모드로 사용.
# ('vim'으로 하면 몇 가지 이유때문에 안 됩니다.)
# 'sed'를 에뮬레이트함.

E_BADARGS=65

if [ -z "$1" ]
then
  echo "사용법: `basename $0` filename"
  exit $E_BADARGS
fi

TARGETFILE=$1
```

```
# 파일에 두 줄을 삽입한 다음 저장.
#--------here document 시작-----------#
vi $TARGETFILE <<x23LimitStringx23
i
예제 파일의 첫번째 줄입니다.
예제 파일의 두번째 줄입니다.
^[
ZZ
x23LimitStringx23
#----------here document 끝-----------#

# 위에서 ^[ 는 Control-V 와 Escape 를 순서대로 눌렀을 때 나타나는 표시입니다.

exit 0
```

위의 스크립트는 **vi**보다 **ex**를 써서 구현했다면 더 효과적이었을 겁니다. **ex** 명령어의 목록을 갖고 있는 Here document를 *ex 스크립트*이라고 부르는데 이들은 충분히 자신만의 카테고리를 만들 수 있습니다.

예 **17-2. broadcast:** 로그인 해 있는 모든 사람들에게 메세지 보내기

```
#!/bin/bash

wall <<zzz23EndOfMessagezzz23
점심에 피자 먹습니다. 주문 하실 분은 시스템 관리자한테 이메일을 보내세요.
    (멸치(anchovy, 옮긴이: 윽, 이런 토핑도 있나요?)나 버섯 토핑을 추가하실 분은 돈
을 더 내셔야 합니다.)
# 메세지가 더 있다면 여기에 적으면 됩니다.
# 주의: 'wall'은 이 주석들도 메세지로 내보냅니다.
zzz23EndOfMessagezzz23

# wall <message-file 처럼 해서 더 효과적으로 할 수 있습니다.
# 어쨌든, 스크립트에서 메세지 형태를 결정해 두면, 손이 덜 갑니다.

exit 0
```

예 **17-3. cat**으로 여러 줄의 메세지 만들기

```
#!/bin/bash

# 'echo' 는 한 줄짜리 메세지를 찍기에 아주 좋습니다만,
# 여러줄의 메세지를 찍으려면 문제가 많습니다.
# here document 를 이용한 'cat'을 쓰면 이 문제가 해결됩니다.

cat <<End-of-message
-------------------------------------
메세지의 첫 번째줄입니다.
메세지의 두 번째줄입니다.
```

```
메세지의 세 번째줄입니다.
메세지의 네 번째줄입니다.
메세지의 마지막 줄입니다.
-------------------------------------
End-of-message

exit 0


#--------------------------------------------------------------
# 위에서 "exit 0"을 했기 때문에 다음 코드는 실행되지 않습니다.

# S.C. 가 다음처럼 하는 것도 가능하다는 것을 지적해 주었습니다.
echo "-------------------------------------
메세지의 첫 번째줄입니다.
메세지의 두 번째줄입니다.
메세지의 세 번째줄입니다.
메세지의 네 번째줄입니다.
메세지의 마지막 줄입니다.
-------------------------------------"
# 하지만 'echo'의 내용이 이스케이프되지 않으면 큰따옴표로 쿼우트되지 않아도 됩니다.
```

here document의 제한 문자열에 - 옵션을 붙이면(**`<<-LimitString`**) 출력에서 탭을 지워 줍니다(빈 칸은 아님). 이걸 쓰면 스크립트를 읽기가 좋아집니다.

예 **17-4.** 탭이 지워진 여러 줄의 메세지

```
#!/bin/bash
# 앞의 예제와 똑같지만...

#  here document 에 - 옵션을 주면(<<-)
#  문서안에 들어 있는 탭을 무시해 줍니다. 하지만 빈 칸은 *아닙니다*.

cat <<-ENDOFMESSAGE
메세지의 첫 번째줄입니다.
메세지의 두 번째줄입니다.
메세지의 세 번째줄입니다.
메세지의 네 번째줄입니다.
메세지의 마지막 줄입니다.
ENDOFMESSAGE
# 이 스크립트의 출력은 왼쪽에 붙어서 나옵니다.
# 메세지의 앞에 들어 있는 탭은 보이지 않습니다.

# 위의 5줄짜리 "message"는 앞에 빈 칸이 아니라 탭이 들어 있습니다.
# 빈 칸은 <<- 의 영향을 받지 않습니다.


exit 0
```

here document는 매개변수 치환과 명령어 치환을 지원합니다. 따라서 매개변수에 따라 다른 출력을 얻을 수 있습니다.

예 **17-5. Here document**에서 매개변수 치환하기

```
#!/bin/bash
# 매개변수 치환을 쓰는 다른 'cat' here document.

# 명령어줄 인자없이 실행시켜 보세요,              ./scriptname
# 명령어줄 인자를 한 개만 줘서 실행시켜 보세요,   ./scriptname Mortimer
# 두 낱말을 쿼우트해서 명령어줄 인자로 줘보세요,  ./scriptname "Mortimer Jones"

CMDLINEPARAM=1     # 최소한의 명령어줄 인자.

if [ $# -ge $CMDLINEPARAM ]
then
  NAME=$1          # 명령어줄 인자가 하나 이상이라면,
                   # 그냥 첫번째 인자만 받아들임.
else
  NAME="John Doe"  # 명령어줄 인자가 없을 때의 기본값.
fi

RESPONDENT="이 멋진 스크립트의 저자"


cat <<Endofmessage

안녕하세요. $NAME 씨.
전 $RESPONDENT 인데요, 만나서 반갑습니다. $NAME 씨.

# 이 주석도 출력됩니다(왜일까요?).

Endofmessage

# 빈 칸도 출력되기 때문에 "주석"도 출력됩니다. 주의하세요.

exit 0
```

here document의 앞에 나오는 “제한 문자열”을 쿼우트 해주거나 이스케이프 시켜 주면 내부에서 매개변수 치환이 못 일어나게 해 줍니다. 이건 거의 쓸모가 없겠죠.

예 **17-6.** 매개변수 치환 끄기

```
#!/bin/bash
#  매개변수 치환을 끈 'cat' here document.

NAME="John Doe"
RESPONDENT="이 멋진 스크립트의 저자"
```

```
cat <<’Endofmessage’

안녕하세요, $NAME 씨.
전 $RESPONDENT 인데요, 만나서 반갑습니다. $NAME 씨.

Endofmessage

# "제한 문자열"(limit string)을 쿼우트 해 주거나 이스케이프 시키면
# 매개변수 치환이 일어나지 않습니다.
# here document 시작부분에서
#  cat <<"Endofmessage"
#  cat <<\Endofmessage
# 이라고 해도 똑같은 결과가 나옵니다.
```

다음은 매개변수 치환을 쓰는 here document가 들어 있는 유용한 스크립트입니다.

예 **17-7. upload: “Sunsite” incoming** 디렉토리에 파일 한 쌍을 업로드

```
#!/bin/bash
# upload.sh

# 파일 두 개(Filename.lsm, Filename.tar.gz)를
# Sunsite(metalab.unc.edu)의 incoming 디렉토리로 업로드.

E_ARGERROR=65

if [ -z "$1" ]
then
  echo "사용법: `basename $0` filename"
  exit $E_ARGERROR
fi


Filename=`basename $1`          # 파일이름에서 경로명을 떼어내고,

Server="metalab.unc.edu"
Directory="/incoming/Linux"
# 이렇게 하드코딩할 필요는 없고,
# 명령어줄 인자로 처리하도록 할 수도 있습니다.

Password="your.e-mail.address"   # 알맞게 고치세요.

ftp -n $Server <<End-Of-Session
# -n 옵션은 자동 로그인을 막아줍니다.

user anonymous "$Password"
binary
```

```
bell                # 각 파일 전송이 끝날때마다 '벨'을 울려줍니다.
cd $Directory
put "$Filename.lsm"
put "$Filename.tar.gz"
bye
End-Of-Session

exit 0
```

:를 더미(dummy) 명령어로 써서 here document의 출력을 받아 들이게 할 수 있습니다. 이렇게 하면 실제로는 "아무개"(anonymous) here document를 만들어 냅니다.

예 17-8. "아무개"(anonymous) Here Document

```
#!/bin/bash

: <<TESTVARIABLES
${HOSTNAME?}${USER?}${MAIL?}  # 변수중 하나라도 세트가 안 돼 있으면 에러 메세지
를 출력.
TESTVARIABLES

exit 0
```

> 참고: Here documents는 임시 파일을 만들지만 사용후 지워지기 때문에 다른 프로세스가 접근할 수 없습니다.
>
> ```
> bash$ bash -c 'lsof -a -p $$ -d0' << EOF
> > EOF
> lsof    1213 bozo    0r   REG    3,5    0 30386 /tmp/t1213-0-sh (deleted)
> ```

> **경고**
>
> 몇몇 유틸리티들은 **here document** 안에서 제대로 동작하지 않습니다.

"here document"로 하기에 너무 복잡한 작업들은 **expect** 스크립트 언어를 고려해 보기 바랍니다. **expect**는 사용자와 입력을 주고 받는(interactive) 프로그램에게 원하는대로 입력을 넣어 줄 수 있게 해 주는 스크립트 언어입니다.

18장. 쉬어가기

이 엉뚱한 잠깐의 휴식이 독자들에게 쉴 여유를 주고 조금이라도 웃었으면 좋겠네요.

리눅스 사용자 여러분 안녕하세요. 여러분은 읽고 있는 것은 여러분에게 행운과 성공을 가져다 줄 것입니다. 이 것을 복사해서 여러분 친구 열 명에게 이메일을 보내세요. 복사하기 전에 끝에 있는 사람들 중 첫번째 사람한테 100줄 짜리 Bash 스크립트를 보내세요. 그 다음에 그 사람들 이름을 다 지우고 여러분 친구 이름을 적으세요.

이 행운의 고리를 끊지 마세요! 꼭 48시간 안에 보내야 합니다. 브룩클린의 윌프레드 P.는 이메일 열 통을 안 보냈는데 그의 직업 설명란이 "코볼 프로그래머"로 바뀐것을 알았습니다. 뉴포트 뉴스의 하워드 L.은 이메일 열 통을 다 보내고 나서 한 달이 지나자 **xbill**을 할 수 있는 비어울프(Beowulf) 클러스터를 기증 받았습니다. 시카고의 아멜리아 V.는 이 편지를 깔깔 웃으며 무시했는데, 조금 뒤에 그녀의 터미널에 불이 나서 지금은 MS 윈도우용 문서를 작성하고 있습니다.

행운의 고리를 끊지 마세요! 오늘 열 부를 복사해서 보내세요!

약간 수정된 **UNIX**식의 **"**행운의 쿠키**(fortune cookies)"** 제공

Part 4. 고급 주제들(Advanced Topics)

19장. 정규 표현식(Regular Expressions)

쉘 스크립트를 완전하게 구사하기 위해서, 정규 표현식은 꼭 정복해야 합니다. 스크립트에서 자주 쓰이는 sed나 awk같은 명령어들이 정규 표현식을 사용합니다.

19.1. 정규 표현식의 간략한 소개

표현식(expression)이란 문자 그대로의 의미 이상으로 해석되는 메타문자(metacharacters)라고 부르는 문자들의 집합을 말합니다. 예를 들어, 인용 부호(quote symbol)는 어떤 사람이 말한 것을 나타내 주기도 하지만 또한 그 뒤에 나오는 심볼에 대해서 메타적 의미를 부여하기도 합니다. 정규 표현식은 유닉스에 특별한 특징을 부여하는 문자들과 메타문자들의 집합입니다. [1]

정규 표현식은 주로 텍스트 탐색과 문자열 조작에 쓰입니다. 정규 표현식은 하나의 문자와 일치(*match*)하거나, 혹은 문자열의 일부분(substring)이나 전체 문자열인 문자 집합들과 일치하게 됩니다.

- 별표(*)는 바로 앞의 문자열이나 정규 표현식에서 **0**개 이상 반복되는 문자를 나타냅니다.

  "1133*" 은 `11 + 하나 이상의 3 + 가능한 다른 문자들` 을 나타냅니다: `113`, `1133`, `11312`, 기타 등등.

- 점(.)은 뉴라인을 제외한 오직 한 개의 글자와 일치합니다. [2]

  "13." 은 `13 + 빈칸을 포함한 최소 한 글자` 를 나타냅니다: `1133`, `11333`, 하지만 `13`은 뒤에 한 글자가 빠져 있기 때문에 아닙니다.

- 캐럿(^)은 줄의 시작을 나타내지만 가끔 문맥에 따라서는 정규 표현식에서 문자 집합의 의미를 반대로 해석해 줍니다.

- 정규 표현식의 제일 끝에 나오는 달러 표시($)는 줄 끝과 일치합니다.

  "^$" 는 빈 줄과 일치합니다.

- 대괄호([...])는 단일 정규 표현식에서 표현하기 위해 문자들을 집합으로 묶어 줍니다.

  "[xyz]" 는 `x`, `y`, `z` 중에 한 글자와 일치합니다.

  "[c-n]" 는 `c`에서 `n` 사이에 들어 있는 한 문자와 일치합니다.

  "[B-Pk-y]" 는 `B`에서 `P`까지 중이나 `k`에서 `y`까지 중의 한 글자와 일치합니다.

"[a-z0-9]" 는 소문자나 숫자중의 한 문자와 일치합니다.

"[^b-d]" 는 *b*에서 *d*사이의 문자를 제외한 모든 문자를 나타냅니다. ^은 바로 뒤에 나오는 정규 표현식의 의미를 반대로 해석하게 해 줍니다(다른 문맥에서 !의 의미와 비슷함).

여러개의 대괄호로 묶인 문자들은 일반적인 낱말 패턴을 나타냅니다. "[Yy][Ee][Ss]"는 *yes*, *Yes*, *YES*, *yEs*, 등등을 나타냅니다.
"[0-9][0-9][0-9][0-9][0-9][0-9]-[0-9][0-9][0-9][0-9][0-9][0-9][0-9]"는 주민등록번호와 일치합니다.

-

역슬래쉬(\)는 특수 문자를 원래의 문자 의미대로 해석하게 해줍니다(escape).

"\$" 는 정규 표현식에서 줄 끝(end-of-line)을 나타내는 의미대신 "$" 문자 그대로 해석하게 해줍니다. 비슷하게 "\\"는 그냥 "\" 문자 그 자체를 나타냅니다.

- 확장 정규 표현식**.** egrep, awk, 펄에서 쓰입니다.
-

물음표(?)는 자기 앞에 나오는 정규 표현식이 0개나 한개인 것과 일치하고, 보통은 한 개의 문자와 일치할 때 쓰입니다.

-

더하기(+)는 자기 앞에 나오는 하나 이상의 정규 표현식과 일치합니다. *와 비슷하게 동작하지만 반드시 하나 이상과 일치합니다.

```
# GNU 버전의 sed와 awk에서는 "+"를 쓸 수 있지만,
# 이스케이프(escape)를 해 줘야 됩니다.

echo a111b | sed -ne ’/a1\+b/p’
echo a111b | grep ’a1\+b’
echo a111b | gawk ’/a1+b/’
# 위의 세개는 모두는 동일합니다.

# Thanks, S.C.
```

-

이스케이프된 "중괄호"(\{ \})는 바로 앞에 나온 정규 표현식의 빈도수를 나타냅니다.

중괄호를 이스케이프 시키지 않으면 중괄호 문자 그대로 해석되기 때문에 꼭 이스케이프를 시켜야 합니다. 이 방법은 기술적으로 볼 때, 기본적인 정규 표현식의 일부가 아닙니다.

"[0-9]\{5\}" 는 0에서 9까지의 문자가 정확히 5번 나오는 것을 나타냅니다.

경고

"전통적"인 awk에서는 중괄호를 정규 표현식으로 쓸 수 없습니다. 하지만, **gawk**에서 `--re-interval`을 주면 중괄호를 이스케이프 시키지 않고도 정규 표현식에서 쓸 수가 있습니다.

```
bash$ echo 2222 | gawk --re-
interval '/2{3}/'
2222
```

-

  소괄호인 ( )는 정규 표현식 그룹을 묶어줍니다. 다음에 설명할 "|" 연산자와 같이 쓰면 아주 좋습니다.

-

  | "or" 정규 표현식 연산자는 가능한 문자들중 어떤 것과도 일치합니다.

```
bash$ egrep 're(a|e)d' misc.txt
People who read seem to be better informed than those who do not.
The clarinet produces sound by the vibration of its reed.
```

- **POSIX 문자 클래스(POSIX Character Classes). `[:class:]`**

  일치하는 문자의 범위를 지정하는 다른 방법입니다.

-

  `[:alnum:]` 는 알파벳이나 숫자와 일치하고 **`[A-Za-z0-9]`** 와 같은 표현입니다.

-

  `[:alpha:]` 는 알파벳과 일치하고 **`[A-Za-z]`** 와 같은 표현입니다.

-

  `[:blank:]` 는 빈 칸이나 탭과 일치합니다.

-

  `[:cntrl:]` 는 제어 문자들과 일치합니다.

-

  `[:digit:]` 는 10진 숫자들과 일치하고 **`[0-9]`** 와 같은 표현입니다.

-

[:graph:] (출력가능한 그래픽 문자들). 아스키 33 - 126 의 문자들과 일치합니다. 빈 칸 문자가 포함되지 않는다는 것만 제외하고는 밑에서 설명할 **[:print:]** 와 같습니다.

- **[:lower:]** 는 알파벳 소문자와 일치하고 **[a-z]** 와 같은 표현입니다.
- **[:print:]** (출력 가능한 문자들). 아스키 32 - 126 까지의 문자들과 일치합니다. 위에서 설명한 **[:graph:]** 와 같지만 빈 칸 문자가 포함되어 있습니다.
- **[:space:]** 는 공백문자들과 일치합니다(빈 칸, 수평탭).
- **[:upper:]** 는 알파벳 대문자와 일치하고 **[A-Z]** 와 같은 표현입니다.
- **[:xdigit:]** 는 16진수 숫자와 일치하고 **[0-9A-Fa-f]** 와 같은 표현입니다.

> 중요: POSIX 문자 클래스는 보통 쿼우팅이나 이중 대괄호([[ ]])를 해 줘야 합니다.
>
> ```
> bash$ grep [[:digit:]] test.file
> abc=723
> ```
>
> 이 문자 클래스들은 제한된 확장 형태로 globbing에서 쓰일 수도 있습니다.
>
> ```
> bash$ ls -l ?[[:digit:]][[:digit:]]?
> -rw-rw-r--    1 bozo  bozo         0 Aug 21 14:47 a33b
> ```
>
> 스크립트에서 POSIX 문자 클래스가 쓰이는 예제는 예 12-14 와 예 12-15 를 참고하세요.

Sed, awk, 펄은 스크립트에서 정규 표현식을 인자로 받아 파일이나 I/O 스크림을 걸러주거나 필터링 해줍니다. 이런 예제는 예 A-7 와 예 A-12 를 참고하세요.

Dougherty와 Robbins가 쓴 "Sed & Awk"에서 정규 표현식에 대한 완전하고 명쾌한 사용법을 볼 수 있습니다(*서지사항* 참고).

19.2. Globbing

Bash 자체는 정규 표현식을 이해하지 못 합니다. 스크립트에서는 sed 와 awk 같은 명령어나 유틸리티가 정규 표현식을 해석해 줍니다.

Bash는 "globbing"이라고 하는 파일명 확장을 수행해 주는데 이는 표준 정규 표현식을 쓰지 않고, 대신에 와일드 카드를 인식하고 확장해 줍니다. globbing은 표준 와일드 카드 문자인 *, ?, 대괄호속의 문자 목록, 다른 특수 문자들(일치하지 않도록 부정해 주는 ^ 같은 문자)을 해석해 줍니다. 하지만 globbing시, 와일드 카드 문자 사용에는 중요한 몇가지 제한 사항이 있습니다. 예를 들어, `*`는 `.bashrc`처럼 점으로 시작하는 파일과 일치하지 않습니다.[3] 비슷하게, *?*도 파일명 확장에서 쓰이면 정규 표현식의 일부분으로 쓰일 때와는 다른 의미를 갖습니다.

```
bash$ ls -l
total 2
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 a.1
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 b.1
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 c.1
-rw-rw-r--    1 bozo  bozo       466 Aug  6 17:48 t2.sh
-rw-rw-r--    1 bozo  bozo       758 Jul 30 09:02 test1.txt

bash$ ls -l t?.sh
-rw-rw-r--    1 bozo  bozo       466 Aug  6 17:48 t2.sh

bash$ ls -l [ab]*
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 a.1
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 b.1

bash$ ls -l [a-c]*
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 a.1
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 b.1
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 c.1

bash$ ls -l [^ab]*
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 c.1
-rw-rw-r--    1 bozo  bozo       466 Aug  6 17:48 t2.sh
-rw-rw-r--    1 bozo  bozo       758 Jul 30 09:02 test1.txt

bash$ ls -l {b*,c*,*est*}
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 b.1
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 c.1
-rw-rw-r--    1 bozo  bozo       758 Jul 30 09:02 test1.txt


bash$ echo *
a.1 b.1 c.1 t2.sh test1.txt

bash$ echo t*
t2.sh test1.txt
```

echo 명령어도 파일이름에 대해서 와일드 카드 확장을 해 줍니다.

예 10-4를 참고하세요.

주석

1. 어떤 메타문자도 포함하지 않아서 문자열 자체의 뜻을 그대로 갖고 있는 문자열이 가장 간단한 형태의 정규 표현식입니다.
2. sed, awk, grep은 한 줄에 대해서 처리를 하기 때문에 뉴라인을 처리하지 못 합니다. 뉴라인이 들어 있는 여러 줄에 걸친 표현식에서 점(.)은 뉴라인과 일치합니다.

```
#!/bin/bash

sed -e 'N;s/.*/[&]/' << EOF   # Here Document
line1
line2
EOF
# 출력:
# [line1
# line2]



echo

awk '{ $0=$1 "\n" $2; if (/line.1/) {print}}' << EOF
line 1
line 2
EOF
# 출력:
# line
# 1


# Thanks, S.C.

exit 0
```

3. 파일명 확장은 직접적으로 점(dot)을 표시해줘야 도트파일(dotfile)과 일치합니다.

```
~/[.]bashrc    # ~/.bashrc 로 확장되지 않습니다.
~/?bashrc      # 이것도 안 됩니다.
               # globbing에서는 와일드 카드와 메타문자가 점으로 확장되지 않습니
다.

~/.[b]ashrc    # ~./bashrc 로 확장됩니다.
~/.ba?hrc      # 역시 됩니다.
~/.bashr*      # 되겠죠?

# "dotglob" 옵션을 설정하면 이 기능을 꺼 줍니다.

# Thanks, S.C.
```

20장. 서브쉘(Subshells)

여러분이 쉘 스크립트를 실행시키게 되면 다른 명령어 프로세스의 인스턴스를 띄웁니다. 여러분이 실행시킨 명령어가 명령어 줄 프롬프트에서 해석될 때, 스크립트 파일에 들어 있는 명령어 목록도 한 번에 같이 처리하게 됩니다. 이 때 실행되는 각각의 쉘 스크립트는 사실, 콘솔이나 한텀에서 여러분에게 프롬프트를 보여주던 부모 쉘의 서브 프로세스입니다.

쉘 스크립트 자신도 서브 프로세스를 띄울 수 있습니다. 이 서브쉘 덕분에 스크립트가 병렬로 처리되고 결국 동시에 다중 작업을 하는 것과 동일한 결과를 가져옵니다.

병렬적인 명령어 목록

(command1; command2; command3; ...)

중괄호 속에 들어 있는 명령어들은 서브쉘로 돌게 됩니다.

참고**:** 서브쉘의 변수들은 서브쉘이 속해 있는 코드 블럭 밖으로 보이지 않습니다. 이 변수들은 부모 프로세스나 그 서브쉘을 띄운 쉘에서 접근할 수 없기 때문에 실제로는 지역 변수가 됩니다.

예 **20-1.** 서브쉘에서 변수의 통용 범위**(variable scope)**

```
#!/bin/bash
# subshell.sh

echo

outer_variable=Outer

(
inner_variable=Inner
echo "서브쉘의 \"inner_variable\" = $inner_variable"
echo "서브쉘의 \"outer\" = $outer_variable"
)

echo

if [ -z "$inner_variable" ]
then
  echo "inner_variable 은 쉘의 메인에서 정의되지 않았습니다."
else
  echo "inner_variable 은 쉘의 메인에서 정의되었습니다."
fi

echo "쉘 메인의 \"inner_variable\" = $inner_variable"
# $inner_variable 은 초기화되지 않은 것처럼 보이는데,
# 서브쉘에서 정의된 변수는 그 서브쉘의 "지역 변수"이기 때문입니다.
```

```
echo

exit 0
```

예 32-1 참고.

+

서브쉘에서 작업 디렉토리가 변경돼도, 부모 쉘에게 영향을 미치지 않습니다.

예 20-2. 사용자 프로파일 보기

```
#!/bin/bash
# allprofs.sh: 모든 사용자의 프로파일 출력

# 이 스크립트는 Heiner Steven 이 작성하고, 이 문서의 저자가 수정했습니다.

FILE=.bashrc  #  사용자 프로파일을 담고 있는 파일.
              #+ 원래 스크립트에서는 ".profile" 이었습니다.

for home in `awk -F: '{print $6}' /etc/passwd`
do
  [ -d "$home" ] || continue    # 홈 디렉토리가 없는 사용자라면 다음으로 넘어감.
  [ -r "$home" ] || continue    # 읽을수 없는 홈디렉토리라면 역시 그냥 넘어감.
  (cd $home; [ -e $FILE ] && less $FILE)
done

# 'cd $home' 이 서브쉘에서 돌기 때문에,
#+ 스크립트가 끝난 다음, 'cd'를 써서 원래 디렉토리로 다시 돌아갈 필요가 없습니다.

exit 0
```

서브쉘은 특정 명령어 그룹에 그들만의 "전용 환경"(dedicated environment)을 설정해 줄 수 있습니다.

```
COMMAND1
COMMAND2
COMMAND3
(
  IFS=:
  PATH=/bin
  unset TERMINFO
  set -C
  shift 5
  COMMAND4
  COMMAND5
  exit 3 # 서브쉘만 종료시킴.
)
# 부모 쉘의 환경은 영향을 받지 않고 그대로 남아 있습니다.
```

```
COMMAND6
COMMAND7
```

변수가 정의됐는지 아닌지를 확인해 보는데 이걸 응용해 볼 수 있습니다.

```
if (set -u; : $variable) 2> /dev/null
then
  echo "변수가 세트되어 있습니다."
fi

# 이렇게도 할 수 있죠.  [[ ${variable-x} != x || ${variable-y} != y ]]
# 혹은                    [[ ${variable-x} != x$variable ]]
# 역시                    [[ ${variable+x} = x ]])
```

잠금 파일을 확인하는데도 응용해 볼 수 있습니다:

```
if (set -C; : > lock_file) 2> /dev/null
then
  echo "다른 사용자가 이미 실행중입니다."
  exit 65
fi

# Thanks, S.C.
```

프로세스를 다른 서브쉘에서 병렬로 실행시킬 수도 있습니다. 이렇게 하면 복잡한 작업을 여러개로 나누어서 동시에 처리할 수 있습니다.

예 20-3. 프로세스를 서브쉘에서 병렬로 돌리기

```
(cat list1 list2 list3 | sort | uniq > list123) &
(cat list4 list5 list6 | sort | uniq > list456) &
# 동시에 두 종류의 파일들을 합치고 정렬.
# 백그라운드로 돌려서 병렬로 확실히 수행되도록 함.
#
# 다음과 같은 결과.
#   cat list1 list2 list3 | sort | uniq > list123 &
#   cat list4 list5 list6 | sort | uniq > list456 &

wait   # 서브쉘이 끝나기 전에 다음 명령어를 실행하지 않게 함.

diff list123 list456
```

서브쉘로 I/O 재지향을 하려면 **`ls -al | (command)`** 처럼 파이프 연산자인 “|”를 씁니다.

참고: *중괄호속의 명령어 블럭은 서브쉘을 띄우지 않습니다.*

{ command1; command2; command3; ... }

21장. 제한된 쉘(Restricted Shells)

제한된 쉘에서 못 쓰는 명령어들

스크립트나 스크립트의 일부분을 제한된 모드로 동작시키는 것은 제한이 없을 경우에 쓸 수 있는 몇몇 명령어들을 쓰지 못하게 합니다. 보안상의 이유로, 스크립트 사용자의 권한을 제한시키고 스크립트를 돌려서 입을 수 있는 가능한 피해를 최소화해 줍니다.

*cd*로 작업 디렉토리를 바꾸기.

$PATH*나, *$SHELL, *$BASH_ENV*, *$ENV* 환경 변수의 값을 바꾸기.

쉘 환경 변수 옵션인 *$SHELLOPTS*을 읽거나 바꾸기.

출력 재지향.

/'s을 포함한 하나 이상의 명령어 실행.

쉘에서 다른 프로세스로 옮겨가기 위해서 **exec** 부르기.

불순한 목적으로 장난을 칠 수 있거나 스크립트를 뒤집어 엎을 수 있는 다양한 명령어들.

스크립트에서 제한된 모드를 빠져나가기는 행위.

예 **21-1.** 제한된 모드로 스크립트 돌리기

```
#!/bin/bash
# 스크립트 시작부분을 "#!/bin/bash -r" 로 해 주면
# 전체 스크립트가 제한된 모드에서 동작합니다.

echo

echo "디렉토리를 바꾸겠습니다."
cd /usr/local
echo "지금은 `pwd` 에 있습니다."
echo "집(home)으로 돌아갑니다."
cd
echo "지금은 `pwd` 에 있습니다."
echo

# 여기까지는 특별한 게 없는 제한되지 않은 모드였습니다.

set -r
```

```
# set --restricted    도 같은 효과를 가져옵니다.
echo "==> 지금부터는 제한된 모드로 동작합니다. <=="

echo
echo

echo "제한된 모드에서 디렉토리를 바꾸려고 합니다."
cd ..
echo "아직도 `pwd` 에 있군요."

echo
echo

echo "\$SHELL = $SHELL"
echo "제한된 모드에서 쉘을 바꾸려고 합니다."
SHELL="/bin/ash"
echo
echo "\$SHELL= $SHELL"

echo
echo

echo "제한된 모드에서 출력을 재지향 하려고 합니다."
ls -l /usr/bin > bin.files
ls -l bin.files    # 파일이 생성됐는지 어떤지 한 번 봅시다.

echo

exit 0
```

22장. 프로세스 치환(Process Substitution)

*프로세스 치환*은 명령어 치환과 짝을 이루는 개념입니다. 명령어 치환은 **dir_contents=‘ls -al‘** 이나 **xref=$(grep word datafile)** 처럼 명령어의 결과를 어떤 변수의 값으로 설정해 줍니다. 프로세스 치환은 프로세스의 출력을 다른 프로세스에게 넣어 줍니다(다른 말로 하면, 한 명령어의 결과를 다른 명령어에게 전달합니다).

명령어 치환 템플릿

소괄호에 들어 있는 명령어

>(command)

<(command)

이런 식으로 프로세스 치환을 초기화하는데, `/dev/fd/<n>` 이라고 해서 괄호안에 있는 프로세스의 결과를 다른 프로세스에게 전달해 줍니다. [1]

> 참고: “<” 나 “>” 와 소괄호 사이에는 빈 칸이 들어 가지 않습니다. 빈 칸이 들어가면 에러 메세지가 나옵니다.

```
bash$ echo >(true)
/dev/fd/63

bash$ echo <(true)
/dev/fd/63
```

Bash는 파이프를 만들 때, `--fIn`과 `fOut--`, 두 개의 파일 디스크립터를 씁니다. true의 표준입력을 `fOut`(dup2(fOut, 0))으로 연결하고 나면, Bash가 `/dev/fd/fIn` 인자를 **echo**로 전달해 줍니다. `/dev/fd/<n>` 이 없는 시스템에서는 Bash가 임시 파일을 쓸 것입니다.(Thanks, S.C.)

```
cat <(ls -l)
# ls -l | cat 와 같습니다.

sort -k 9 <(ls -l /bin) <(ls -l /usr/bin) <(ls -l /usr/X11R6/bin)
# 3개의 중요한 'bin' 디렉토리안의 파일 목록들을 파일명 순으로 정렬해서 보여줍니다.
# 중요한 것은 3개의 구분된 명령어가 'sort'의 입력으로 들어가는 것입니다.


diff <(command1) <(command2)    # 명령어 출력에서 다른 점을 보여줍니다.

tar cf >(bzip2 -c > file.tar.bz2) dir
# "tar cf /dev/fd/?? dir" 를 부르고, "bzip2 -c > file.tar.bz2" 를 부릅니다.
#
# /dev/fd/<n> 가 시스템에 따른 특징이기 때문에 두 명령어 사이의
```

```
# 파이프가 네임드 파이프일 필요는 없습니다
#
# 위에서 했던 것을 이렇게 흉내를 내 보죠.
#
bzip2 -c < pipe > file.tar.bz2&
tar cf pipe dir
rm pipe
#        이렇게도 할 수 있겠네요.
exec 3>&1
tar cf /dev/fd/4 dir 4>&1 >&3 3>&- | bzip2 -c > file.tar.bz2 3>&-
exec 3>&-


# Thanks, S.C.
```

다음은 독자 한 분이 보내 주신 프로세스 치환에 대한 재밌는 예제입니다.

```
# SuSE 배포판의 스크립트 일부분을 인용:

while read  des what mask iface; do
# 명령어 몇 개...
done < <(route -n)


# 시험하기 위해서 몇 가지를 덧붙여 봅시다.
while read  des what mask iface; do
  echo $des $what $mask $iface
done < <(route -n)

# 출력:
# Kernel IP routing table
# Destination Gateway Genmask Flags Metric Ref Use Iface
# 127.0.0.0 0.0.0.0 255.0.0.0 U 0 0 0 lo


# S.C. 가 지적한 것처럼, 이해하기 쉽게 다시 쓰면:
route -n |
  while read des what mask iface; do   # 파이프 출력을 써서 변수를 세팅.
    echo $des $what $mask $iface
  done  # 위와 똑같은 결과를 보여줍니다.
```

주석

1. 이렇게 하면 네임드 파이프(named pipe, 임시 파일)를 쓰는 것과 같고, 사실 프로세스 치환에서 네임드 파이프가 한 번 쓰입니다.

23장. 함수

어느 정도 제한은 있지만 Bash도 "실제" 프로그래밍 언어들처럼 함수를 지원합니다. 함수란 서브루틴으로, 어떤 동작들이 구현된 코드 블럭이고, 특정한 일을 수행하는 "블랙 박스"입니다. 반복적인 코드나 약간만 다른 일을 반복하는 작업들이 있다면 함수를 쓰도록 고려해 보는 것이 좋습니다.

function *function_name* {
command...
}

혹은

function_name () {
command...
}

두 번째 형태는 C 프로그래머들에게 아주 반가울 것입니다(또한 이 형태가 더 이식성 있습니다).

C 함수 문법과 더욱 비슷하게 왼쪽 중괄호를 다음 줄에 놓아도 됩니다.

function_name ()
{
command...
}

함수는 간단하게 함수 이름을 불러서 실행시킬 수 있습니다.

예 **23-1.** 간단한 함수

```
#!/bin/bash

funky ()
{
  echo "funky 함수입니다."
  echo "funky 함수를 빠져 나갑니다."
} # 함수가 불리기 전에 선언돼 있어야 합니다.

  # 이제 함수를 부릅시다.

funky
```

```
exit 0
```

함수는 함수가 불리기 전에 정의되어야 됩니다. 예를 들어 C에서 처럼 함수를 미리 “선언”하는 방법은 없습니다.

```
# f1
# "f1"이 아직 정의되지 않았기 때문에 에러 메세지가 나옵니다.

# 그렇지만...


f1 ()
{
  echo "\"f1\" 함수에서 \"f2\" 함수 부르기."
  f2
}


f2 ()
{
  echo "\"f2\" 함수."
}

f1  # "f2" 함수는 정의되기 전에 참조되지만 실제로는 여기 전에는
    # 불리지 않기 때문에 가능합니다.

# Thanks, S.C.
```

함수 안에서 다른 함수를 정의하는게 가능하긴 하지만 그렇게 쓸모가 있진 않습니다.

```
f1 ()
{

  f2 () # 중첩됨.
  {
    echo "함수 \"f1\"속의 함수 \"f2\"."
  }

}

# f2
# 에러가 납니다.

f1  # "f1"이 자동으로 "f2"를 부르지 않기 때문에 아무 일도 일어나지 않습니다.
f2  # "f1"을 부름으로써 "f2"의 정의가 이루어졌기 때문에 이제는 "f2"를 불러도 괜찮습
니다.

# Thanks, S.C.
```

함수 선언은 명령어가 아니면 안 돼 보이는 곳에서 조차 가능합니다.

```
ls -l | foo() { echo "foo"; }  # 가능하지만 쓸모 없습니다.



if [ "$USER" = bozo ]
then
  bozo_greet ()   # if/then 문 중간에 들어간 함수 정의.
  {
    echo "Hello, Bozo."
  }
fi

bozo_greet        # bozo 사용자일 때만 동작하고 다른 사용자는 에러가 납니다.



# 어떤 상황에서는 다음처럼 하는 것이 유용할 때가 있습니다.
NO_EXIT=1   # 밑에 나올 함수 정의를 가능하게 해 줍니다.

[[ $NO_EXIT -eq 1 ]] && exit() { true; }     # "and-list" 안에서 함수를 정의.
# $NO_EXIT 가 1이라면 "exit ()"를 선언.
# "true"로 별칭(alias)을 걸어서 내장 명령어인 "exit"를 끕니다.

exit  # 내장 명령어인 "exit"를 부르지 않고 "exit ()" 함수를 부릅니다.

# Thanks, S.C.
```

23.1. 복잡 함수와 함수의 복잡성(Complex Functions and Function Complexities)

함수는 인자를 받아 들일 수 있고 다음 작업들을 위해서 종료 상태를 리턴할 수도 있습니다.

```
function_name $arg1 $arg2
```

함수는 자신에게 넘어온 인자를 위치 매개변수처럼, 인자의 위치로 참조하는데 예를 들면, 첫번째 인자는 `$1`, 두번째 인자는 `$2` 등입니다.

예 **23-2.** 매개변수를 받는 함수

```
#!/bin/bash

func2 () {
   if [ -z "$1" ]                    # 첫번째 매개변수 길이가 0 인지 확인.
```

```
   then
     echo "-첫번째 매개변수 길이가 0 입니다.-"  # 매개변수가 없을 경우에도.
   else
     echo "-첫번째 매개변수는 \"$1\" 입니다.-"
   fi

   if [ "$2" ]
   then
     echo "-두번째 매개변수는 \"$2\" 입니다.-"
   fi

   return 0
}

echo

echo "매개변수 없이 불러보죠."
func2                          # 매개변수 없이 호출
echo


echo "길이가 0 인 매개변수를 넘김."
func2 ""                       # 길이가 0 인 매개변수로 호출
echo

echo "널 매개변수를 넘김."
func2 "$uninitialized_param"   # 널 매개변수로 호출
echo

echo "매개변수 한 개를 넘김."
func2 first           # 매개변수 한 개로 호출
echo

echo "매개변수 두 개를 넘김."
func2 first second    # 매개변수 두 개로 호출
echo

echo "\"\" \"second\" 를 넘김."
func2 "" second       # 첫번째 매개변수 길이는 0
echo                  # 두번째 매개변수는 아스키 문자열

exit 0
```

참고: 다른 프로그래밍 언어들과는 다르게 쉘 스크립트는 보통, 인자를 값에 의해서 받아 들입니다.[1] 실제로는 포인터인 변수 이름이 함수의 매개변수로 넘어간다면 문자열 그대로 취급될 것이기 때문에 역참조가 불가능해 집니다. 함수는 인자를 문자 그대로 해석합니다.

종료와 리턴

종료 상태

함수는 종료 *상태*라고 부르는 값을 리턴합니다. 종료 상태는 **return** 문에 의해서 분명하게 표현되거나 그 함수안에서 마지막으로 실행된 명령어의 종료 상태를 함수 자신의 종료 상태로 가져갑니다(성공이면 0, 실패라면 0이 아닌 에러 코드). $?를 참조해서 종료 상태를 알아낼 수도 있습니다. 이 방법으로써 C 함수와 비슷하게 쉘 스크립트 함수도 효과적으로 "리턴 값"을 가질 수 있습니다.

return

함수를 끝냅니다. **return** 명령어[2] 는 임의의 정수 인자를 가질 수 있는데 이는 이 함수를 부른 쪽에게 함수의 "종료 상태"를 알려 주고, 또한 이 종료 상태는 $? 변수에 할당 됩니다.

예 **23-3.** 두 숫자중 큰 수 찾기

```
#!/bin/bash
# max.sh: 두 정수중 큰 수 찾기.

E_PARAM_ERR=-198    # 함수로 2개 미만의 매개변수가 넘어 왔을 때.
EQUAL=-199          # 두 매개변수가 같을 경우의 리턴값.

max2 ()             # 두 숫자중 큰 수를 리턴.
{                   # 주의: 비교할 숫자는 257보다 작아야 합니다.
if [ -z "$2" ]
then
  return $E_PARAM_ERR
fi

if [ "$1" -eq "$2" ]
then
  return $EQUAL
else
  if [ "$1" -gt "$2" ]
  then
    return $1
  else
    return $2
  fi
fi
}

max2 33 34
return_val=$?

if [ "$return_val" -eq $E_PARAM_ERR ]
then
  echo "매개변수가 두 개 필요합니다."
```

```
elif [ "$return_val" -eq $EQUAL ]
  then
    echo "두 숫자는 같습니다."
else
    echo "두 숫자중 큰 수는 $return_val 입니다."
fi


exit 0

# 독자용 연습문제(초급):
# 대화모드 스크립트로 변경해 보세요.
# 사용자에게 입력(두 숫자)을 물어보게 하면 됩니다.
```

작은 정보: 함수가 문자열이나 배열을 리턴하기 위해서는 전용 변수를 쓰면 됩니다.

```
count_lines_in_etc_passwd()
{
  [[ -r /etc/passwd ]] && REPLY=$(echo $(wc -l < /etc/passwd))
  # /etc/passwd 를 읽을 수 있으면 REPLY 에 줄 수를 세팅.
  # 매개변수 값과 상태 정보를 같이 리턴.
}

if count_lines_in_etc_passwd
then
  echo "/etc/passwd 에는 $REPLY 개의 줄이 있습니다."
else
  echo "/etc/passwd 의 줄 수를 셀 수가 없습니다."
fi

# Thanks, S.C.
```

예 **23-4.** 숫자를 로마 숫자로 바꾸기

```
#!/bin/bash

# 아라비아 숫자를 로마 숫자로 바꾸기
# 범위: 0 - 200
# 조잡하지만 동작은 합니다.

# 처리 가능한 숫자의 범위를 늘리거나 스크립트의 기능을 향상시키는 것은
# 독자 여러분을 위해 연습문제로 남겨 놓습니다.

# 사용법: roman number-to-convert

LIMIT=200
E_ARG_ERR=65
E_OUT_OF_RANGE=66
```

```
if [ -z "$1" ]
then
  echo "사용법: `basename $0` number-to-convert"
  exit $E_ARG_ERR
fi

num=$1
if [ "$num" -gt $LIMIT ]
then
  echo "처리 가능 범위 초과!"
  exit $E_OUT_OF_RANGE
fi

to_roman ()   # 함수를 부르기 전에 미리 선언해 줘야 됩니다.
{
number=$1
factor=$2
rchar=$3
let "remainder = number - factor"
while [ "$remainder" -ge 0 ]
do
  echo -n $rchar
  let "number -= factor"
  let "remainder = number - factor"
done

return $number
       # 독자용 연습문제:
       # 이 함수가 어떻게 동작하는지 설명해 보세요.
       # 힌트: 연속적 빼기에 의한 나누기.
}


to_roman $num 100 C
num=$?
to_roman $num 90 LXXXX
num=$?
to_roman $num 50 L
num=$?
to_roman $num 40 XL
num=$?
to_roman $num 10 X
num=$?
to_roman $num 9 IX
num=$?
to_roman $num 5 V
num=$?
to_roman $num 4 IV
```

```
num=$?
to_roman $num 1 I

echo

exit 0
```

예 10-26 도 참고.

중요**:** 함수가 리턴할 수 있는 가장 큰 양수는 256입니다. **return** 명령어는 이 제한 사항의 원인이 되는 종료 상태의 개념에 아주 밀접하게 연결되어 있습니다. 다행히도, 함수에서 아주 큰 정수를 리턴해야 하는 경우가 생기면 이런 제한 사항을 해결할 방법이 있습니다.

예 **23-5.** 함수에서 큰 값을 리턴하는지 테스트하기

```
#!/bin/bash
# return-test.sh

# 함수에서 리턴할 수 있는 가장 큰 양수는 256 입니다.

return_test ()         # 무조건 넘어온 것을 리턴.
{
  return $1
}

return_test 27         # o.k.
echo $?                # 27 리턴.

return_test 256        # 역시 o.k.
echo $?                # 256 리턴.

return_test 257        # 에러!
echo $?                # 1 리턴(자질구레한 에러용 코드 리턴).

return_test -151896    # 하지만, 큰 음수는 됩니다.
echo $?                # -151896 리턴.

exit 0
```

살펴본 것처럼, 함수는 큰 음수값을 리턴할 수 있습니다. 이를 이용하면 약간의 꽁수를 써서 큰 양수를 리턴할 수도 있습니다.

다른 방법으로는 "리턴 값"을 그냥 전역 변수에 할당하면 됩니다.

```
Return_Val=   # 초과되는 함수 리턴값을 담을 전역 변수.

alt_return_test ()
{
  fvar=$1
  Return_Val=$fvar
  return   # 0 리턴(성공).
}
```

```
alt_return_test 1
echo $?                              # 0
echo "return value = $Return_Val"    # 1

alt_return_test 256
echo "return value = $Return_Val"    # 256

alt_return_test 257
echo "return value = $Return_Val"    # 257

alt_return_test 25701
echo "return value = $Return_Val"    #25701
```

예 **23-6.** 큰 두 정수 비교하기

```
#!/bin/bash
# max2.sh: 아주 큰 두 정수중 큰 수 구하기.

# 이 스크립트는 앞에서 소개했던 "max.sh" 예제를
# 큰 정수에 대해서 동작하도록 수정한 것입니다.

EQUAL=0              # 두 매개변수가 같을 경우의 리턴 값.
MAXRETVAL=256        # 함수가 리턴할 수 있는 최대 양수.
E_PARAM_ERR=-99999   # 매개변수 에러.
E_NPARAM_ERR=99999   # "일반화된"(Normalized) 매개변수 에러.

max2 ()              # 두 숫자중 큰 수를 리턴.
{
if [ -z "$2" ]
then
  return $E_PARAM_ERR
fi

if [ "$1" -eq "$2" ]
then
  return $EQUAL
else
  if [ "$1" -gt "$2" ]
  then
    retval=$1
  else
    retval=$2
  fi
fi

# -------------------------------------------------------------- #
# 여기가 큰 정수를 리턴할 수 있게 해 주는 부분입니다.
if [ "$retval" -gt "$MAXRETVAL" ]    # 범위를 넘는 수라면,
then
  let "retval = (( 0 - $retval ))"   # 음수로 조절해 줍니다.
  # (( 0 - $VALUE )) 가 VALUE 의 부호를 바꿔줍니다.
fi
# 다행스럽게도, 큰 *음수* 리턴은 가능합니다.
```

```
# ------------------------------------------------------------- #

return $retval
}

max2 33001 33997
return_val=$?

# -----------------------------------------------------------------------
-- #
if [ "$return_val" -lt 0 ]                  # "조절된" 음수라면,
then
  let "return_val = (( 0 - $return_val ))"  # 양수로 변환.
fi                                          # $return_val의 "절대값".
# -----------------------------------------------------------------------
-- #


if [ "$return_val" -eq "$E_NPARAM_ERR" ]
then                    # 매개변수 에러 "플래그"도 부호가 바뀝니다.
  echo "에러: 매개변수가 모자랍니다."
elif [ "$return_val" -eq "$EQUAL" ]
  then
    echo "두 숫자가 같습니다."
else
    echo "두 숫자중 큰 수는 $return_val 입니다."
fi

exit 0
```

예 A-6 참고.

독자용 연습문제: 여기서 배운 것을 가지고 앞에서 살펴봤던 로마 숫자 예제가 임의의 큰 입력값을 처리하도록 고쳐 보세요.

재지향

함수의 표준입력을 재지향

함수는 본질적으로 코드 블럭인데 이것은 함수의 표준입력이 재지향 될 수 있다는 뜻입니다(예 4-1에 있는 것처럼).

예 **23-7.** 사용자 계정 이름에서 실제 이름을 알아내기

```
#!/bin/bash

# /etc/passwd 를 보고 사용자 계정이름에서 "실제 이름"을 알아냄.

ARGCOUNT=1  # 한 개 인자만 필요.
E_WRONGARGS=65
```

```
file=/etc/passwd
pattern=$1

if [ $# -ne "$ARGCOUNT" ]
then
  echo "사용법: `basename $0` USERNAME"
  exit $E_WRONGARGS
fi

file_excerpt ()  # 파일에서 패턴을 검색해서 해당 부분을 출력.
{
while read line  # while 에서 꼭 "[ condition ]" 을 안 써도 됩니다.
do
  echo "$line" | grep $1 | awk -F":" '{ print $5 }'  # awk 가 ":" 구분자
를 쓰도록.
done
} <$file  # 함수 표준입력으로 재지향.

file_excerpt $pattern

# 이 전체 스크립트는 다음처럼 해도 됩니다.
#       grep PATTERN /etc/passwd | awk -F":" '{ print $5 }'
# 아니면
#       awk -F: '/PATTERN/ {print $5}'
# 혹은
#       awk -F: '($1 == "username") { print $5 }' # 사용자 이름을 보고 실
제이름을 찾아냄
# 하지만, 이런 방법은 교육적이지는 않은 것 같네요.

exit 0
```

함수의 표준입력을 재지향 해주는 덜 헷갈린 다른 방법이 있는데, 함수 안에 들어 있는 중괄호 코드 블럭으로 표준입력을 재지향 시키는 것입니다.

```
# 이렇게 하지 말고,
Function ()
{
 ...
 } < file

# 이렇게 하세요.
Function ()
{
  {
    ...
   } < file
}

# 비슷하게,
```

```
Function ()  # 이건 되지만,
{
  {
   echo $*
  } | tr a b
}

Function ()  # 이건 안 됩니다.
{
  echo $*
} | tr a b   # 여기서는 중첩된 코드 블럭이 꼭 있어야 됩니다.


# Thanks, S.C.
```

23.2. 지역 변수와 재귀 함수(Local Variables and Recursion)

재귀 함수는 지역 변수를 써서 구현할 수 있습니다.

지역 변수

*지역적*으로 선언된 변수는 선언된 곳이 포함되어 있는 코드 블럭에서만 보이게 됩니다. 즉, 지역적 "통용 범위"(local scope)를 갖습니다. 함수에 있어 지역 변수는 오직 그 함수 블럭 안에서만 의미를 갖습니다.

예 23-8. 지역 변수의 영역(Local variable visibility)

```
#!/bin/bash

func ()
{
  local a=23
  echo
  echo "함수 안에서 a = $a"
  echo
}

func

# 이제 지역 변수인 'a'가 함수 밖에서도 보이는지 살펴보죠.

echo "함수 밖에서 a = $a"  # 아니죠. 'a'는 전역적으로 접근할 수 없습니다.
echo
```

```
exit 0
```

지역 변수를 쓰면 재귀 함수[3] 를 쓸수 있지만, 이 방법은 일반적으로 쓸데 없이 많은 작업량을 필요로 하기 때문에 쉘 스크립트에서는 쓰지 않도록 권장합니다.[4]

예 23-9. 지역 변수를 쓴 재귀 함수

```
#!/bin/bash

#               factorial
#               ---------


# bash 가 재귀 함수를 지원할까요?
# 음, 그렇긴 하지만 재귀 함수를 쓰려면 머리가 뒤죽박죽 될 겁니다.


MAX_ARG=5
E_WRONG_ARGS=65
E_RANGE_ERR=66


if [ -z "$1" ]
then
  echo "사용법: `basename $0` number"
  exit $E_WRONG_ARGS
fi

if [ "$1" -gt $MAX_ARG ]
then
  echo "범위 초과(최대 5)."
  # 현실적으로 이것보다 더 큰 범위를 원한다면
  # 실제 프로그래밍 언어로 다시 작성하기 바랍니다.
  exit $E_RANGE_ERR
fi

fact ()
{
  local number=$1
  # "number"를 지역 변수로 선언해 주지 않으면 제대로 동작하지 않습니다.
  if [ "$number" -eq 0 ]
  then
    factorial=1    # 0 의 팩토리얼 = 1.
  else
    let "decrnum = number - 1"
    fact $decrnum  # 재귀 함수 호출.
    let "factorial = $number * $?"
  fi
```

```
  return $factorial
}

fact $1
echo "$1 의 팩토리얼은 $? 입니다."

exit 0
```

스크립트에서 재귀 함수를 쓰는 예제인 예 A-11도 참고하세요. 스크립트에서 쓰이는 재귀 함수는 특히, 리소스를 많이 잡아 먹고 속도가 느리기 때문에 일반적으로 스크립트에는 적당하지 않습니다.

주석

1. 간접 변수 참조(예 35-2 참조)는 함수에 변수 포인터를 넘겨주는 서투른 방법을 제공해 줍니다.

```
#!/bin/bash

ITERATIONS=3  # 입력을 몇 번이나 받을 것인지.
icount=1

my_read () {
  # my_read varname 이라고 부르세요.
  # 이 함수는 기본값으로 처리될 이전 입력값을 대괄호로 묶어서 보여주고,
  # 새 값을 물어봅니다.

  local local_var

  echo -n "값을 넣으세요"
  eval 'echo -n "[$'$1'] "'  # 이전 값.
  read local_var
  [ -n "$local_var" ] && eval $1=\$local_var

  # "And-list": "local_var" 가 참이라면 "$1"을 "loval_var"로 세팅.
}

echo

while [ "$icount" -le "$ITERATIONS" ]
do
  my_read var
  echo "Entry #$icount = $var"
  let "icount += 1"
  echo
done
```

```
# 이 교육적인 예제는 Stephane Chazelas 가 제공해 주었습니다.

exit 0
```

2. **return** 명령어는 Bash의 내장 명령입니다.
3. Herbert Mayer는 재귀 함수를 "똑같은 알고리즘을 더 간단하게 써서 표현하는 것"이라고 정의했습니다. 재귀 함수란 자기 자신을 부르는 함수를 말합니다.
4. 재귀가 너무 많이 일어나면 스크립트가 세그폴트(segfault)를 내면서 죽을 수도 있습니다.

```
#!/bin/bash

recursive_function ()
{
(( $1 < $2 )) && f $(( $1 + 1 )) $2;
#  첫번째 매개변수가 두번째보다 작은 동안
#+ 첫번째 매개변수를 하나 증가시키고 자신을 다시 부름.
}

recursive_function 1 50000  # 50,000 번의 재귀가 일어남!
# 당연히 세그폴트가 나겠죠.

# 재귀가 이렇게 많이 일어나면 스택에 할당된 메모리를 모두 써버리기 때문에
# C 프로그램이라도 세그폴트가 날 수 있습니다.

# Thanks, S.C.

exit 0  # 이 스크립트는 정상적으로 종료하지 못 합니다.
```

24장. 별칭(Aliases)

bash의 별칭(alias)은 본래, 긴 명령어들을 치지 않기 위한 키보드 단축키나 약어일 뿐입니다. 예를 들어, `~/.bashrc` 파일에 **alias lm="ls -l | more"**라고 적어주면 명령어줄에서 **`lm`**이라고 칠 때마다 자동으로 **ls -l | more**로 바뀝니다. 이렇게 하면 명령어 행에서 엄청난 타이핑을 줄일 수 있고 아주 복잡한 명령어나 옵션의 조합들을 일일이 다 기억하고 있지 않아도 됩니다. **alias rm="rm -i"**(지울 때 물어보기 모드)라고 세팅해 놓으면 중요한 파일을 실수로 지워버리지 않게 하기 때문에 큰 사고를 막아 줍니다.

스크립트에서는 별칭(alias)이 제한된 쓰임새를 갖습니다. 별칭에 매크로 확장같은 C 전처리기(preprocessor)기 같은 기능이 있었다면 아주 좋았을텐데, 불행하게도 Bash는 별칭에 속한 인자들을 확장하지 않습니다. [1] 게다가, if/then문, 루프, 함수같은 "복합문"(compound construct)안에서는 별칭 자체의 확장이 되질 않습니다. 아마 거의 항상 그럴테지만, 별칭으로 무엇을 하려던 간에 함수에서 구현하는 것이 더 효과적일 것입니다.

예 **24-1.** 스크립트에서 쓰이는 별칭**(alias)**

```
#!/bin/bash
# 오래된 시스템에서는 #!/bin/bash2 라고 해야 됩니다.

shopt -s expand_aliases
# 이 옵션을 꼭 써야 별칭을 확장시킬 수 있습니다.


# 먼저 재미로 하나 해보죠.
alias Jesse_James='echo "\"Alias Jesse James\" was a 1959 comedy star-
ring Bob Hope."'
Jesse_James

echo; echo; echo;

alias ll="ls -l"
# 별칭을 정의하려면 작은따옴표(')나 큰따옴표(") 중 하나를 써야 됩니다.

echo "별칭인 \"ll\" 해보기:"
ll /usr/X11R6/bin/mk*   #* 잘 됩니다.

echo

directory=/usr/X11R6/bin/
prefix=mk*  # 와일드 카드가 문제를 일으키는지 한 번 봅시다.
echo "Variables \"directory\" + \"prefix\" = $directory$prefix"
echo

alias lll="ls -l $directory$prefix"

echo "별칭인 \"lll\" 해보기:"
```

```
lll          # /usr/X11R6/bin 에서 mk 로 시작하는 모든 파일들의 모든(long) 정보
를 보여줍니다.
# 별칭은 와일드 카드를 포함한 변수의 연결을 잘 처리합니다.




TRUE=1

echo

if [ TRUE ]
then
  alias rr="ls -l"
  echo "별칭 \"rr\" 을 if/then 문 안에서 해보기:"
  rr /usr/X11R6/bin/mk*   #* 에러 메세지가 나옵니다!
  # 별칭은 복합문 안에서 확장되지 않습니다.
  echo "하지만, 이미 확장된 별칭은 인식합니다:"
  ll /usr/X11R6/bin/mk*
fi

echo

count=0
while [ $count -lt 3 ]
do
  alias rrr="ls -l"
  echo "별칭 \"rrr\" 을 \"while\" 루프안에서 해보기:"
  rrr /usr/X11R6/bin/mk*   #* 역시 확장되지 않습니다.
  let count+=1
done

echo; echo

alias xyz="cat $1"   # 별칭에서 위치 매개변수 시도.
xyz                  # Bash 문서는 이런 시도를 하지 말라고 제안하고 있습니다만,
# 이 스크립트에 파일명을 줘서 돌리면 제대로 되는것처럼 보입니다.

exit 0
```

참고: **unalias** 명령어는 이전에 세팅되어 있던 별칭을 지워줍니다.

예 24-2. unalias: 별칭을 설정, 해제하기

```
#!/bin/bash

shopt -s expand_aliases  # 별칭 확장을 킴.
```

```
alias llm='ls -al | more'
llm

echo

unalias llm              # 별칭을 해제.
llm
# 'llm'이 더 이상 인식되지 않기 때문에 에러 메세지가 나옵니다.

exit 0
```

```
bash$ ./unalias.sh
total 6
drwxrwxr-x    2 bozo     bozo         3072 Feb  6 14:04 .
drwxr-xr-x   40 bozo     bozo         2048 Feb  6 14:04 ..
-rwxr-xr-x    1 bozo     bozo          199 Feb  6 14:04 unalias.sh

./unalias.sh: llm: command not found
```

주석

1. 하지만 위치 매개변수는 확장하는 것처럼 보입니다.

25장. 리스트(List Constructs)

"and list"와 "or list"는 여러 명령어들을 연속적으로 실행시킬 수 있게 해줍니다. 이걸로 아주 복잡하게 중첩돼 있는 **if/then**이나 **case** 문을 효과적으로 대치할 수 있습니다.

명령어들을 묶기

and list

```
command-1 && command-2 && command-3 && ... command-n
```

각 명령어들은 바로 앞의 명령어가 true(0) 값을 리턴하는 동안 차례대로 실행됩니다. false(0이 아닌 값)가 처음 리턴될 때, 전체 명령어 사슬이 끊어지면서 종료됩니다(false를 처음 리턴한 명령어가 마지막으로 실행되는 명령어가 됩니다).

예 **25-1. "and list"**를 써서 명령어줄 인자 확인하기

```
#!/bin/bash
# "and list"

if [ ! -z "$1" ] && echo "첫번째 인자 = $1" && [ ! -z "$2" ] && echo "두
번째 인자 = $2"
then
  echo "두 개 이상의 인자가 넘어왔습니다."
  # 모든 명령어 사슬이 참을 리턴했을 경우.
else
  echo "두 개 미만의 인자가 넘어왔습니다."
  # 명령어 사슬중 최소 하나가 거짓을 리턴.
fi
# 주의할 게 하나 있습니다.
# if [ ! -z $1 ] 는 잘 되지만 똑같을 것 같은
# if [ -n $1 ] 는 잘 안 됩니다.
# 하지만
#  if [ -n "$1" ]
# 처럼 쿼우트를 걸면 됩니다. 조심해서 쓰세요!
# 테스트에서 쓰이는 변수는 항상 쿼우트를 해서 쓰세요.


# "순수한" if/then 문을 써서 똑같은 일을 합니다.
if [ ! -z "$1" ]
then
  echo "첫번째 인자 = $1"
fi
if [ ! -z "$2" ]
then
  echo "두번째 인자 = $2"
```

```
  echo "두 개 이상의 인자가 넘어왔습니다."
else
  echo "두 개 미만의 인자가 넘어왔습니다."
fi
# 이 방법은 "and list"를 쓰는 것보다 더 길고 덜 세련돼 보입니다.


exit 0
```

예 **25-2.** “**and list**”를 써서 명령어줄 인자를 확인하는 다른 방법

```
#!/bin/bash

ARGS=1        # 원하는 인자 수.
E_BADARGS=65  # 틀린 인자 수일 경우의 종료값.

test $# -ne $ARGS && echo "사용
법: `basename $0` $ARGS argument(s)" && exit $E_BADARGS
# condition-1 이 참이면(틀린 인자 갯수) 같은 줄의 나머지 부분이 실행되고 스크립
트가 종료됨.

# 다음 줄은 위의 테스트가 실패할 경우에만 실행됩니다.
echo "인자 갯수가 맞습니다."

exit 0

# 종료값을 확인하려면 이 스크립트가 종료된 다음 "echo $?"를 해 보세요.
```

or list

```
command-1 || command-2 || command-3 || ... command-n
```

각 명령어들은 바로 앞의 명령어가 false를 리턴하는 동안 차례대로 실행됩니다. false가 처음 리턴될 때, 전체 명령어 사슬이 끊어지면서 종료됩니다(true를 처음 리턴한 명령어가 마지막으로 실행되는 명령어가 됩니다). 이는 분명히 “and list”와 정반대로 동작합니다.

예 **25-3.** “**or lists**”와 “**and list**”를 같이 쓰기

```
#!/bin/bash

# "Delete", 별로 정교하지 않은 파일 지우기 유틸리티.
# 사용법: delete filename

E_BADARGS=65

if [ -z "$1" ]
then
```

```
  echo "사용법: `basename $0` filename"
  exit $E_BADARGS
fi


file=$1  # 파일이름을 세트.

[ ! -f "$1" ] && echo "\"$1\" 파일을 찾을 수 없습니다. \
존재하지 않는 파일은 지울 수 없습니다."
# 파일이 존재하지 않을 때 에러 메세지를 내기 위해 AND LIST 사용.
# 두 줄에 걸친 echo 메세지 중간의 뉴라인을 이스케이프 시켰습니다. 잘 보세요.

[ ! -f "$1" ] || (rm -f $1; echo "File \"$file\" deleted.")
# OR LIST 를 써서 파일이 존재하는 경우에는 파일을 삭제.
# ( command1 ; command2 ) 는 사실, AND LIST 의 다른 버전입니다.

# 로직이 뒤바뀐 것에 주의하세요.
# AND LIST 는 참일 때 실행되고, OR LIST 는 거짓일 때 실행됩니다.

exit 0
```

경고

"or list"에 들어 있는 첫번째 명령어가 true를 리턴해야 동작합니다.

중요: **`and list`**나 **`or list`**의 종료 상태는 마지막으로 실행된 명령어의 종료 상태를 가져갑니다.

"and"와 "or" list를 아주 멋지게 연결해서 쓸 수도 있지만 그렇게 되면 전체 로직이 뒤죽박죽 되기 쉽고 디버깅할 때 아주 힘들어 집니다.

```
false && true || echo false    # false

# 똑같습니다.
( false && true ) || echo false     # false
# 하지만 이건 다르네요.
false && ( true || echo false )     # (아무것도 echo되지 않죠)

# 중요한 것은, 논리 연산자인 "&&"와 "||"가 같은 우선 순위를 갖기 때문에
# 왼쪽에서 오른쪽으로 그룹을 지어 전체를 평가해야 합니다.

# 뭘 어떻게 하고 있는지 모른다면 이렇게 복잡하게 쓰지 않는것이 좋습니다.

# Thanks, S.C.
```

변수를 테스트해보기 위해서 **`and / or list`**를 쓰는 예제는 예 A-6를 참고하세요.

26장. 배열

bash 새 버젼부터는 1차원 배열을 지원합니다. **`variable[xx]`**처럼 선언할 수도 있고 **`declare -a variable`**처럼 직접적으로 지정해 줄 수도 있습니다. 배열 변수를 역참조하려면(내용을 알아내려면) **`${variable[xx]}`**처럼 중괄호 표기법을 쓰면 됩니다.

예 **26-1.** 간단한 배열 사용법

```
#!/bin/bash


area[11]=23
area[13]=37
area[51]=UFOs

# 배열 멤버들은 인접해 있거나 연속적이지 않아도 됩니다.

# 몇몇 멤버를 초기화 되지 않은 채 놔둬도 됩니다.
# 배열 중간이 비어 있어도 괜찮습니다.


echo -n "area[11] = "
echo ${area[11]}    #  {중괄호}가 필요

echo -n "area[13] = "
echo ${area[13]}

echo "area[51]의 값은 ${area[51]} 입니다."

# 초기화 안 된 배열 변수는 빈 칸으로 찍힙니다.
echo -n "area[43] = "
echo ${area[43]}
echo "(area[43]은 할당되지 않았습니다)"

echo

# 두 배열 변수의 합을 세 번째 배열 변수에 할당합니다.
area[5]=`expr ${area[11]} + ${area[13]}`
echo "area[5] = area[11] + area[13]"
echo -n "area[5] = "
echo ${area[5]}

area[6]=`expr ${area[11]} + ${area[51]}`
echo "area[6] = area[11] + area[51]"
echo -n "area[6] = "
echo ${area[6]}
# 문자열에 정수를 더하는 것이 허용되지 않기 때문에 동작하지 않습니다.
```

```
echo; echo; echo

# -----------------------------------------------------------------
# 다른 배열인 "area2"를 봅시다.
# 배열 변수에 값을 할당하는 다른 방법을 보여줍니다...
# array_name=( XXX YYY ZZZ ... )

area2=( zero one two three four )

echo -n "area2[0] = "
echo ${area2[0]}
# 아하, 배열 인덱스가 0부터 시작하는군요(배열의 첫번째 요소는 [0]이지 [1]이 아닙니
다).

echo -n "area2[1] = "
echo ${area2[1]}    # [1]은 배열의 두번째 요소입니다.
# -----------------------------------------------------------------

echo; echo; echo

# -----------------------------------------------
# 또 다른 배열 "area3".
# 배열 변수에 값을 할당하는 또 다른 방법...
# array_name=([xx]=XXX [yy]=YYY ...)

area3=([17]=seventeen [24]=twenty-four)

echo -n "area3[17] = "
echo ${area3[17]}

echo -n "area3[24] = "
echo ${area3[24]}
# -----------------------------------------------

exit 0
```

배열 변수는 독특한 문법을 갖고 있고 표준 Bash 연산자들도 배열에 쓸 수 있는 특별한 옵션을 갖고 있습니다.

```
 array=( zero one two three four five )

echo ${array[0]}       #  zero
echo ${array:0}        #  zero
                       #  첫번째 요소의 매개변수 확장.
echo ${array:1}        #  ero
                       #  첫번째 요소의 두 번째 문자에서부터 매개변수 확장.

echo ${#array}         #  4
```

```
                    #  배열 첫번째 요소의 길이.
```

몇몇 Bash 내장 명령들은 배열 문맥에서 그 의미가 약간 바뀝니다. 예를 들어, unset 은 배열의 한 요소를 지워주거나 배열 전체를 지워줍니다.

예 **26-2.** 배열의 특별한 특성 몇 가지

```
#!/bin/bash

declare -a colors
# 크기 지정없이 배열을 선언하게 해줍니다.

echo "좋아하는 색깔을 넣으세요(빈 칸으로 구분해 주세요)."

read -a colors    # 아래서 설명할 특징들 때문에, 최소한 3개의 색깔을 넣으세요.
#  'read'의 특별한 옵션으로 배열에 읽은 값을 넣어 줍니다.

echo

element_count=${#colors[@]}
# 배열 요소의 총 갯수를 알아내기 위한 특별한 문법.
#     element_count=${#colors[*]} 라고 해도 됩니다.
#
#  "@" 변수는 쿼우트 안에서의 낱말 조각남(word splitting)을 허용해 줍니다.
#+ (공백문자에 의해 나눠져 있는 변수들을 추출해 냄).

index=0

while [ "$index" -lt "$element_count" ]
do    # 배열의 모든 요소를 나열해 줍니다.
  echo ${colors[$index]}
  let "index = $index + 1"
done
# 각 배열 요소는 한 줄에 하나씩 찍히는데,
# 이게 싫다면  echo -n "${colors[$index]} " 라고 하면 됩니다.
#
# 대신 "for" 루트를 쓰면:
#   for i in "${colors[@]}"
#   do
#     echo "$i"
#   done
# (Thanks, S.C.)

echo

# 좀 더 우아한 방법으로 모든 배열 요소를 다시 나열.
  echo ${colors[@]}          # echo ${colors[*]}   라고 해도 됩니다.
```

```
echo

# "unset" 명령어는 배열 요소를 지우거나 배열 전체를 지워줍니다.
unset colors[1]              # 배열의 두번째 요소를 삭제.
                             # colors[1]=    라고 해도 됩니다.
echo  ${colors[@]}           # 배열을 다시 나열하는데 이번에는 두 번째 요소가 빠져
있습니다.

unset colors                 # 배열 전체를 삭제.
                             #  unset colors[*] 나
                             #+ unset colors[@] 라고 해도 됩니다.
echo; echo -n "색깔이 없어졌어요."
echo ${colors[@]}            # 배열을 다시 나열해 보지만 비어있죠.

exit 0
```

위의 예제에서 살펴본 것처럼 **${array_name[@]}**나 **${array_name[*]}**는 배열의 모든 원소를 나타냅니다. 배열의 원소 갯수를 나타내려면 앞의 표현과 비슷하게 **${#array_name[@]}**나 **${#array_name[*]}**라고 하면 됩니다. **${#array_name}**는 배열의 첫번째 원소인 **${array_name[0]}**의 길이(문자 갯수)를 나타냅니다.

예 **26-3.** 빈 배열과 빈 원소

```
#!/bin/bash
# empty-array.sh

# 빈 배열과 빈 요소를 갖는 배열은 다릅니다.

array0=( first second third )
array1=( ” )   # "array1" 은 한 개의 요소를 갖고 있습니다.
array2=( )      # 요소가 없죠... "array2"는 비어 있습니다.

echo

echo "array0 의 요소들:  ${array0[@]}"
echo "array1 의 요소들:  ${array1[@]}"
echo "array2 의 요소들:  ${array2[@]}"
echo
echo "array0 의 첫번째 요소 길이 = ${#array0}"
echo "array1 의 첫번째 요소 길이 = ${#array1}"
echo "array2 의 첫번째 요소 길이 = ${#array2}"
echo
echo "array0 의 요소 갯수 = ${#array0[*]}"  # 3
echo "array1 의 요소 갯수 = ${#array1[*]}"  # 1  (놀랍죠!)
echo "array2 의 요소 갯수 = ${#array2[*]}"  # 0

echo
```

```
exit 0  # Thanks, S.C.
```

${array_name[@]}**와 **${array_name[*]}의 관계는 $@ 와 $*의 관계와 비슷합니다. 이 강력한 배열 표기법은 쓸모가 아주 많습니다.

```
# 배열 복사.
array2=( "${array1[@]}" )

# 배열에 원소 추가.
array=( "${array[@]}" "새 원소" )
# 혹은
array[${#array[*]}]="새 원소"

# Thanks, S.C.
```

--

배열을 쓰면 쉘 스크립트에서도 아주 오래되고 익숙한 알고리즘을 구현할 수 있습니다. 이것이 반드시 좋은 생각인지 아닌지는 독자 여러분이 결정할 일입니다.

예 **26-4.** 아주 오래된 친구**:** 버블 정렬**(Bubble Sort)**

```
#!/bin/bash

# 불완전한 버블 정렬

# 버블 정렬 알고리즘을 머리속에 떠올려 보세요. 이 스크립트에서는...

# 정렬할 배열을 매번 탐색할 때 마다 인접한 두 원소를 비교해서
# 순서가 다르면 두 개를 바꿉니다.
# 첫번째 탐색에서는 "가장 큰" 원소가 제일 끝으로 갑니다.
# 두번째 탐색에서는 두번째로 "가장 큰" 원소가 끝에서 두 번째로 갑니다.
# 이렇게 하면 각 탐색 단계는 배열보다 작은 수 만큼을 검색하게 되고,
# 뒤로 갈수록 탐색 속도가 빨라지는 것을 느낄 수 있을 겁니다.


exchange()
{
  # 배열의 두 멤버를 바꿔치기 합니다.
  local temp=${Countries[$1]} # 바꿔치기할 두 변수를 위한 임시 저장소
  Countries[$1]=${Countries[$2]}
  Countries[$2]=$temp

  return
}

declare -a Countries  # 변수 정의, 밑에서 초기화 되기 때문에 여기서는 안 써도 됩니
다.
```

```
Countries=(Netherlands Ukraine Zair Turkey Russia Yemen Syria Brazil Ar-
gentina Nicaragua Japan Mexico Venezuela Greece England Is-
rael Peru Canada Oman Denmark Wales France Kashmir Qatar Liechten-
stein Hungary)
# X로 시작하는 나라 이름은 생각이 안 나네요, 쩝...

clear  # 시작하기 전에 화면을 깨끗이 지우고...

echo "0: ${Countries[*]}"  # 0번째 탐색의 배열 전체를 보여줌.

number_of_elements=${#Countries[@]}
let "comparisons = $number_of_elements - 1"

count=1 # 탐색 횟수.

while [ $comparisons -gt 0 ]   # 바깥쪽 루프의 시작
do

  index=0  # 각 탐색 단계마다 배열의 시작 인덱스를 0으로 잡음

  while [ $index -lt $comparisons ] # 안쪽 루프의 시작
  do
    if [ ${Countries[$index]} \> ${Countries[`expr $index + 1`]} ]
    # 순서가 틀리면...
    # \> 가 아스키 비교 연산자였던거 기억나시죠?
    then
      exchange $index `expr $index + 1`  # 바꿉시다.
    fi
    let "index += 1"
  done # 안쪽 루프의 끝


let "comparisons -= 1"
# "가장 큰" 원소가 제일 끝으로 갔기 때문에 비교횟수를 하나 줄일 필요가 있습니다.

echo
echo "$count: ${Countries[@]}"
# 각 탐색 단계가 끝나면 결과를 보여줍니다.
echo
let "count += 1"   # 탐색 횟수를 늘립니다.

done  # 바깥쪽 루프의 끝

# 끝!

exit 0

 --
```

배열을 쓰면 에라토스테네스의 체(Sieve of Erastosthenes)의 쉘 스크립트 버전을 구현할 수 있습니다. 물론, 이렇게 철저히 리소스에 의존하는 어플리케이션은 C 같은 컴파일 언어로 쓰여져야 합니다. 이 쉘 스크립트 버전은 굉장히 느리게 동작합니다.

예 **26-5.** 복잡한 배열 어플리케이션**:** 에라토스테네스의 체**(Sieve of Erastosthenes)**

```
#!/bin/bash
# sieve.sh

# 에라토스테네스의 체(Sieve of Erastosthenes)
# 소수를 찾아주는 고대의 알고리즘.

# 이 스크립트는 똑같은 C 프로그램보다 두 세배는 더 느리게 동작합니다.

LOWER_LIMIT=1       # 1 부터.
UPPER_LIMIT=1000    # 1000 까지.
# (시간이 주체못할 정도로 남아 돈다면 이 값을 더 높게 잡아도 됩니다.)

PRIME=1
NON_PRIME=0

let SPLIT=UPPER_LIMIT/2
# 최적화:
# 오직 상한값의 반만 확인해 보려고 할 경우 필요.


declare -a Primes
# Primes[] 는 배열.


initialize ()
{
# 배열 초기화.

i=$LOWER_LIMIT
until [ "$i" -gt "$UPPER_LIMIT" ]
do
  Primes[i]=$PRIME
  let "i += 1"
done
# 무죄가 밝혀지기 전까지는 배열의 모든 값을 유죄(소수)라고 가정.
}

print_primes ()
{
# Primes[] 멤버중 소수라고 밝혀진 것들을 보여줍니다.

i=$LOWER_LIMIT
```

```
until [ "$i" -gt "$UPPER_LIMIT" ]
do

  if [ "${Primes[i]}" -eq "$PRIME" ]
  then
    printf "%8d" $i
    # 숫자당 8 칸을 줘서 예쁘게 보여줍니다.
  fi

  let "i += 1"

done

}

sift () # 소수가 아닌 수를 걸러냅니다.
{

let i=$LOWER_LIMIT+1
# 1 이 소수인 것은 알고 있으니, 2 부터 시작합니다.

until [ "$i" -gt "$UPPER_LIMIT" ]
do

if [ "${Primes[i]}" -eq "$PRIME" ]
# 이미 걸러진 숫자(소수가 아닌 수)는 건너뜁니다.
then

  t=$i

  while [ "$t" -le "$UPPER_LIMIT" ]
  do
    let "t += $i "
    Primes[t]=$NON_PRIME
    # 모든 배수는 소수가 아니라고 표시합니다.
  done

fi

  let "i += 1"
done


}


# 함수들을 순서대로 부릅니다.
initialize
sift
```

```
print_primes
# 이런것을 바로 구조적 프로그래밍이라고 한답니다.

echo

exit 0



# ----------------------------------------------- #
# 다음 코드는 실행되지 않습니다.

# 이것은 Stephane Chazelas 의 향상된 버전으로 실행 속도가 좀 더 빠릅니다.

# 소수의 최대 한계를 명령어줄에서 지정해 주어야 됩니다.

UPPER_LIMIT=$1                  # 명령어줄에서의 입력.
let SPLIT=UPPER_LIMIT/2         # 최대수의 중간.

Primes=( ″ $(seq $UPPER_LIMIT) )

i=1
until (( ( i += 1 ) > SPLIT ))  # 중간까지만 확인 필요.
do
  if [[ -n $Primes[i] ]]
  then
    t=$i
    until (( ( t += i ) > UPPER_LIMIT ))
    do
      Primes[t]=
    done
  fi
done
echo ${Primes[*]}

exit 0
```

이 배열 기반의 소수 생성기와 배열을 쓰지 않는 예 A-11를 비교해 보세요.

--

배열의 “첨자”(subscript)를 능숙하게 조작하려면 임시로 쓸 변수가 있어야 합니다. 다시 말하지만, 이런 일이 필요한 프로젝트들은 펄이나 C 처럼 더 강력한 프로그래밍 언어의 사용을 고려해 보기 바랍니다.

예 **26-6.** 복잡한 배열 어플리케이션**:** 기묘한 수학 급수 탐색**(Exploring a weird mathematical series)**

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

```
# Douglas Hofstadter 의 유명한 "Q-급수"(Q-series):

# Q(1) = Q(2) = 1
# Q(n) = Q(n - Q(n-1)) + Q(n - Q(n-2)), for n>2

# "무질서한" Q-급수는 이상하고 예측할 수 없는 행동을 보입니다.
# 이 급수의 처음 20개 항은 다음과 같습니다:
# 1 1 2 3 3 4 5 5 6 6 6 8 8 8 10 9 10 11 11 12

# Hofstadter 의 책, "Goedel, Escher, Bach: An Eternal Golden Braid",
# p. 137, ff. 를 참고하세요.


LIMIT=100     # 계산할 항 수
LINEWIDTH=20  # 한 줄에 출력할 항 수

Q[1]=1        # 처음 두 항은 1.
Q[2]=1

echo
echo "Q-급수 [$LIMIT 항]:"
echo -n "${Q[1]} "             # 처음 두 항을 출력
echo -n "${Q[2]} "

for ((n=3; n <= $LIMIT; n++))  # C 형태의 루프 조건.
do   # Q[n] = Q[n - Q[n-1]] + Q[n - Q[n-2]]  for n>2
# Bash 는 복잡한 배열 연산을 잘 처리할 수 없기 때문에
# 위의 식을 한번에 계산하지 않고 중간에 다른 항을 두어 계산할 필요가 있습니다.

  let "n1 = $n - 1"        # n-1
  let "n2 = $n - 2"        # n-2

  t0=`expr $n - ${Q[n1]}`  # n - Q[n-1]
  t1=`expr $n - ${Q[n2]}`  # n - Q[n-2]

  T0=${Q[t0]}              # Q[n - Q[n-1]]
  T1=${Q[t1]}              # Q[n - Q[n-2]]

Q[n]=`expr $T0 + $T1`      # Q[n - Q[n-1]] + Q[n - ![n-2]]
echo -n "${Q[n]} "

if [ `expr $n % $LINEWIDTH` -eq 0 ]    # 예쁜 출력
then   #    나머지
  echo # 각 줄이 구분되도록 해 줌.
fi

done
```

```
echo

exit 0

# 여기서는 Q-급수를 반복적으로 구현했습니다.
# 좀 더 직관적인 재귀적 구현은 독자들을 위해 남겨 놓겠습니다.
# 경고: 이 급수를 재귀적으로 계산하면 "아주" 긴 시간이 걸립니다.
```

--

bash는 1차원 배열만 지원합니다만, 약간의 속임수를 쓰면 다차원 배열을 흉내낼 수 있습니다.

예 **26-7. 2**차원 배열을 흉내낸 다음**,** 기울이기**(tilting it)**

```
#!/bin/bash
# 2차원 배열을 시뮬레이트.

# 2차원 배열은 열(row)을 연속적으로 저장해서 구현합니다.

Rows=5
Columns=5

declare -a alpha     # C 에서
                     # char alpha[Rows][Columns];
 # 인 것처럼. 하지만 불필요한 선언입니다.

load_alpha ()
{
local rc=0
local index


for i in A B C D E F G H I J K L M N O P Q R S T U V W X Y
do
  local row=`expr $rc / $Columns`
  local column=`expr $rc % $Rows`
  let "index = $row * $Rows + $column"
  alpha[$index]=$i   # alpha[$row][$column]
  let "rc += 1"
done

# declare -a alpha=( A B C D E F G H I J K L M N O P Q R S T U V W X Y )
# 라고 하는 것과 비슷하지만 이렇게 하면 웬지 2차원 배열같은 느낌이 들지 않습니다.
}

print_alpha ()
{
local row=0
```

```
local index

echo

while [ "$row" -lt "$Rows" ]   # "열 우선"(row major) 순서로 출력
                               # 열(바깥 루프)은 그대로고 행이 변함.
do
  local column=0

  while [ "$column" -lt "$Columns" ]
  do
    let "index = $row * $Rows + $column"
    echo -n "${alpha[index]} "  # alpha[$row][$column]
    let "column += 1"
  done

  let "row += 1"
  echo

done

# 간단하게 다음처럼 할 수도 있습니다.
#   echo ${alpha[*]} | xargs -n $Columns

echo
}

filter ()     # 배열의 음수 인덱스를 걸러냄.
{

echo -n "  "  # 기울임(tilt) 제공.

if [[ "$1" -ge 0 &&  "$1" -lt "$Rows" && "$2" -ge 0 && "$2" -
lt "$Columns" ]]
then
    let "index = $1 * $Rows + $2"
    # 이제, 회전(rotate)시켜 출력.
    echo -n " ${alpha[index]}"  # alpha[$row][$column]
fi

}




rotate ()  # 배열 왼쪽 아래를 기준으로 45도 회전.
{
local row
local column
```

```
for (( row = Rows; row > -Rows; row-- ))  # 배열을 뒤에서부터 하나씩 처리.
do

  for (( column = 0; column < Columns; column++ ))
  do

    if [ "$row" -ge 0 ]
    then
      let "t1 = $column - $row"
      let "t2 = $column"
    else
      let "t1 = $column"
      let "t2 = $column + $row"
    fi

    filter $t1 $t2   # 배열의 음수 인덱스를 걸러냄.
  done

  echo; echo

done

# 배열 회전(array rotation)은 Herbert Mayer 가 쓴
# "Advanced C Programming on the IBM PC"에 나온 예제(143-146 쪽)에서
# 영감을 받아 작성했습니다(서지사항 참고).

}


#------------------------------------------------------#
load_alpha     # 배열을 읽고,
print_alpha    # 출력한 다음,
rotate         # 반시계 방향으로 45도 회전.
#------------------------------------------------------#


# 이 스크립트는 예제를 위한 예제이기 때문에 약간 어색한 면이 있습니다.
#
# 독자를 위한 연습문제 1:
# 배열을 읽어 들이고 출력하는 함수를
# 좀 더 교육적이고 우아하게 다시 작성해 보세요.
#
# 연습문제 2:
# 배열 회전 함수가 어떻게 동작하는지 알아내 보세요.
# 힌트: 배열의 역인덱싱이 의미하는 바가 뭘까요?

exit 0
```

27장. 파일들

시스템 구동(startup) 파일

이 파일들은 Bash 가 사용자 쉘로 동작하거나 시스템 초기화후 동작하는 모든 Bash 스크립트에서 별칭(alias)과 환경 변수를 쓸 수 있도록 해 줍니다.

`/etc/profile`

주로 환경을 세팅해 주는 파일로서, 시스템 전체에 기본으로 적용(Bash [1] 뿐만 아니라 Bourne 타입의 모든 쉘에 적용됩니다).

`/etc/bashrc`

시스템 전체에 영향을 미치는 함수들과 Bash 용 별칭(alias) 포함

`$HOME/.bash_profile`

사용자용 Bash 환경 기본값 세팅으로 사용자의 홈 디렉토리에 있습니다(`/etc/profile`의 로컬 형태).

`$HOME/.bashrc`

사용자용 bash 초기화 파일로서, 각 사용자의 홈 디렉토리에 있습니다(`/etc/bashrc`의 로컬 형태). 오직 대화형 모드(interactive)의 쉘이나 사용자 스크립트만 이 파일을 읽습니다. `.bashrc` 파일의 샘플을 부록 F에서 살펴보기 바랍니다.

로그아웃 파일

`$HOME/.bash_logout`

사용자용 명령 파일(instruction file)로서, 각 사용자의 홈 디렉토리에 있습니다. 로그인 해 있던 Bash 쉘을 종료하게 되면 이 파일에 들어 있는 명령어가 실행됩니다.

주석

1. 이 파일은 **csh**이나 **tcsh**등 고전적인 본쉘(**sh**)에서 유래하지 않거나 본쉘과 상관없는 쉘에는 적용되지 않습니다.

28장. /dev 와 /proc

전형적으로 리눅스나 유닉스는 특별한 목적을 갖는 `/dev` 와 `/proc` 을 갖고 있습니다.

28.1. `/dev`

`/dev` 디렉토리는 존재하거나 존재하지 않는 물리적 디바이스들의 항목을 갖고 있습니다. [1] df 를 옵션 없이 치면 `/dev` 디렉토리의 현재 마운트된 파일시스템을 포함하는 하드 드라이브의 파티션을 보여줍니다.

```
bash$ df
Filesystem           1k-blocks      Used Available Use%
Mounted on
/dev/hda6               495876    222748    247527  48% /
/dev/hda1                50755      3887     44248   9% /boot
/dev/hda8               367013     13262    334803   4% /home
/dev/hda5              1714416   1123624    503704  70% /usr
```

`/dev` 디렉토리에는 `/dev/loop0`같은 루프백 디바이스가 들어 있는데, 루프백 디바이스란 보통 파일을 블럭 디바이스처럼 접근할 수 있게 해 주는 일종의 속임수(gimmick)입니다. [2] 이 루프백 디바이스는 파일 시스템 전체를 하나의 큰 파일로 마운트 할 수 있게 해 줍니다. 예 13-6와 예 13-5를 참고하세요.

`/dev/null`, `/dev/zero`, `/dev/urandom` 같은 몇몇 가상 디바이스들은 다른 특화된 용도가 있습니다.

28.2. `/proc`

`/proc` 디렉토리는 실제로는 가상 파일시스템입니다. `/proc` 디렉토리에 들어 있는 파일들은 현재 실행중인 시스템과 커널 프로세스에 대한 정보및 통계 자료를 반영하고 있습니다.

```
bash$ cat /proc/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 ttyS
  5 cua
  7 vcs
 10 misc
 14 sound
 29 fb
```

```
 36 netlink
128 ptm
136 pts
162 raw
254 pcmcia

Block devices:
  1 ramdisk
  2 fd
  3 ide0
  9 md
```

```
bash$ cat /proc/interrupts
           CPU0
  0:      84505          XT-PIC  timer
  1:       3375          XT-PIC  keyboard
  2:          0          XT-PIC  cascade
  5:          1          XT-PIC  soundblaster
  8:          1          XT-PIC  rtc
 12:       4231          XT-PIC  PS/2 Mouse
 14:     109373          XT-PIC  ide0
NMI:          0
ERR:          0
```

```
bash$ cat /proc/partitions
major minor  #blocks  name     rio rmerge rsect ruse wio wmerge wsect wuse run-
ning use aveq

   3     0    3007872 hda 4472 22260 114520 94240 3551 18703 50384 549710 0 111550 644030
   3     1      52416 hda1 27 395 844 960 4 2 14 180 0 800 1140
   3     2          1 hda2 0 0 0 0 0 0 0 0 0 0 0
   3     4     165280 hda4 10 0 20 210 0 0 0 0 0 210 210
   ...
```

```
bash$ cat /proc/loadavg
0.13 0.42 0.27 2/44 1119
```

셸 스크립트는 `/proc`에 있는 파일에서 데이터를 뽑아낼 수도 있습니다.[3]

```
kernel_version=$( awk '{ print $3 }' /proc/version )
```

```
CPU=$( awk '/model name/ {print $4}' < /proc/cpuinfo )

if [ $CPU = Pentium ]
then
  어떤 명령어 실행
  ...
else
  다른 명령어 실행
  ...
fi
```

`/proc` 디렉토리에는 별나게도 숫자로 된 디렉토리들이 들어 있습니다. 이 서브 디렉토리들은 현재 실행중인 프로세스 ID를 나타냅니다. 각 서브 디렉토리에는 해당 프로세스에 대한 유용한 정보들을 담고 있는 파일이 들어 있는데, `stat`과 `status` 파일은 프로세스에 대한 통계 정보를 담고 있고, `cmdline` 파일은 해당 프로세스가 실행됐을 때의 명령어 줄 인자를 담고 있고, `exe` 파일은 실행중인 프로세스의 완전한 경로명을 심볼릭 링크하고 있습니다. 각 서브 디렉토리에는 이런식의 다른 파일들도 더 있지만 쉘 스크립팅의 관점에서는 이 정도면 충분할 겁니다.

예 **28-1.** 특정 **PID**와 관련있는 프로세스 찾기

```
#!/bin/bash
# pid-identifier.sh: PID에 해당하는 프로세스의 완전한 경로명을 찾기.

ARGNO=1  # 필요한 인자 수.
E_WRONGARGS=65
E_BADPID=66
E_NOSUCHPROCESS=67
E_NOPERMISSION=68
PROCFILE=exe

if [ $# -ne $ARGNO ]
then
  echo "사용법: `basename $0` PID-number" >&2  # 에러 메세지 >표준에러.
  exit $E_WRONGARGS
fi

pidno=$( ps ax | grep $1 | awk '{ print $1 }' | grep $1 )
# "ps" 목록에서 pid(첫번째 필드)를 확인.
# 그 다음에는 이 스크립트가 실행시킨 프로세스가 아닌 실제 프로세스인지 확인.
# 마지막의 "grep $1"이 이런 가능성을 제거해 줍니다.
if [ -z "$pidno" ]  # 결과가 아무것도 없다면 길이가 0인 문자열이 됩니다.
then                # 주어진 pid로 실행중인 프로세스가 없음.
  echo "실행중인 프로세스가 없습니다."
  exit $E_NOSUCHPROCESS
fi

# 대신 이렇게 해도 됩니다:
```

```
#   if ! ps $1 > /dev/null 2>&1
#   then                # 주어진 pid로 실행중인 프로세스 없음.
#     echo "실행중인 프로세스가 없습니다."
#     exit $E_NOSUCHPROCESS
#    fi

# 이 전체 과정은 "pidof" 명령어로 간단하게 할 수 있습니다.


if [ ! -r "/proc/$1/$PROCFILE" ]  # 읽기 퍼미션 확인.
then
  echo "$1 프로세스는 실행중이지만,"
  echo "/proc/$1/$PROCFILE 을 읽을 수 없습니다."
  exit $E_NOPERMISSION  # 일반 사용자는 /proc 의 몇몇 파일에 접근할 수 없습니다.
fi

# 바로 앞의 두 테스트들을 이렇게 할 수도 있습니다:
#    if ! kill -0 $1 > /dev/null 2>&1 # '0' 은 시그널이 아니고 해당 프로세스에
  # 시그널을 보낼 수 있는지를 확인해서
  # 실행 여부를 알 수 있습니다.
#    then echo "PID 가 존재하지 않거나 소유자가 아닙니다." >&2
#    exit $E_BADPID
#    fi



exe_file=$( ls -l /proc/$1 | grep "exe" | awk '{ print $11 }' )
# 혹은       exe_file=$( ls -l /proc/$1/exe | awk '{print $11}' )
#
# /proc/pid-number/exe 는 그 실행중인 프로세스의
# 완전한 경로명에 대한 심볼릭 링크입니다.

if [ -e "$exe_file" ]  # /proc/pid-number/exe 가 존재한다면...
then                 # 해당 프로세스가 존재하는 것임.
  echo "$1 번 프로세스는 $exe_file 으로 실행됐습니다."
else
  echo "실행중인 프로세스가 없습니다."
fi


# 이렇게 공을 들여 작성한 이 스크립트는
# ps ax | grep $1 | awk '{ print $5 }'
# 로 "거의" 대치 가능합니다.
# 하지만 'ps'의 5번째 필드가 실행 파일의 경로이름이 아니라
# 그 프로세스의 argv[0] 이기 때문에 제대로 동작하지 않습니다.
#
# 하지만 다음은 제대로 될 겁니다.
#       find /proc/$1/exe -printf '%l\n'
#       lsof -aFn -p $1 -d txt | sed -ne 's/^n//p'
```

```
# Stephane Chazelas 가 추가 설명을 해 주었습니다.

exit 0
```

예 28-2. 온라인 연결 상태

```
#!/bin/bash

PROCNAME=pppd        # ppp 데몬
PROCFILENAME=status  # 찾을 곳
NOTCONNECTED=65
INTERVAL=2           # 매 2 초마다 업데이트

pidno=$( ps ax | grep -v "ps ax" | grep -
v grep | grep $PROCNAME | awk '{ print $1 }' )
# 'ppp 데몬'인 'pppd'의 프로세스 번호를 찾는데,
# 그 프로세스를 찾기 위한 프로세스 자신은 제거해 줘야 됩니다.
#
#  하지만, Oleg Philon 이 지적했듯이,
#+ 그냥 간단히 "pidof"를 써서 아주 간단하게 할 수 있습니다.
#  pidno=$( pidof $PROCNAME )
#
#  여기서 배울 점:
#+ 명령어들이 너무 복잡해 진다면, 간단한 방법을 찾아 볼 것.


if [ -z "$pidno" ]   # pid 가 없다면 그 프로세스는 실행중이 아님.
then
  echo "연결중이 아닙니다."
  exit $NOTCONNECTED
else
  echo "연결중입니다."; echo
fi

while [ true ]       # 무한 루프, 이 부분은 좀 더 개선될 수 있습니다.
do

  if [ ! -e "/proc/$pidno/$PROCFILENAME" ]
  # 프로세스가 실행중이면, "status" 파일도 존재합니다.
  then
    echo "연결이 끊어졌습니다."
    exit $NOTCONNECTED
  fi

netstat -s | grep "packets received"  # 몇 가지 접속 통계.
netstat -s | grep "packets delivered"
```

```
  sleep $INTERVAL
  echo; echo

done

exit 0

# 이 스크립트는 Control-C 로만 끝낼 수 있습니다.

#    독자들을 위한 연습문제:
#    "q"를 눌렀을 때 종료하도록 개선시켜 보세요.
#    사용자가 좀 더 쓰기 좋게(user-friendly) 만들어 보세요.
```

주의

보통, `/proc` 밑에 있는 파일에 쓰기를 하면, 파일 시스템을 손상 시킬 수도 있고 시스템을 망가트릴 수도 있기 때문에 위험합니다.

주석

1. `/dev` 디렉토리에 들어 있는 항목들은 물리적 장치나 가상의 장치용 마운트 포인트를 제공해 줍니다. 이 목록들은 드라이브 공간을 아주 조금밖에 차지하지 않습니다.

   `/dev/null`이나 `/dev/zero`, `/dev/urandom`같은 파일들은 가상 장치입니다. 물리적으로 진짜 존재하는 장치가 아니라 소프트웨어적으로만 존재하는 장치입니다.

2. 블럭 디바이스는 데이터를 블럭 단위로 읽거나 쓰는데, 한 글자 단위로 데이터에 접근하는 문자 디바이스와 대조를 이룹니다. 하드 드라이브나 시디롬 드라이브같은 것이 블럭 디바이스고, 키보드 같은 것이 문자 디바이스입니다.

3. procinfo, free, vmstat, lsdev, uptime 같은 명령어들이 이렇게 동작합니다.

29장. 제로와 널(Of Zeros and Nulls)

`/dev/zero` 와 `/dev/null`

`/dev/null`의 쓰임새

`/dev/null`을 "블랙홀"이라고 상상해 보세요. 이 파일은 거의 읽기 전용 파일이나 마찬가집니다. 이 파일에 쓰는 모든것은 영원히 사라져 버릴겁니다. 이 파일에서 무언가를 읽으려고 하거나 어떤 결과를 바라는 것은 무의미한 일입니다. 그럼에도 불구하고, `/dev/null`은 명령어 줄이나 스크립트에서 아주 유용하게 쓸 수 있습니다.

표준출력이나 표준에러 막기(예 30-1에서 인용):

```
rm $badname 2>/dev/null
#           에러 메세지[stderr]는 완전히 사라져 버립니다.
```

파일 자체와 모든 퍼미션은 그대로 가지면서 내용만 지우기(예 2-1 와 예 2-2에서 인용):

```
cat /dev/null > /var/log/messages
#  : > /var/log/messages   라고 해도 같지만, 이렇게 하면 새 프로세스를 띄우
지 않습니다.

cat /dev/null > /var/log/wtmp
```

로그 파일의 내용을 자동으로 비우기(상용 웹 사이트에서 보내는 귀찮은 "쿠키"를 처리할 때 특별히 좋습니다):

예 29-1. 쿠키 항아리를 숨기기

```
if [ -f ~/.netscape/cookies ]  # 있다면 지우고,
then
  rm -f ~/.netscape/cookies
fi

ln -s /dev/null ~/.netscape/cookies
# 이제 모든 쿠키는 디스크에 저장되지 않고 블랙홀로 보내집니다.
```

`/dev/zero`의 쓰임새

`/dev/zero`도 `/dev/null`처럼 가상 파일(pseudo file)이지만, 실제로 널 값을 갖고 있습니다(아스키 문자 0이 아닌 숫자 0). 이 파일에 무언가를 쓰면 그 출력은 사라집니다. 이 파일에서 널 값을 읽어 내는 것은 아주 어렵습니다만 od 명령어나 헥사 에디터로 할 수는 있습니다. `/dev/zero`는 특정한 길이의 초기화된 더미 파일을 임시 스왑 파일로 만드는데 주로 쓰입니다.

예 29-2. `/dev/zero`로 스왑 파일 세팅하기

```
#!/bin/bash

# 스왑 파일 만들기.
# 루트로 실행시키세요.

ROOT_UID=0         # 루트 $UID 는 0.
E_WRONG_USER=65    # 루트가 아님.

FILE=/swap
BLOCKSIZE=1024
MINBLOCKS=40
SUCCESS=0

if [ "$UID" -ne "$ROOT_UID" ]
then
  echo; echo "이 스크립트는 루트만 실행시킬 수 있습니다."; echo
  exit $E_WRONG_USER
fi


if [ -n "$1" ]
then
  blocks=$1
else
  blocks=$MINBLOCKS              # 명령어줄에서 지정해 주지 않으면
fi                               # 40 블럭을 기본값으로 세트.

if [ "$blocks" -lt $MINBLOCKS ]
then
  blocks=$MINBLOCKS              # 최소 40 블럭이어야 됩니다.
fi


echo "Creating swap file of size $blocks blocks (KB)."
dd if=/dev/zero of=$FILE bs=$BLOCKSIZE count=$blocks  # Zero out file.

mkswap $FILE $blocks             # Designate it a swap file.
swapon $FILE                     # Activate swap file.

echo "Swap file created and activated."

exit $SUCCESS
```

`/dev/zero`의 다른 응용으로는, 파일이 0으로만 이루어진 지정된 크기를 갖게 하는 것인데, 루프백 디바이스를 마운트 하는 등의 특별한 목적을 위해 쓰입니다. 예 13-6와 예 12-33를 참고하세요.

예 29-3. 램디스크 만들기

```
#!/bin/bash
# ramdisk.sh

#  "ramdisk" 란 시스템의 RAM 의 일정 부분(segment)을
#+ 파일시스템처럼 쓰는 것을 말합니다.
#  램디스크의 장점은 읽고/쓰기가 아주 빠르다는데 있습니다.
#  단점: 휘발성이 있기 때문에 시스템이 리부트되거나 꺼지면 그 내용을 잃어버립니다.
#        램디스크로 할당한 만큼의 메모리를 못 쓰게 됩니다.
#
#  램디스크가 뭐가 좋을까요?
#  테이블이나 사전처럼 아주 큰 데이타를 램디스크에 올려 놓으면
#+ 디스크 접근 속도보다 메모리 접근 속도가 훨씬 빠르기 때문에 데이타 탐색 속도
가 빨라집니다.


E_NON_ROOT_USER=70              # 루트로 실행.
ROOTUSER_NAME=root

MOUNTPT=/mnt/ramdisk
SIZE=2000                       # 2K 블럭(필요에 따라 수정)
BLOCKSIZE=1024                  # 1K (1024 byte) 블럭 크기
DEVICE=/dev/ram0                # 첫번째 램 디바이스

username=`id -nu`
if [ "$username" != "$ROOTUSER_NAME" ]
then
  echo "\"`basename $0`\" 는 루트로 실행시켜야 됩니다."
  exit $E_NON_ROOT_USER
fi

if [ ! -d "$MOUNTPT" ]          #  마운트 포인트가 존재하는지 확인해서
then                            #+ 이 스크립트를 여러번 실행시켜도 에러가 나
지 않도록 함.
  mkdir $MOUNTPT
fi

dd if=/dev/zero of=$DEVICE count=$SIZE bs=$BLOCKSIZE  # 램 디바이스 초기
화(zero out).
mke2fs $DEVICE                  # 램 디스크에 ext2 파일시스템을 만들고,
mount $DEVICE $MOUNTPT          # 마운트.
chmod 777 $MOUNTPT              # 일반 사용자도 접근 가능하게.
                                # 하지만 언마운트는 루트만.

echo "이제 \"$MOUNTPT\" 를 쓸 수 있습니다."
# 이제부터는 일반사용자까지도 램디스크에 파일을 저장할 수 있습니다.

#  주의할 점은 램디스크가 휘발성을 갖기 때문에 리부트나 전원이 꺼질 때에는
```

```
#+ 그 내용이 없어집니다.
#  저장하고 싶은 것이 있다면 램디스크가 아닌 일반 디렉토리로 복사해서 쓰면 됩니다.

# 리부트후에 램디스크를 다시 셋업하고 싶다면 이 스크립트를 실행시키면 됩니다.
#  /mnt/ramdisk 를 이 스크립트를 통하지 않고
#+ 다른 방법으로 다시 마운트하려고 한다면 제대로 동작하지 않을 것입니다.

exit 0
```

30장. 디버깅

Bash 쉘에는 디버거도 없고 디버깅용 명령어도 없습니다. 스크립트의 문법 에러나 명백한 오자(typos)등이 만들어 내는 암호같은 에러 메세지들은 제대로 동작하지 않는 스크립트를 디버깅하는데 아무 도움도 되지 않습니다.

예 **30-1.** 버그 있는 스크립트

```
#!/bin/bash
# ex74.sh

# 버그 있는 스크립트.

a=37

if [$a -gt 27 ]
then
  echo $a
fi

exit 0
```

스크립트의 출력:

```
./test23: [37: command not found
```

위의 스크립트는 뭐가 잘못된 걸까요?(힌트: **if** 다음을 잘 살펴보세요)

스크립트가 실행은 되지만 생각했던대로 동작하지 않는다면 어떻게 할까요? 이런 것을 보통, 로직 에러라고 합니다.

예 **30-2. test24,** 버그가 있는 다른 스크립트

```
#!/bin/bash

#  현재 디렉토리에서 파일 이름에 빈 칸이 들어간 모든 파일들을
#+ 지우려고 하는데 안 됩니다. 왜 그럴까요?


badname=`ls | grep ' '`

# echo "$badname"

rm "$badname"

exit 0
```

예 30-2에서 뭐가 잘못 됐는지 알아보려면 **`echo "$badname"`**이 있는 줄의 주석을 풀어 보세요. echo 문은 여러분이 바라던 값을 제대로 얻었는지 아는데 유용하게 쓰입니다.

이렇게 특별한 경우에, **`rm "$badname"`**이라고 하면 원하는 결과를 얻지 못하는데 왜냐하면 여기서는 `$badname`이 쿼우트 되면 안 되기 때문입니다. `$badname`을 쿼우트 해주면 **rm**이 단지 한 개의 인자(하나의 파일명과 일치)를 갖도록 해 줍니다. 부분적인 해결책은 `$badname`의 쿼우트를 없애고, `$IFS`가 뉴라인 문자만을 갖도록 **`IFS=$'\n'`**라고 해주면 됩니다. 하지만, 더 간단하게 하려면 이렇게 하면 됩니다.

```
# 빈 칸이 들어 있는 파일명을 지우는 알맞은 방법.
rm *\ *
rm *" "*
rm *' '*
# Thank you. S.C.
```

버그 있는 스크립트의 증상을 요약해 보면,

1. syntax error 메세지를 내면서 죽는다
2. 죽지는 않지만 생각했던 대로 동작하지 않는다(로직 에러, logic error).
3. 죽지도 않고 생각했던 대로 동작하지만, 처리하기 까다로운 부효과(side effect)가 있다(로직 폭탄, logic bomb).

제대로 동작하지 않는 스크립트 디버깅에 쓸 수 있는 방법으로는

1. 중요한 부분에 echo 문으로 변수값을 찍어서 어떻게 돌아가고 있는지 살펴 본다.
2. 중요한 부분에 **tee** 필터를 걸어서 프로세스나 데이타 흐름을 확인해 본다.
3. `-n -v -x` 옵션을 건다.

   `sh -n scriptname`는 스크립트를 돌리지는 않고 단순히 문법 에러(syntax error)만 확인합니다. 스크립트 안에서 **`set -n`**이나 **`set -o noexec`**이라고 해도 같은 동작을 합니다. 조심할 점은 이 옵션에 걸리지 않고 그냥 넘어가는 문법 에러도 있다는 것입니다.

   `sh -v scriptname`는 각 명령어를 실행하기 전에 그 명령어를 echo 해 줍니다. 스크립트 안에서 **`set -v`**이나 **`set -o verbose`**라고 해도 같은 동작을 합니다.

   `-n` 과 `-v` 플래그를 같이 쓰면 아주 좋습니다. **`sh -nv scriptname`** 이라고 하면 문법 체크를 아주 자세하게 해줍니다.

   `sh -x scriptname`는 각 명령어의 결과를 간단한 형태로 echo 시켜 줍니다. 스크립트 안에서 **`set -x`**나 **`set -o xtrace`**라고 해도 똑같습니다.

   스크립트에 **`set -u`**나 **`set -o nounset`**을 넣어두면, 선언 없이 쓰이는 변수들에 대해서 unbound variable 에러 메세지를 출력해 줄 것입니다.

4. 스크립트의 아주 중요한 지점에서 변수나 조건을 테스트 하기 위해서 “assert” 함수 쓰기.(이 아이디어는 C 에서 빌려왔습니다.)

예 **30-3. “assert”**로 조건을 테스트하기

```
#!/bin/bash
# assert.sh

assert ()                 #  조건이 거짓이라면,
{                         #+ 에러 메세지를 내고 스크립트를 종료.
  E_PARAM_ERR=98
  E_ASSERT_FAILED=99


  if [ -z "$2" ]          # 매개변수가 맞게 넘어오지 않았음.
  then
    return $E_PARAM_ERR   # 그냥 넘어감.
  fi

  lineno=$2

  if [ ! $1 ]
  then
    echo "Assertion failed:  \"$1\""
    echo "File $0, line $lineno"
    exit $E_ASSERT_FAILED
  # else
  #   return
  #   스크립트를 계속 실행시킴.
  fi
}


a=5
b=4
condition="$a -lt $b"     # 에러 메세지를 내고 스크립트를 종료.
                          #  "condition"을 다른 것으로 바꿔보고
                          #+ 어떻게 되는지 살펴보세요.

assert "$condition" $LINENO
# "assert"가 실패하지 않을 경우에만 다음 부분이 실행됩니다.


# 다른 명령어들.
# ...

exit 0
```

5. exit 잡아채기(trapping at exit).

스크립트에서 **exit**를 쓰면 프로세스 종료를 의미하는 0번 시그널을 날려서 자기 자신을 종료시킵니다.[1] **exit**를 잡아채서(trap) 강제로 변수값을 “출력”하도록 하는 등의 유용한

작업을 할 수 있습니다. 이렇게 하려면 **trap** 명령어가 스크립트의 첫 번째 명령어여야 합니다.

시그널 잡아채기(Trapping signals)

trap

시그널을 받았을 때의 동작을 지정해주는데, 디버깅에 유용하게 쓸 수 있습니다.

참고: 시그널이란 간단히 말해서 프로세스에게 보내는 메세지입니다. 커널이 보내든 다른 프로세스가 보내든지간에 주어진 동작(보통은 종료)을 하라고 말해 주는 것입니다. 예를 들면, 실행중인 프로그램에 **Control-C**를 눌러서 사용자 인터럽트(INT 시그널)를 보낼 수 있습니다.

```
trap ” 2
# 아무 동작도 지정하지 않고 단순히 2번 인터럽트(Control-C)를 무시합니다.

trap ’echo "Control-C는 무시됩니다."’ 2
# Control-C가 눌렸을 때의 메세지.
```

예 30-4. exit 잡아채기(Trapping at exit)

```
#!/bin/bash

trap ’echo Variable Listing --- a = $a  b = $b’ EXIT
# EXIT 는 스크립트가 종료될 때 발생하는 시그널의 이름입니다.

a=39

b=36

exit 0
#  스크립트 파일에 들어 있는 모든 명령어를 다 실행하고 나면
#+ 어쨌든 스크립트가 종료되기 때문에, ’exit’ 명령을 주석 처리해도
#+ 결과가 같음에 주의하기 바랍니다.
```

예 30-5. Control-C 가 눌렸을 때 깨끗이 청소하기

```
#!/bin/bash
# logon.sh: 여러분이 아직도 로그인해 있는지를 확인해 주는 아주 간단한 스크립트.


TRUE=1
LOGFILE=/var/log/messages
# $LOGFILE 은 읽을 수 있어야 됩니다(chmod 644 /var/log/messages).
TEMPFILE=temp.$$
# 이 스크립트의 프로세스 ID로 "유일한" 임시 파일 이름을 만듭니다.
```

```
KEYWORD=address
#  로그인을 하게 되면 /var/log/messages 에
#+ "remote IP address xxx.xxx.xxx.xxx" 란 줄이 덧붙여 집니다.
ONLINE=22
USER_INTERRUPT=13

trap ’rm -f $TEMPFILE; exit $USER_INTERRUPT’ TERM INT
#  스크립트가 Control-C 에 의해 인터럽트를 받았을 경우에 임시 파일을 지워줍니다.

echo

while [ $TRUE ]  # 무한 루프.
do
  tail -1 $LOGFILE> $TEMPFILE
  # 시스템 로그 파일의 마지막 줄을 임시 파일로 저장.
  search=`grep $KEYWORD $TEMPFILE`
  # 성공적인 로그인을 나타내는 "IP address"란 문구가 들어 있는지 확인.

  if [ ! -z "$search" ] # 빈 칸이 들어 있을 수 있기 때문에 쿼우트 필요.
  then
     echo "On-line"
     rm -f $TEMPFILE    # 임시 파일 지우기.
     exit $ONLINE
  else
     echo -n "."        #  연속적인 점들이 찍히도록 -n 옵션으로
                    #+ echo 가 뉴라인을 무시하도록 함.
  fi

  sleep 1
done


#  주의: KEYWORD 변수를 "Exit" 로 바꾸면 로그인 상태에서 갑작스럽게 로그아웃이
#+ 됐는지를 확인해 볼 수 있습니다.

# 연습문제: 위의 주의사항대로 스크립트를 바꾸고 예쁘게 다듬어 보세요.

exit 0


# Nick Drage 가 다른 방법을 제안해 주었습니다:

while true
  do ifconfig ppp0 | grep UP 1> /dev/null && echo "connected" && exit 0
  echo -n "."   # 연결될 때까지 점(.....)을 출력.
  sleep 2
done

# 문제점: 이 스크립트를 끝내기에는 Control-C 를 누르는 것만으로 부족합니다.
```

```
#          (점이 계속 에코됩니다.)
# 연습문제: 이 문제를 해결해 보세요.



# Stephane Chazelas 도 다른 방법을 제안해 주었습니다:

CHECK_INTERVAL=1

while ! tail -1 "$LOGFILE" | grep -q "$KEYWORD"
do echo -n .
   sleep $CHECK_INTERVAL
done
echo "On-line"

# 연습문제: 위에서 설명한 각 방법들의 장점과 단점을 생각해 보세요.
```

참고: **trap** 명령어에 DEBUG 인자를 주면 스크립트의 모든 명령어 다음에 주어진 동작을 수행하도록 합니다. 이는 변수를 추적하는 등의 일을 가능케 합니다.

예 **30-6.** 변수 추적하기

```
#!/bin/bash
# 옮긴이: 이 스크립트가 제대로 동작하려면 #!/bin/bash2 로 바꿔줘야 됩니다.

trap 'echo "추적할 변수> \$variable = \"$variable\""' DEBUG
# 매 명령어마다 $variable 의 값을 에코해 줍니다.

variable=29

echo "\"\$variable\" 은 $variable 로 초기화됨."

let "variable *= 3"
echo "\"\$variable\" 은 3이 곱해짐."

#  "echo $variable" 을 많이 써서 스크립트가 꼴사나와지고
#+ 시간을 많이 잡아먹는 복잡한 스크립트에
#+ "trap 'commands' DEBUG" 를 쓰면 아주 유용하겠죠.

# 이 사실을 알려준 Stephane Chazelas 에게 감사를 표합니다.

exit 0
```

참고: **`trap '' SIGNAL`**(두 개의 붙어 있는 작은 따옴표)이라고 하면 스크립트 나머지 부분에서 SIGNAL을 못 쓰게 합니다. **`trap SIGNAL`**이라고 하면 SIGNAL을 다시 받을 수 있게 해 줍니다. 이는 달갑지 않은 인터럽트로부터 스크립트의 중요한 부분을 보호해 줍니다.

```
trap ″ 2  # 2번 시그널은 Control-C 인데 이제 안 먹힙니다.
command
command
command
trap 2     # Control-C 가 다시 먹게 합니다.
```

주석

1. 관습적으로 *0번 시그널*은 exit로 할당돼 있습니다.

31장. 옵션

옵션은 쉘이나 스크립트의 행동을 변경하도록 설정해 줍니다.

set 명령어는 스크립트 안에서 옵션을 켤 수 있게 해 줍니다. 옵션이 적용될 곳에서 **set -o -option-name** 이라고 하거나 간단하게 **set -option-abbrev** 라고 하면 됩니다. 이 두 가지 형태는 동일하게 동작합니다.

```
#!/bin/bash

set -o verbose
# 모든 명령어는 실행전에 echo 됨.
```

```
#!/bin/bash

set -v
# 위와 완전히 똑같음.
```

참고**:** 스크립트에서 옵션을 끄고 싶다면 **set +o option-name** 이나 **set +option-abbrev** 라고 하세요.

```
#!/bin/bash

set -o verbose
# 명령어가 echo 를 킵니다.
command
...
command

set +o verbose
# 명령어가 echo 를 끕니다.
command
# echo 되지 않습니다.


set -v
# 명령어 echo 를 킵니다.
command
...
command

set +v
```

```
# 명령어 echo 를 끕니다.
command

exit 0
```

스크립트 헤더인 *#!* 뒤에 옵션을 적어줘서 옵션을 켤 수도 있습니다.

```
#!/bin/bash -x
#
# 스크립트 내용이 나옵니다.
```

명령어 줄에서 옵션을 켜는 방법도 있습니다. **set**으로 안 먹는 몇몇 옵션은 이 방법을 쓰면 됩니다. 이런 옵션중 하나인 `-i`는 스크립트를 대화형 모드(interactive)로 돌게 합니다.

`bash -v script-name`

`bash -o verbose script-name`

다음은 몇 개의 유용한 옵션들 목록입니다. 단축형이나 완전한 형태중 하나로 지정해 줄 수 있습니다.

표 **31-1. bash** 옵션들

단축형	이름	뜻
`-C`	noclobber	파일이 재지향에 의해 덮어 써지지 않게 막아줌(>| 를 쓰면 가능할 수 있음)
`-D`	(none)	$ 다음에 나오는 큰 따옴표로 쿼우트된 문자열 목록을 보여주기만 하고 명령어를 실행시키지 않음
`-a`	allexport	정의된 모든 변수를 export 시킴
`-b`	notify	백그라운드로 돌던 작업의 종료를 알려줌(스크립트에서는 그렇게 자주 쓰이지 않음)
`-c ...`	(none)	**...**에서 명령어를 읽어 들임
`-f`	noglob	파일명 확장(globbing)을 끔
`-i`	interactive	스크립트를 대화형**(interactive)** 모드로 돌게함

단축형	이름	뜻
`-p`	privileged	스크립트를 "suid"로 돌게함(조심할 것!)
`-r`	restricted	스크립트를 제한된 모드로 돌게함(**21**장 참고)
`-u`	nounset	정의 안 된 변수 사용시 에러 메세지 출력후 강제 종료
`-v`	verbose	명령어 실행 전에 명령어를 표준출력으로 출력
`-x`	xtrace	`-v`와 비슷하나 명령어를 확장
`-e`	errexit	첫번째 에러에서 스크립트를 취소(0 이 아닌 상태로 종료하는 명령어)
`-n`	noexec	스크립트의 명령어를 읽기만 하고 실행은 안 함(문법 체크)
`-s`	stdin	표준입력에서 명령어를 읽어들임
`-t`	(none)	첫번째 명령어 바로 다음에 종료
`-`	(none)	옵션 플래그의 끝. 나머지 인자들은 모두 위치 매개변수로 인식.
`--`	(none)	위치 인자로 안 받아 들임. 인자가 주어지면(`-- arg1 arg2`), 위치 매개변수는 인자로 세트됨.

32장. 몇 가지 지저분한 것들(Gotchas)

> *투란도트(Turandot): 수수께끼는 세 개, 그러나 죽음은 하나!*
>
> *칼라프(Caleph): 아니오, 수수께끼는 세 개, 생명이 하나!*
>
> *푸치니(Puccini)*

변수명에 예약어나 예약 문자를 할당하기.

```
case=value0       # 문제가 생깁니다.
23skidoo=value1   # 역시 문제가 생깁니다.
# 숫자로 시작하는 변수명은 쉘이 예약해 놓았습니다.
# 대신 _23skidoo=value1 를 쓰세요. 밑줄로 시작하는 변수는 괜찮습니다.

# 하지만...      밑줄로만 된 변수명은 제대로 동작하지 않습니다.
_=25
echo $_           # $_ 은 마지막 명령어의 마지막 인자로 세트되는 특수한 변수입니다.

xyz((!*=value2    # 심각한 문제가 생깁니다.
```

변수명에 하이픈이나 다른 예약 문자를 쓰기.

```
var-1=23
# 대신 'var_1' 를 쓰세요.
```

변수와 함수 이름을 똑같이 쓰기. 이렇게 하면 스크립트를 이해하기가 힘듭니다.

```
do_something ()
{
  echo "이 함수는 \"$1\" 로 뭔가를 합니다."
}

do_something=do_something

do_something do_something

# 문법적으로는 다 맞지만 엄청나게 헷갈리죠?
```

적절치 못한 곳에 공백문자를 쓰는 것(다른 프로그래밍 언어들과는 달리 bash는 공백문자에 대해 좀 까다로운 반응을 보입니다).

```
var1 = 23   # 'var1=23' 이 맞습니다.
# 이렇게 하면 bash는 "="과 "23" 인자를 갖는 "var1" 명령어를 실행시키려고 합니다.

let c = $a - $b   # 'let c=$a-$b' 나 'let "c = $a - $b"' 가 맞습니다.
```

```
if [ $a -le 5]     # if [ $a -le 5 ]    가 맞습니다.
# if [ "$a" -le 5 ]    라고 하면 더 좋겠죠.
# [[ $a -le 5 ]] 도 동작합니다.
```

초기화 안 된 변수(값이 할당되기 전의 변수)가 “0” 인 것으로 가정하기. 초기화 안 된 변수의 값은 0이 아니라 “널”(null)입니다.

테스트 문에서 **=** 과 **-eq** 를 섞어 쓰기.**=** 은 문자를 비교하는 것이고 **-eq** 은 정수를 비교하는 것임을 명심하세요.

```
if [ "$a" = 273 ]       # $a 가 정수인가요? 문자열인가요?
if [ "$a" -eq 273 ]     # $a 가 정수라면.

# 때때로 반대의 결과없이 -eq 와 = 를 섞어 쓸 수 있습니다.
# 하지만...


a=273.0    # 정수가 아닙니다.

if [ "$a" = 273 ]
then
  echo "비교가 잘 됩니다."
else
  echo "비교가 잘 안 됩니다."
fi    # 비교가 잘 안 됩니다.

# a=" 273" 와 a="0273" 도 마찬가집니다.


# 비슷하게, 정수가 아닌 숫자에 "-eq"를 쓰는것도 문제를 일으킵니다.

if [ "$a" -eq 273.0 ]
then
  echo "a = $a'
fi  # 에러 메세지를 내고 abort 됩니다.
# test.sh: [: 273.0: integer expression expected
```

가끔은 “테스트” 대괄호([ ])안에서 쓰이는 변수를 큰 따옴표로 쿼우트 해줘야 할 필요가 있습니다. 이렇게 안 하면 예상치 못한 결과가 나올 수 있습니다. 예 7-5, 예 16-2, 예 9-5를 참고하세요.

스크립트의 소유자가 스크립트에서 쓰이는 명령어에 대한 실행 권한이 없어서 그 명령어를 실행시키지 못할 수도 있습니다. 명령어 줄에서 실행 시키지 못하는 명령어는 스크립트에서 실행시킨다 해도 역시 실패할 것입니다. 해결책은, 문제가 되는 명령어의 속성을 바꿔보는데, 정 안 되면 suid 비트를 설정해 보기 바랍니다.(당연히 루트로 해야겠죠).

- 를 재지향 연산자(실은 아니지만)처럼 쓰려고 하면 보통은 이상한 결과를 가져옵니다.

```
command1 2> - | command2  # command1의 에러 출력을 파이프
로 재지향 하려고 하지만...
#    ...제대로 동작하지 않을 겁니다.

command1 2>& - | command2  # 역시 실패합니다.

Thanks, S.C.
```

Bash 버전 2 이상에만 들어 있는 기능은 에러 메세지를 내면서 실패할 경우가 있습니다. 오래된 리눅스 머신의 경우에는 기본 설치시 Bash 버전 1.XX 대가 깔려 있을 수 있습니다.

```
#!/bin/bash

minimum_version=2
# Chet Ramey 가 Bash 에 기능을 계속 추가하고 있기 때문에
# $minimum_version 을 2.XX 나 적당한 버전으로 세트하면 됩니다.
E_BAD_VERSION=80

if [ "$BASH_VERSION" \< "$minimum_version" ]
then
  echo "이 스크립트는 Bash 버전 $minimum_version 이나 그 이상에서만 동작합니다."
  echo "Bash 를 업그레이드할 것을 강력히 추천합니다."
  exit $E_BAD_VERSION
fi

...
```

리눅스 머신이 아닌 곳의 본 쉘 스크립트(**#!/bin/sh**)에서 Bash 전용 기능을 쓰게 되면 예상치 못한 동작을 할 수 있습니다. 리눅스에서는 **sh**이 보통 **bash**를 나타내지만 일반적인 유닉스에서도 항상 이렇지는 않습니다.

도스 형태의 뉴라인(*\r\n*)으로 된 스크립트는 실행되지 않는데, **#!/bin/bash\r\n**이 우리가 원하는 **#!/bin/bash\n**과 같지 않아서 인식할 수 없기 때문입니다. 이럴 때는 그 스크립트가 유닉스 식의 뉴라인을 갖도록 변환해 줘야 합니다.

스크립트 헤더가 **#!/bin/sh**이라고 되어 있으면 완전한 Bash 호환 모드로 동작하지 않을 수 있습니다. 몇몇 Bash 전용 기능들을 못 쓸 수도 있습니다. 완전한 Bash 전용 확장 기능을 쓰고 싶은 스크립트는 헤더를 **#!/bin/bash**라고 해 주면 됩니다.

스크립트는 변수를 자기 부모 프로세스나 쉘, 환경으로 **export** 할 수 없습니다. 우리가 생물학 시간에 배웠듯이 자식은 부모에게서 물려 받지만 그 반대는 안 됩니다.

```
WHATEVER=/home/bozo
export WHATEVER
exit 0
```

```
bash$ echo $WHATEVER

bash$
```

역시, 명령어 프롬프트로 돌아가서 보면 $WHATEVER는 세트되어 있지 않습니다.

서브쉘 안에서 변수를 세트하고 조작한 다음 서브쉘 바깥에서 똑같은 변수를 사용하려고 한다면 별로 유쾌하지 않은 결과를 가져 옵니다.

예 **32-1.** 서브쉘 함정**(Subshell Pitfalls)**

```
#!/bin/bash
# 서브쉘 변수의 함정.

outer_variable=outer
echo
echo "outer_variable = $outer_variable"
echo

(
# 서브쉘 시작

echo "서브쉘 안에서의 outer_variable = $outer_variable"
inner_variable=inner  # 셋
echo "서브쉘 안에서의 inner_variable = $inner_variable"
outer_variable=inner  # 변수값 변경이 전역적으로 영향을 받을까요?
echo "서브쉘 안에서의 outer_variable = $outer_variable"

# 서브쉘 끝
)

echo
echo "서브쉘 밖에서의 inner_variable = $inner_variable"  # 언셋.
echo "서브쉘 밖에서의 outer_variable = $outer_variable"  # 안 변했죠.
echo

exit 0
```

스크립트 안에서 “suid” 명령어를 쓰면 시스템 보안을 위협할 수 있습니다. [1]

쉘 스크립트를 CGI 프로그래밍에 쓰는 것은 문제가 있습니다. 쉘 스크립트 변수는 “타입 세이프”(typesafe)하지 않기 때문에 CGI에 대해서만은 원하는 결과를 가져 오지 않을 수 있습니다. 게다가 “크래커 대해서 안전한”(cracker-proof) 쉘 스크립트를 짜는 것은 굉장히 어렵습니다.

Danger is near thee --

Beware, beware, beware, beware.

Many brave hearts are asleep in the deep.

So beware --

Beware.

A.J. Lamb and H.W. Petrie

주석

1. 스크립트에 **suid** 소유권을 거는 것은 무의미 합니다.

33장. 스타일 있게 스크립트 짜기

쉘 스크립트를 구조적이고 조직적으로 작성하는 방법에 익숙해 지기 바랍니다. 책상 앞에 앉아서 코딩하기 전에 계획하고, 생각을 정리할 시간을 조금이라도 갖으면 “급하게” 짰거나 “편지 봉투 뒤에 끄적인” 스크립트라 할지라도 큰 도움이 될 것입니다.

이와 함께 약간의 스타일 가이드 라인을 제시하겠습니다. 이는 공식 쉘 스크립팅 스타일시트를 제시하기 위한 것이 아닙니다.

33.1. 비공식 쉘 스크립팅 스타일시트

- 코드에 주석을 다세요. 주석을 달면 다른 사람이 이해하거나 인식하기 쉽게 해 주고, 여러분이 그 스크립트를 관리하기 편하게 해 줍니다.

  ```
  PASS="$PASS${MATRIX:$(($RANDOM%${#MATRIX})):1}"
  # 여러분이 작년에 이걸 짰을 때는 아주 완벽하게 이해를 했겠지만 지금은 완전히 수수께
  끼입니다.
  # (Antek Sawicki 의 "pw.sh" 스크립트에서 인용)
  ```

  스크립트와 함수에 간단한 설명을 담고 있는 헤더를 추가하세요.

  ```
  #!/bin/bash

  #************************************************#
  #                   xyz.sh                       #
  #           written by Bozo Bozeman              #
  #                July 05, 2001                   #
  #                                                #
  #           프로젝트 파일들 정리하기.            #
  #************************************************#

  BADDIR=65                       # 존재 하지 않는 디렉토리.
  projectdir=/home/bozo/projects  # 정리할 디렉토리.

  #--------------------------------------------------------#
  # cleanup_pfiles ()
  # 지정된 디렉토리에 들어 있는 모든 파일을 지움.
  # 매개변수: $target_directory
  # 리턴: 성공시 0, 잘못되면 $BADDIR.
  #--------------------------------------------------------#
  cleanup_pfiles ()
  {
    if [ ! -d "$1" ]  # 해당 디렉토리가 존재하는 확인.
    then
      echo "$1 는 디렉토리가 아닙니다."
      return $BADDIR
  ```

```
  fi

  rm -f "$1"/*
  return 0   # 성공.
}

cleanup_pfiles $projectdir

exit 0
```

주석을 달기 전에 스크립트 첫번째 줄에 **#!/bin/bash**를 두는 것을 잊으면 안됩니다.

- “매직 넘버”[1] (“고정된”, hard-wired 문자 상수) 쓰지 않기. 그 대신 의미 있는 변수 이름을 쓰기 바랍니다. 이렇게 하면 스크립트를 이해하기 쉽고, 변경 가능하면서 추가적인 분석 없이도 업데이트가 가능하게 합니다.

```
if [ -f /var/log/messages ]
then
  ...
fi
# 일년후, 이 스크립트가 /var/log/syslog 를 확인하도록 변경하려고 합니다.
# 이 부분이 나오는 곳을 손으로 일일이 다 찾아서 바꿔줘야 합니다.
# 그리고 잘못되지 않기를 바래야 합니다.

# 더 좋은 방법:
LOGFILE=/var/log/messages  # 여기만 변경하면 됩니다.
if [ -f $LOGFILE ]
then
  ...
fi
```

- 서술적인 변수명을 고르세요.

```
fl=`ls -al $dirname`              # 암호같죠?
file_listing=`ls -al $dirname`    # 훨씬 좋네요.


MAXVAL=10   # 스크립트 상수로 쓰이는 변수명은 모두 대문자로 쓰세요.
while [ "$index" -le "$MAXVAL" ]
...


E_NOTFOUND=75                    # 에러 코드용 변수는 "E_"로 시작하고
                                 # 모두 대문자로.
if [ ! -e "$filename" ]
then
  echo "$filename 을 찾을 수 없습니다."
  exit $E_NOTFOUND
fi
```

```
MAIL_DIRECTORY=/var/spool/mail/bozo  # 환경 변수에는 모두 대문자.
export MAIL_DIRECTORY


GetAnswer ()                         # 함수명에는 대소문자를 섞어 쓰면 좋습니
다.
{
  prompt=$1
  echo -n $prompt
  read answer
  return $answer
}

GetAnswer "제일 좋아하는 숫자는? "
favorite_number=$?
echo $favorite_number


_uservariable=23                # 가능하지만 추천하지 않습니다.
# 사용자용 변수는 밑줄로 시작하지 않는 것이 좋습니다.
# 밑줄로 시작하는 변수는 시스템용으로 남겨 두기 바랍니다.
```

- 조직적이고 의미 있는 종료 코드를 쓰세요.

```
E_WRONG_ARGS=65
...
...
exit $E_WRONG_ARGS
```

부록 C 참고.

- 복잡한 스크립트를 간단한 모듈로 나누고, 적당한 곳에서 함수를 사용하세요. 예 35-3를 참고.
- 간단한 것으로 할 수 있는데 복잡하게 하지 마세요.

```
COMMAND
if [ $? -eq 0 ]
...
# 중복되고 직관적이지 않습니다.

if COMMAND
...
# 더 간단하죠(읽기 쉽지는 않을지 몰라도).
```

주석

1. 이 문맥에서 쓰인 “매직 넘버”는 파일 타입을 지정하기 위한 매직 넘버와 전혀 다른 뜻으로 쓰이고 있습니다.

34장. 자질구레한 것들

> *비록 본 쉘의 소스 코드를 들여다 보는게 약간의 도움이 될 지는 몰라도 그 문법은 아무도 알지 못한다.*
>
> *Tom Duff*

34.1. 대화(interactive)형 모드와 비대화(non-interactive)형 모드 쉘과 스크립트

대화형 모드의 쉘은 명령어를 `tty` 의 사용자 입력으로부터 읽어 들입니다. 이 대화형 모드 쉘은 그 중에서도 특히나 기본적으로, 활성화시 시작 파일(startup)을 읽어 들이고, 프롬프트를 표시해 주고, 작업 제어를 해줍니다. 사용자는 쉘과 대화를 할 수 있게 됩니다.

스크립트를 실행시키는 쉘은 항상 비대화형 모드 쉘로 동작하지만 계속 자신의 `tty`에 접근할 수 있습니다. 심지어는 스크립트에서 대화형 모드 쉘을 흉내내는 것도 가능합니다.

```
#!/bin/bash
MY_PROMPT='$ '
while :
do
  echo -n "$MY_PROMPT"
  read line
  eval "$line"
done

exit 0

# 이 예제 스크립트와 위에서 설명한 많은 부분들은 Stephane Chazelas 가
# 제공해 주었습니다.
```

보통 read 문(예 11-2 참고)으로 사용자가 입력을 받아 들이는 스크립트를 대화형 모드 스크립트라고 합시다. "실제 세계"에서는 이것보다 조금 더 복잡하지만, 지금 당장은 tty 에 연결되어 있고, 사용자가 콘솔이나 한텀에서 실행시킨 스크립트를 대화형 모드 스크립트라고 가정합니다.

초기화(init)와 시스템 구동(startup) 스크립트는 사람의 개입없이 돌아야 하기 때문에 반드시 비대화형 모드여야 합니다. 많은 수의 관리용이나 시스템 유지용 스크립트 역시 비대화형 모드 스크립트입니다. 항상 같고 반복적인 작업의 자동화에는 비대화형 모드 스크립트가 필요합니다.

비대화형 모드 스크립트는 백그라운드로 돌 수 있지만, 대화형 모드 스크립트는 결코 받을 수 없는 사용자 입력을 기다리면서 멈춰 있게 됩니다. 이렇게 백그라운드로 도는 대화형 모드 스크립트에 대해서는 **expect** 스크립트나 스크립트 안에 here document를 사용해서 입력을 집

어 넣을 수 있습니다. 가장 간단한 방법은 **read** 문에 입력 재지향을 걸어서(**read variable <file**) 파일에서 입력을 받도록 하는 것입니다. 이렇게 까다로운 몇 가지 해결책 덕분에 평범한 용도의 스크립트들이 대화형 모드나 비대화형 모드 양 쪽에서 돌 수 있습니다.

만약에 스크립트가 현재 대화형 모드로 돌고 있는지 아닌지는 단순히 프롬프트 변수인 $PS1이 세트됐는지 아닌지로 알아낼 수 있습니다(사용자가 입력을 할 차례라면(being prompted), 스크립트는 프롬프트를 표시해야 하기 때문입니다).

```
if [ -z $PS1 ] # 프롬프트가 없다?
then
  # 비대화형 모드
  ...
else
  # 대화형 모드
  ...
fi
```

다른 방법으로는, $- 플래그에서 "i" 옵션이 있는지 없는지로 확인할 수도 있습니다.

```
case $- in
*i*)    # 대화형 모드 쉘
;;
*)      # 비대화형 모드 쉘
;;
# (1993년 "UNIX F.A.Q."에서 인용)
```

> 참고**:** 스크립트에 -i 옵션을 주거나 **#!/bin/bash -i** 헤더를 주면 강제로 대화형 모드로 동작하도록 할 수 있습니다. 이렇게 했을 경우에는 스크립트가 엉뚱하게 동작한다거나 에러가 없는데도 에러 메세지를 보여줄 수도 있기 때문에 조심해야 합니다.

34.2. 쉘 래퍼(Shell Wrappers)

"래퍼"란 시스템 명령어나 유틸리티들을 해당 매개변수들과 함께 쉘 스크립트로 만들어 두는 것을 말합니다. 명령어줄에서 아주 복잡하게 부를 명령어를 스크립트로 만들어 두면 실행시킬 때 아주 간단해 집니다. 특히 sed와 awk에 이 쉘 래퍼를 쓰면 아주 좋습니다.

sed 나 **awk** 스크립트는 명령어 줄에서 **`sed -e 'commands'`**라고 치거나 **`awk 'commands'`**이라고 쳐서 실행 시킬 수 있습니다. 이런 스크립트를 Bash 스크립트 안에서 사용하게 되면 좀 더 간단하게 쓸 수 있고 "재사용"할 수 있습니다. 예를 들어 **sed**의 출력과 **awk**의 입력을 파이프로 연결하는 것처럼 sed와 awk의 기능을 연결해서도 쓸 수도 있습니다. 저장된 실행 파일이라면 다시 치는 불편함 없이 원래 형태나 약간 변경된 형태로 계속 실행 시킬수가 있습니다.

예 **34-1.** 쉘 래퍼**(shell wrapper)**

```
#!/bin/bash

# 파일에서 빈 줄을 지워주는 간단한 스크립트.
# 인자 확인 안 함.

# 명령어줄에서
#    sed -e '/^$/d' filename
# 이라고 하는 것과 똑같음.

sed -e /^$/d "$1"
# '-e'는 다음에 나오는 것을 "편집" 명령어로 해석(여기서는 넣어도 되고 안 넣어도 됨).
# '^'는 줄 처음, '$'는 줄 끝.
# 줄 처음과 줄 끝 사이에 아무 것도 없는 것(빈 줄)과 일치.
# 'd' 는 삭제 명령어.

#  명령어줄 인자를 쿼우트 해 주면 파일이름에
#+ 공백문자나 특수문자가 들어가도 상관없습니다.

exit 0
```

예 **34-2.** 조금 복잡한 쉘 래퍼**(shell wapper)**

```
#!/bin/bash

# "subst", 파일에서 어떤 패턴을 다른 패턴으로 바꿔주는 스크립트.
# 즉, "subst Smith Jones letter.txt".

ARGS=3
E_BADARGS=65   # 필요한 인자가 빠져있음.

if [ $# -ne "$ARGS" ]
# 스크립트로 넘겨진 인자의 갯수를 확인(항상 이렇게 하세요).
then
  echo "사용법: `basename $0` old-pattern new-pattern filename"
  exit $E_BADARGS
fi

old_pattern=$1
new_pattern=$2

if [ -f "$3" ]
then
    file_name=$3
else
    echo "\"$3\" 은 없는 파일입니다."
    exit $E_BADARGS
fi
```

```
# 여기가 가장 중요한 부분입니다.
sed -e "s/$old_pattern/$new_pattern/g" $file_name
# 's'는 sed의 치환(substitution) 명령어이고,
# /pattern/ 은 주소 매칭을 실행시킵니다.
#  전역(global) 플래그인 "g"를 쓰면 단지 첫번째 일치하는 $old_pattern만
#+ 치환시키지 않고 각 줄에서 일치하는 "모든" $old_pattern을 치환시킵니다.
# 더 자세한 설명은 'sed' 문서를 읽어보세요.

exit 0    # 스크립트의 실행이 성공이라면 0을 리턴.
```

예 34-3. awk 스크립트 쉘 래퍼(shell wrapper)

```
#!/bin/bash

# 대상 파일에서 주어진 열을 다 더하기.

ARGS=2
E_WRONGARGS=65

if [ $# -ne "$ARGS" ] # 명령어줄 인자 수가 적당한지 확인.
then
   echo "사용법: `basename $0` filename column-number"
   exit $E_WRONGARGS
fi

filename=$1
column_number=$2

# 쉘 변수를 awk 에게 넘기는 것은 약간 까다롭습니다.
# 더 자세한 것은 awk 문서를 참고하세요.

# 여러줄짜리 awk 스크립트는    awk ' ..... '   처럼 부르면 됩니다.


# awk 스크립트 시작.
# -----------------------------
awk '

{ total += $'"${column_number}"'
}
END {
     print total
}

' "$filename"
# -----------------------------
# awk 스크립트 끝.
```

```
#   쉘 스크립트에 내장된 awk 스크립트에게 쉘 변수를 넘기는 것은
#   안전하지 않을 수가 있기 때문에, Stephane Chazelas 가 다음과 같은 대안을
#   제안해 주었습니다:
#   ---------------------------------------
#   awk -v column_number="$column_number" '
#   { total += $column_number
#   }
#   END {
#       print total
#   }' "$filename"
#   ---------------------------------------


exit 0
```

하나로 된 만능 툴이나 스위스 군용 칼 같은 기능이 필요한 스크립트들을 위해 펄이 있습니다. 펄은 sed와 awk의 기능을 묶어 **C** 의 큰 서브셋으로 만듭니다. 모듈로 동작하며 객체 지향 프로그래밍에서 부엌 싱크대까지 모든 것을 지원합니다. 짧은 펄 스크립트는 쉘 스크립트 상에서 효과적으로 쓰일 수 있고, 펄이 쉘 스크립트를 완전히 대체할 수 있을 거란 얘기도 있지만 저자는 그렇게 생각하지 않습니다.

예 **34-4. Bash** 스크립트에 내장된 펄

```
#!/bin/bash

# 펄 스크립트보다 쉘 스크립트가 먼저 올 수 있습니다.
echo "\"$0\" 에서 내장된 펄 스크립트보다 먼저 등장."
echo "=================================================================="

perl -e 'print "내장된 펄 스크립트 부분.\n"'
# sed 처럼 펄도 "-e" 옵션을 씁니다.

echo "=================================================================="
echo "하지만, 스크립트에는 쉘이나 시스템 명령어가 올 수도 있습니다."

exit 0
```

심지어는 Bash 스크립트와 펄 스크립트를 한 파일에서 같이 쓸 수도 있습니다. 그 스크립트가 어떻게 불리냐에 따라 Bash 부분이 실행될지 펄 부분이 실행될지가 결정됩니다.

예 **34-5.** 하나로 묶인 **Bash** 스크립트와 펄 스크립트

```
#!/bin/bash
# bashandperl.sh

echo "스크립트의 Bash 쪽에서 보내는 축하 메세지."
```

```
# 이 다음에 Bash 명령어가 더 올 수 있습니다.

exit 0
# Bash 부분의 끝.

# =========================================================

#!/usr/bin/perl
# 이 부분은 -x 옵션을 줘야 실행됩니다.

print "스크립트의 펄 쪽에서 보내는 축하 메세지.\n";
# 이 다음에 펄 명령어가 더 올 수 있습니다.

# 펄 부분의 끝.
```

```
bash$ bash bashandperl.sh
스크립트의 Bash 쪽에서 보내는 축하 메세지.


bash$ perl -x bashandperl.sh
스크립트의 펄 쪽에서 보내는 축하 메세지.
```

34.3. 테스트와 비교: 다른 방법

테스트문에 있어서는, [[ ]] 가 `[ ]` 보다는 더 적당할 수도 있습니다. 비슷하게, 산술 연산에 있어서도 (( ))를 쓰는것이 더 나을 수도 있습니다.

```
a=8

# 다음에 나오는 비교 연산은 모두 똑같습니다.
test "$a" -lt 16 && echo "yes, $a < 16"         # "and list"
/bin/test "$a" -lt 16 && echo "yes, $a < 16"
[ "$a" -lt 16 ] && echo "yes, $a < 16"
[[ $a -lt 16 ]] && echo "yes, $a < 16"          # [[ ]] 와 (( )) 안에 들
어 있는
(( a < 16 )) && echo "yes, $a < 16"             # 변수는 쿼우트해 줄 필요가 없
습니다.

city="New York"
# 마찬가지로, 다음에 나오는 비교 연산도 모두 같은 겁니다.
test "$city" \< Paris && echo "Paris가 $city 보다 더 커요."  # 아스키 순서에
서 더 커요.
/bin/test "$city" \< Paris && echo "Paris가 $city 보다 더 커요."
[ "$city" \< Paris ] && echo "Paris가 $city 보다 더 커요."
```

```
[[ $city < Paris ]] && echo "Paris가 $city 보다 더 커요."   # $city 를 쿼우
트 안 해줘도 됩니다.

# Thank you, S.C.
```

34.4. 최적화

대부분의 쉘 스크립트로는 별로 복잡하지 않은 문제를 간단하게 풀 수 있습니다. 이런 경우에는 속도를 위한 최적화는 별 이슈가 되지 않습니다. 하지만 아주 중요한 작업을 수행하는 스크립트일 경우, 제대로 돌긴 하지만 속도가 너무 느린 경우를 생각해 봅시다. 이 스크립트를 컴파일 언어로 다시 작성하는 것은 별로 좋은 생각이 아닙니다. 이럴 경우에 가장 간단한 방법은 속도를 떨어뜨리는 부분만을 다시 작성하는 것입니다. 코드 최적화의 이론들을 이 초라한 쉘 스크립트에도 적용할 수가 있을까요?

스크립트에 나오는 루프문을 살펴 보세요. 반복적인 연산들은 시간을 많이 잡아 먹습니다. time이나 times를 써서 계산을 많이 하는 명령어들의 시간을 재 보기 바랍니다. 어떤 부분에서 시간이 중요한 변수라면 이 부분을 C 나 어셈블러로 다시 짜는 것도 고려해 보기 바랍니다.

파일 I/O를 최소화 하세요. Bash 는 파일을 다루는데 특별히 효과적이지 않기 때문에 파일 연산이 필요한 곳에서는 awk 나 펄처럼 더 적당한 도구를 쓰는 것도 좋은 방법입니다.

스크립트를 구조적이고 논리적으로 만드세요. 이렇게 하면 나중에 필요할 때 다시 작성하거나 더 업데이트 된 버전을 만들수 있습니다. 고급 언어에 적용할 수 있는 몇몇 최적화 기법들도 스크립트에 써 먹을 수 있지만 루프 unrolling 같은 것은 거의 무의미합니다. 무엇보다도 상식적인 선에서 해결하도록 하세요.

34.5. 팁 모음(Assorted Tips)

- 어떤 사용자가 어떤 특정한 세션이나 얼마 이상의 세션동안 스크립트를 돌렸는지를 보고 싶다면 각 스크립트마다 다음 줄들을 넣어 주면 됩니다. 이렇게 하면 스크립트 이름과 실행 시간의 기록들을 한 파일에서 계속 추적할 수 있게 됩니다.

  ```
  # 추적할 스크립트의 끝에 다음을 추가(>>)하세요.

  date>> $SAVE_FILE       # 날짜와 시간.
  echo $0>> $SAVE_FILE    # 스크립트 이름.
  echo>> $SAVE_FILE       # 빈 줄 구분자.

  # 당연한 얘기지만, SAVE_FILE 은 ~/.scripts-run 같은 이름으로
  # ~/.bashrc 에 환경 변수로 정의하고 export 시켜야 됩니다.
  ```

-

>> 연산자는 파일의 끝에 줄들을 추가해 줍니다. 그럼 이미 존재하는 파일의 맨 앞에 추가(prepend)하려면 어떻게 할까요?

```
file=data.txt
title="***데이타 텍스트 파일의 제목 줄입니다***"

echo $title | cat - $file >$file.new
# "cat -" 은 표준출력을 $file 과 연결시켜 줍니다.
#  $title 이 "맨 앞"에 추가된 새 파일이 만들어 집니다.
```

당연한 이야기지만, sed 도 이렇게 할 수 있습니다.

- 쉘 스크립트는 **Tcl**이나 **wish** 스크립트, 심지어는 Makefile 에서 내장 명령어처럼 동작할 수 있습니다. C 프로그램에서 *`system()`* 함수를 써서 *`system("script_name");`*이라고 외부 쉘 명령어를 부르는 것처럼 부를 수도 있습니다.
- 여러분이 자주 쓰고 유용해 보이는 정의들이나 함수들은 하나로 모아서 한 파일에 둔 다음, 다른 스크립트에서 이 파일을 도트(dot, **.**) 명령어나 source 명령어로 "포함"(include)해서 쓰도록 하세요.

```
# SCRIPT LIBRARY
# ------ -------

# 주의:
# "#!" 를 쓰면 안 됨.
# "실제 코드" 도 안 됨.


# 유용한 변수 정의

ROOT_UID=0              # 루트는 $UID 0.
E_NOTROOT=101           # 루트 사용자가 아닌 에러.
MAXRETVAL=256           # 함수의 최대(양수) 리턴값.
SUCCESS=0
FAILURE=-1



# 함수

Usage ()                # "사용법:" 메세지.
{
  if [ -z "$1" ]        # 넘어온 인자 없음.
  then
    msg=filename
  else
    msg=$@
  fi
```

```
  echo "사용법: `basename $0` "$msg""
}


Check_if_root ()       # 스크립트를 루트로 돌리는지 확인.
{                      # "ex39.sh" 예제에서 발췌.
  if [ "$UID" -ne "$ROOT_UID" ]
  then
    echo "이 스크립트는 루트로 실행시켜야 됩니다."
    exit $E_NOTROOT
  fi
}


CreateTempfileName ()  # "유일한" 임시 파일을 생성.
{                      # "ex51.sh" 예제에서 발췌.
  prefix=temp
  suffix=`eval date +%s`
  Tempfilename=$prefix.$suffix
}


isalpha2 ()            # "전체 문자열"이 알파벳으로만 되어 있는지 확인.
{                      # "isalpha.sh" 에서 발췌.
  [ $# -eq 1 ] || return $FAILURE

  case $1 in
  *[!a-zA-Z]*|"") return $FAILURE;;
  *) return $SUCCESS;;
  esac                 # Thanks, S.C.
}


abs ()                            # 절대값.
{                                 # 주의: 최대 리턴값 = 256.
  E_ARGERR=-999999

  if [ -z "$1" ]                  # 인자가 필요함.
  then
    return $E_ARGERR              # 명백한 에러값이 리턴.
  fi

  if [ "$1" -ge 0 ]               # 음수가 아니면,
  then                            #
    absval=$1                     # 그냥 그대로.
  else                            # 음수면,
    let "absval = (( 0 - $1 ))"   # 부호를 변경.
  fi
```

```
  return $absval
}
```

- 스크립트를 좀 더 명확하고 읽기 쉽게 하기 위해서 특별한 용도의 주석을 쓰세요.

```
## 경고.
rm -rf *.zzy   ##  "rm" 에 "-rf" 옵션을 주면 아주 위험하고
               ##+ 특히나 와일드 카드와 같이 쓰면 더욱 위험합니다.

#+ 연속되는 줄.
#  이건 여러줄짜리 주석의
#+ 첫번째 줄이고,
#+ 여긴 마지막 줄.

#* 주의.

#o 리스트 아이템.

#> 다른 관점.
while [ "$var1" != "end" ]    #> while test "$var1" != "end"
```

- $? 종료 상태 변수를 쓰면 매개변수가 오직 숫자로만 이루어졌는지 확인할 수 있고 그렇다면 그 매개변수를 정수로 처리할 수 있습니다.

```
#!/bin/bash

SUCCESS=0
E_BADINPUT=65

test "$1" -ne 0 -o "$1" -eq 0 2>/dev/null
# 정수는 0 과 같거나 0 과 같지 않습니다.
# 2>/dev/null 는 에러 메세지를 없애 줍니다.

if [ $? -ne "$SUCCESS" ]
then
  echo "사용법: `basename $0` integer-input"
  exit $E_BADINPUT
fi

let "sum = $1 + 25"             # $1 이 정수가 아니라면 에러를 냅니다.
echo "Sum = $sum"

# 명령어줄 매개변수뿐만 아니라 어떤 변수라도 이런식으로 테스트 할 수 있습니다.

exit 0
```

- 이중 중괄호를 쓰면 for 나 while 루프에서도 C 형태의 문법으로 변수를 세팅하거나 증가시킬수 있습니다. 예 10-11와 예 10-16를 참고하세요.

- run-parts 명령어는 여러 명령어 스크립트를 차례대로 실행시킬 때 편한데 특히 cron 이나 at과 같이 쓰면 더욱 편합니다.
- 쉘 스크립트에서 X 윈도우 위젯을 부를 수 있다면 아주 멋지겠죠. **Xscript**나 **Xmenu**, **widtools** 같은 것들을 쓰면 X 윈도우 위젯을 부를 수 있습니다. 앞의 두 개는 더 이상 개발되지 않는 것 같은데, 다행스럽게도 **widtools**은 여기 (http://www.batse.msfc.nasa.gov/~mallozzi/home/software/xforms/src/widtools-2.0.tgz)에서 구할 수 있습니다.

> **경고**
>
> **widtools**(widget tools)은 설치할 때 **XForms** 라이브러리가 있어야 됩니다. 게다가 일반적인 리눅스 시스템에서 빌드하기 전에 `Makefile`을 조심스럽게 편집해야 하는 번거로움도 있습니다. 끝으로, 제공되는 여섯개의 위젯중에 세개는 제대로 동작하지 않고 세그폴트를 일으킵니다.

위젯을 이용해서 더 효과적인 스크립팅을 하려면 **Tk**나 **wish**(**Tcl**에서 나왔죠), **PerlTk**(펄용 Tk 확장), **tksh**(ksh용 Tk 확장), **XForms4Perl**(펄용 XForms 확장), **Gtk-Perl**(펄용 Gtk 확장), **PyQt**(파이썬용 Qt 확장)를 써 보세요.

34.6. 괴상한 것(Oddities)

과연 스크립트가 자신을 재귀적으로 부를 수 있을까요?

예 **34-6.** 자신을 재귀적으로 부르는 스크립트

```
#!/bin/bash
# recurse.sh

#  스크립트가 자신을 재귀적으로 부를 수 있을까요?
#  부를 수 있지만, "개념 증명"(proof of concept) 따위의 쓰임새가 아니라면,
#+ 전혀 실용적이지 않습니다.

RANGE=10
MAXVAL=9

i=$RANDOM
let "i %= $RANGE"  # 0 에서 $MAXVAL 사이의 랜덤한 숫자 만들기.

if [ "$i" -lt "$MAXVAL" ]
then
  echo "i = $i"
  ./$0             #   스크립트는 자기 자신의 새 인스턴스를
                   #+  재귀적으로 만들어 냅니다.
                   #   만들어내 $i 가 $MAXVAL 과 같을 때까지
```

```
                         #+  각 자식 스크립트도 똑같은 동작을 합니다.

#  "if/then" 대신 "while" 루프를 쓰면 문제가 생깁니다.
#  독자들을 위한 연습문제: 왜 문제가 생길까요?

exit 0
```

> **경고**
>
> 재귀가 너무 많이 일어나면 스크립트의 스택 영역을 다 써버리기 때문에 세그폴트가 납니다.

34.7. 이식성 문제(Portability Issues)

이 책은 특별히 GNU/Linux 시스템에서의 Bash 스크립팅을 다룹니다만, **sh**이나 **ksh** 사용자도 많은 것을 배울 수 있을 것입니다.

공교롭게도 여러 쉘과 스크립팅 언어들은 POSIX 1003.2 표준으로 움직이고 있는 것처럼 보입니다. `--posix` 옵션을 주거나 헤더에 **set -o posix** 라고 해서 Bash 를 부르면 이 POSIX 표준과 거의 호환이 되게 실행해 줍니다. 이렇게 하지 않더라도 Chet Ramey 가 **ksh**의 기능을 Bash 최신 버전으로 부지런히 포팅하고 있기 때문에, 거의 대부분의 Bash 스크립트들은 별다른 수정없이 **ksh**에서 잘 동작하고 반대도 잘 동작할 것입니다.

상업용 유닉스 머신에서 표준 명령어의 GNU 전용 기능들을 사용하게 되면 제대로 동작하지 않을 수도 있습니다. 하지만 이 문제는 최근 몇년동안 조금씩 문제가 되질 않고 있는데, "거대"(big-iron) 유닉스에서 쓰이는 특허가 걸린 유틸리티들이 점차 GNU 용으로 바뀌고 있기 때문입니다. 칼데라에서 최근 내놓고 있는 원래의 많은 유닉스 유틸리티의 소스들은 이런 경향을 점차 더 가속화 시키고 있습니다.

34.8. 윈도우즈에서의 쉘 스크립팅

다른 OS 를 쓰는 사용자라도 유닉스 식의 쉘 스크립트를 쓸 수 있기 때문에 이 책에서 소개한 많은 것들을 써서 이익을 얻을 수 있습니다. Cygnus의 Cygwin (http://sourceware.cygnus.com/cygwin/) 패키지나 Mortice Kern Associates 의 MKS 유틸리티 (http://www.mkssoftware.com/)를 쓰면 윈도우즈에서도 쉘 스크립트의 기능을 쓸 수 있습니다.

35장. Bash, 버전 2

지금 여러분이 시스템에서 쓰고 있는 **Bash** 는 버전 2.XX 대 입니다.

```
bash$ echo $BASH_VERSION
2.04.21(1)-release
```

예전 버전에 비해 배열 변수[1] 문자열및 매개변수 확장, 향상된 변수 간접 참조등을 포함한 다른 특징들이 업데이트 됐습니다.

예 35-1. 문자열 확장

```
#!/bin/bash

# 문자열 확장.
# Bash 버전 2 부터 소개됨.

# $'xxx' 형태의 문자열들은 표준 이스케이프 문자로 해석됩니다.

echo $'벨 3번 울리기 \a \a \a'
echo $'3번의 폼피드(form feed) \f \f \f'
echo $'10번의 뉴라인(newline) \n\n\n\n\n\n\n\n\n\n'

exit 0
```

예 35-2. 간접 변수 참조 **-** 새로운 방법

```
#!/bin/bash

# 변수 간접 참조.
# C++ 의 참조 특성이 약간 가미되었습니다.


a=letter_of_alphabet
letter_of_alphabet=z

echo "a = $a"           # 직접 참조.

echo "Now a = ${!a}"    # 간접 참조.
# ${!variable} 표기법은 예전의 "eval var1=\$$var2" 보다 훨씬 좋습니다.

echo

t=table_cell_3
table_cell_3=24
echo "t = ${!t}"        # t = 24
```

```
table_cell_3=387
echo "t 의 값이 ${!t} 로 바뀌었습니다."    # 387

#  배열이나 테이블 멤버를 참조할 때나
#+ 다차원 배열을 시뮬레이션 할 경우에 쓸 만 합니다.
# 인덱싱을 옵션으로 가졌다면 더 좋았을 겁니다(휴~~).

exit 0
```

예 35-3. 배열과 약간의 트릭을 써서 한 벌의 카드를 4명에게 랜덤하게 돌리기

```
#!/bin/bash
# 오래된 시스템이라면 #!/bin/bash2 로 실행시켜야 할지도 모릅니다.

# 카드:
# 카드 한 벌을 4명에게 무작위로 돌리기.

UNPICKED=0
PICKED=1

DUPE_CARD=99

LOWER_LIMIT=0
UPPER_LIMIT=51
CARDS_IN_SUIT=13
CARDS=52

declare -a Deck
declare -a Suits
declare -a Cards
# 하나짜리 3차원 배열이라면 더 쉽고 더 직관적이었을 겁니다.
# 아마 Bash 다음 버전에서는 다차원 배열을 지원할 겁니다.


initialize_Deck ()
{
i=$LOWER_LIMIT
until [ "$i" -gt $UPPER_LIMIT ]
do
  Deck[i]=$UNPICKED   # 카드 "한 벌"(deck)을 모두 안 고른 상태로 둠.
  let "i += 1"
done
echo
}

initialize_Suits ()
{
Suits[0]=C # 클로버(Clubs)
```

```
Suits[1]=D # 다이아몬드(Diamonds)
Suits[2]=H # 하트(Hearts)
Suits[3]=S # 스페이드(Spades)
}

initialize_Cards ()
{
Cards=(2 3 4 5 6 7 8 9 10 J Q K A)
# 배열을 초기화하는 또 다른 방법.
}

pick_a_card ()
{
card_number=$RANDOM
let "card_number %= $CARDS"
if [ "${Deck[card_number]}" -eq $UNPICKED ]
then
  Deck[card_number]=$PICKED
  return $card_number
else
  return $DUPE_CARD
fi
}

parse_card ()
{
number=$1
let "suit_number = number / CARDS_IN_SUIT"
suit=${Suits[suit_number]}
echo -n "$suit-"
let "card_no = number % CARDS_IN_SUIT"
Card=${Cards[card_no]}
printf %-4s $Card
# 카드를 칸 단위로 깔끔하게 출력.
}

seed_random ()  # 랜덤 넘버 발생기 seed.
{
seed=`eval date +%s`
let "seed %= 32766"
RANDOM=$seed
}

deal_cards ()
{
echo

cards_picked=0
while [ "$cards_picked" -le $UPPER_LIMIT ]
```

```
do
  pick_a_card
  t=$?

  if [ "$t" -ne $DUPE_CARD ]
  then
    parse_card $t

    u=$cards_picked+1
    # 임시로 1 부터 인덱싱하는 형태로 바꿈.
    let "u %= $CARDS_IN_SUIT"
    if [ "$u" -eq 0 ]    # 중첩된 if/then 조건 테스트.
    then
     echo
     echo
    fi
    # 다른 사람.

    let "cards_picked += 1"
  fi
done

echo

return 0
}


# 구조적 프로그래밍:
# 전체 프로그램 로직은 함수로 모듈화 되어 있습니다.

#================
seed_random
initialize_Deck
initialize_Suits
initialize_Cards
deal_cards

exit 0
#================



# 연습문제 1:
# 전체에 대해 완전한 주석을 달아보세요.

# 연습문제 2:
# 한 사람이 받은 카드를 종류순으로 정렬해서 보여주도록 고쳐보세요.
# 필요하다면 편리한 다른 기능을 써도 괜찮습니다.
```

```
# 연습문제 3:
# 스크립트를 간략화하고, 합리적인 로직이 되도록 고쳐보세요.
```

주석

1. Chet Ramey 는 펄에서 쓰이는 연관 배열(associative arrays) 을 다음 Bash 에서 사용할 수 있도록 하겠다고 약속했습니다.

36장. 후기(Endnotes)

36.1. 저자 후기(Author's Note)

제가 왜 Bash 스크립팅 책을 쓰게 됐는지에 대한 약간은 이상한 이야기를 해야겠습니다. 몇 년 전이었던걸로 기억하는데, 쉘 스크립트를 배워야 했는데 이렇게 필요에 의해 배우게 되는 것이 특정 주제에 대해 좋은 책을 읽는 것보다 더 좋은 방법입니다. 사실은 지금 이 문서처럼 스크립트의 모든것을 다루는 튜토리얼이나 레퍼런스 같은, 혹은 최소한 비슷하게 생긴 책을 사려고 마음 먹었는데 불행하게도 그런 책을 찾을 수가 없었고, 내가 원한다면 직접 써야만하는 상황이었습니다. 그래서 여러분이 지금 이 문서를 읽을 수 있게 된겁니다.

이 일로 인해 한 미친 교수에 대한 외전(外典, apocryphal story)이 생각났는데, 그 교수는 도서관이든 책방이든 어디에서든지 아무 책이나 보기만 하면 그 책을 써야 하고, 쓸 수 있고, 게다가 더 잘 쓸 수 있다는 강박관념에 완전히 사로잡혀서 그 즉시 집으로 달려가 그 책과 같은 제목의 책을 썼습니다. 몇 년후 그가 죽었을 때 그의 이름으로 낸 책이 수천권에 달해 아시모프가 봤더라도 부끄러워할 정도였습니다. 어쩌면 그 책들이 아무런 가치가 없었을지도 모릅니다만 그게 정말 중요한 걸까요? 비록 그가 그런 강박관념에 사로잡혀 있었다 할지라도 저는 지금 그의 꿈을 갖고 살고 있기 때문에 그 늙은 노인네를 존경하지 않을 수 없습니다.

36.2. 저자에 대해서

저자는 그저 이 책을 쓰고 싶어 썼지 자격증이나 능력을 자랑하려고 쓴 것이 아닙니다.[1] 이 책은 그가 했던 다른 작업, HOW-2 Meet Women: The Shy Man's Guide to Relationships (http://personal.riverusers.com/~thegrendel/hmw45.zip)과 다소 다릅니다. 그는 또한 Software-Building HOWTO (http://linuxdoc.org/HOWTO/Software-Building-HOWTO.html)의 저자이기도 합니다.

저자는 1995년부터 리눅스를 써 왔고(슬랙웨어 2.2, 커널 1.2.1), 일회용 암호화 유틸리티인 cruft (http://ibiblio.org/pub/Linux/utils/file/cruft-0.2.tar.gz), 대출(mortgage, 抵當) 계산기인 mcalc (http://ibiblio.org/pub/Linux/apps/financial/mcalc-1.6.tar.gz), Scrabble® adjudicator 인 judge (http://ibiblio.org/pub/Linux/games/amusements/judge-1.0.tar.gz), 낱말 게임 목록 패키지인 yawl (http://ibiblio.org/pub/Linux/libs/yawl-0.2.tar.gz)등을 발표했습니다. CDC 3800 에서 FORTRAN IV 를 가지고 처음 프로그래밍을 시작했는데, 요즘은 이것들이 아주 그리워집니다.

그는 외지고 쓸쓸한 곳에서 개를 데리고 부인과 함께 살면서, 사람들이 상처받기 쉬운 것에 대해서 조심스럽게 대합니다.(he cherishes human frailty).

36.3. 이 책을 만드는데 쓴 도구들

36.3.1. 하드웨어

레드헷 7.1이 돌고 있는 IBM Thinkpad, 760XL 랩탑(P166, 램 80 메가)을 썼습니다. 확실히 느리고 케케묵은 키보드를 갖고 있지만 어떤것 못지 않게 훌륭합니다.

36.3.2. 소프트웨어와 프린트웨어

i. Bram Moolenaar가 만든 SGML을 인식하는 강력한 텍스트 에디터 vim (http://www.vim.org).

ii. SGML 문서를 다른 포맷으로 변환하는데 쓰이는 DSSSL 렌더링 엔진인 OpenJade (http://www.netfolder.com/DSSSL/).

iii. Norman Walsh의 DSSSL 스타일시트 (http://nwalsh.com/docbook/dsssl/).

iv. Norman Walsh와 Leonard Muellner가 쓴 **DocBook, The Definitive Guide**(오렐리, ISBN 1-56592-580-7). DocBook SGML 포맷으로 문서를 쓰고 싶어하는 사람을 위한 표준 레퍼런스 문서.

36.4. 크레딧

공동체의 참여가 있었기에 이 프로젝트는 가능했습니다. 여러분들의 도움과 피드백이 없었다면 불가능했을 이 작업을 가능케 해준 여러분에게 진심으로 감사드립니다.

Philippe Martin (mailto:feloy@free.fr)이 이 문서를 DocBook/SGML로 변환해 주었습니다. 조그만 프랑스 회사에서 소프트웨어 개발자로 일하고 있는 그는, 여가 시간에 GNU/Linux 문서들과 소프트웨어를 즐기며, 문학 작품을 읽고, 음악을 연주하고, 친구들과 웃고 떠들면서 마음의 안정을 찾는다고 합니다. 프랑스나 바스크 어디에선가 그를 만날 수 있을 겁니다. 이메일은 feloy@free.fr (mailto:feloy@free.fr)로 보내면 됩니다.

Philippe Martin은 또한, $9 이상의 위치 매개변수도 {중괄호} 표기법으로 표현할 수 있다는 것을 지적해 주었습니다. 예 5-5를 참고하세요.

Stephane Chazelas (mailto:stephane_chazelas@yahoo.fr)는 아주 많은 양의 예제 스크립트, 이 문서에 대한 정정및 추가 정보들을 보내줬습니다. 그는 이 문서의 공헌자 이상인 실질적인 편집 책임자 역할을 맡아 줬습니다. 메르씨 보꾸!

특히 **Patrick Callahan**, **Mike Novak**, **Pal Domokos**가 버그를 잡고, 애매모호한 부분들을 지적해 주고, 좀 더 확실한 설명과 변경 사항들을 제안해 준 것을 고맙게 생각합니다. 이들과 나눈 쉘 스크립팅과 일반적인 문서화 이슈에 대한 활발한 토의 덕분에 이 문서를 좀 더 읽기 좋게 만들 수 있었습니다.

Jim Van Zandt에게는 이 문서 0.2 버전의 에러와 빠진 부분을 지적해 준 것에 대해 깊이 감사합니다. 그는 또한 유익한 스크립트도 제공해 주었습니다.

Jordi Sanfeliu (mailto:mikaku@arrakis.es)가 멋진 트리 스크립트(예 A-12)를 쓰도록 허락해 준 것에 대해 많은 감사를 드립니다.

Noah Friedman (mailto:friedman@prep.ai.mit.edu)에게 그의 문자열 함수 스크립트를 쓰도록 해준 것에 대해 감사의 말을 전합니다.

Emmanuel Rouat는 명령어 치환과 별칭(alias)에 대해서 정정과 추가를 제안해 주었습니다. 그는 또한 아주 멋진 `.bashrc` 예제 파일(부록 F)도 제공해 주었습니다.

Heiner Steven (mailto:heiner.steven@odn.de)은 친절하게도 자신의 진법 변환 스크립트(예 12-29)의 사용을 허가해 주었습니다. 또, 많은 부분에 대해서 정정을 해 줬고 도움이 되는 많은 제안을 해 줬습니다. 정말 고맙습니다.

Florian Wisser는 문자열 테스트 스크립트(예 7-5)의 몇몇 세세한 부분을 비롯해 다른 부분들도 설명해 주었습니다.

Oleg Philon 는 cut 과 pidof 에 대한 제안을 보내 줬습니다.

Marc-Jano Knopp 는 도스 배치 파일을 정정해 주었습니다.

차현진(옮긴이: 저군요 :-)이 이 문서의 한국어 번역을 하면서 몇몇 오타를 발견하고 지적해 준 것에 대해 감사하게 생각합니다.

Gabor Kiss, Leopold Toetsch, Nick Drage가 도움이 되는 많은 제안을 해주고 여러 에러를 지적해 주었습니다(스크립트 아이디어!).

세련되고 강력한 스크립팅 도구인 **Bash**를 만들어준 Chet Ramey (mailto:chet@po.cwru.edu)에게 고마움의 뜻을 표합니다.

무엇보다도, 많은 격려를 해주고 정신적으로 지원을 해준 나의 아내, Anita에게 고맙다는 말을 하고 싶습니다.

주석

1. 그럴수 있는 사람은 그렇게 하고 아닌 사람은 MCSE 를 따세요.

서지사항

DoughertyDale 그리고 RobbinsArnold, *Sed and Awk*, 2nd edition, O'Reilly and Associates, 1997, 1-156592-225-5.

쉘 스크립트의 강력한 모든 기능을 쓰려면 최소한 **sed**와 **awk**에 대해서 잘 알고 있어야 됩니다. 이 책은 표준 튜토리얼입니다. 여기에는 "정규 표현식"이 아주 훌륭하게 소개되어 있습니다. 이 책을 꼭 읽어보기 바랍니다.

*

FrischAeleen, *Essential System Administration*, 2nd edition, O'Reilly and Associates, 1995, 1-56592-127-5.

이 훌륭한 시스템 관리자 매뉴얼은 시스템 관리자를 위한 쉘 스크립팅을 아주 멋지게 소개해 놓고, 시스템 구동(startup)과 초기화 스크립트에 대해서도 아주 잘 설명해 놓고 있습니다. 3 판이 나올 때가 됐는데 아직 안 나오는군요(팀 오렐리, 듣고 있어요?).

*

KochanStephen 그리고 WoodsPatrick, *Unix Shell Programming*, Hayden, 1990, 067248448X.

이 책은 약간 오래되긴 했지만 표준 레퍼런스입니다.

*

MatthewNeil 그리고 StonesRichard, *Beginning Linux Programming*, Wrox Press, 1996, 1874416680.

이 책은 쉘 스크립트에 대해 아주 자세한 설명을 포함해서 리눅스에서 가능한 다양한 프로그래밍 언어들을 깊이 있게 다루고 있습니다.

*

MayerHerbert, *Advanced C Programming on the IBM PC*, Windcrest Books, 1989, 0830693637.

알고리즘과 일반적인 프로그래밍 습관을 아주 훌륭하게 다루고 있습니다.

*

MedinetsDavid, *Unix Shell Programming Tools*, McGraw-Hill, 1999, 0070397333.

쉘 스크립트에 대한 좋은 정보를 예제와 함께 보여 주고 Tcl과 펄에 대해 짧게 소개합니다.

*

NewhamCameron 그리고 RosenblattBill, *Learning the Bash Shell*, 2nd edition, O'Reilly and Associates, 1998, 1-56592-347-2.

셸 입문서로 아주 훌륭하지만 프로그래밍 자체에 대한 언급이 불충분하고 충분한 예제가 부족합니다.

*

OlczakAnatole, *Bourne Shell Quick Reference Guide*, ASP, Inc., 1991, 093573922X.

아주 편한 포켓 레퍼런스입니다만, Bash 만의 특징들이 빠져 있습니다.

*

PeekJerry, O'ReillyTim, 그리고 LoukidesMike, *Unix Power Tools*, 2nd edition, O'Reilly and Associates, Random House, 1997, 1-56592-260-3.

셸 프로그래밍에 대해서 유익하고 자세한 설명의 몇 개의 절을 포함하고 있으나 튜토리얼이 되기에는 좀 모자랍니다. 위에서 소개했던 Dougherty 와 Robbins 의 책에서 설명한 정규 표현식 튜토리얼의 많은 부분을 다시 설명하고 있습니다.

*

RobbinsArnold, *Bash Reference Card*, SSC, 1998, 1-58731-010-5.

아주 훌륭한 Bash 포켓 레퍼런스(외출할 때 꼭 가지고 나가세요). 세일시에 4.95 달러에 살 수 있지만 pdf 포맷으로 온라인 (http://www.ssc.com/ssc/bash/)상에서 공짜로 다운로드 받을 수도 있습니다.

*

RobbinsArnold, *Effective Awk Programming*, Free Software Foundation / O'Reilly and Associates, 2000, 1-882114-26-4.

최고의 **awk** 튜토리얼및 레퍼런스. 이 책의 무료 전자 버전은 **awk** 문서의 일부분이고, 최신 버전의 출판된 책은 O'Reilly and Associates 에서 구할 수 있습니다.

저자는 이 책을 보고 영감을 얻어서 이 문서를 작성했습니다.

*

RosenblattBill, *Learning the Korn Shell*, O'Reilly and Associates, 1993, 1-56592-054-6.

셸 스크립팅에 대해서 훌륭한 지침서 역할을 해 주는 아주 잘 쓰여진 책입니다.

*

SheerPaul, *LINUX: Rute User's Tutorial and Exposition*, 1st edition, , 2002, 0-13-033351-4.

리눅스 시스템 관리자를 위한 아주 자세하고 읽기 좋은 소개서.

책으로도 구할 수 있고 온라인 (http://rute.sourceforge.net/)으로도 구할 수 있습니다.

*

SieverEllen 그리고 and the Staff of O'Reilly and Associates, *Linux in a Nutshell*, 2nd edition, O'Reilly and Associates, 1999, 1-56592-585-8.

Bash 섹션도 포함하고 있는 최고의 만능 리눅스 명령어 레퍼런스.

*

The UNIX CD Bookshelf, 2nd edition, O'Reilly and Associates, 2000, 1-56592-815-6.

시디롬에 **UNIX Power Tools**, **Sed and Awk**, **Learning the Korn Shell**을 포함해서 6개의 유닉스 책이 들어 있습니다. 70달러면 여러분이 필요했던 완벽한 형태의 모든 유닉스 레퍼런스와 튜토리얼을 가질 수 있습니다. 집세를 못 내고 빚을 져야 한다 하더라도 이 책은 꼭 사기 바랍니다.

불행하게도 현재 절판됐습니다.

*

오렐리의 펄 책들(사실, 오렐리 책이면 아무거나).

리눅스 가제트 (http://www.linuxgazette.com) 53, 54, 55, 57, 59호에 Ben Okopnik가 잘 써놓은 **introductory Bash scripting** 기사. 56호의 "The Deep, Dark Secrets of Bash" 설명.

리눅스 저널 (http://www.linuxjournal.com) 3, 4호(1994년 7, 8월)에 걸쳐 출판된 Chet Ramey의 **bash - The GNU Shell**.

Mike G 의 Bash-Programming-Intro HOWTO (http://www.linuxdoc.org/HOWTO/Bash-Prog-Intro-HOWTO.html).

Richard 의 UNIX Scripting Universe (http://www.injunea.demon.co.uk/index.htm).

Chet Ramey 의 Bash F.A.Q. (ftp://ftp.cwru.edu/pub/bash/FAQ)

Lucc's Shell Scripts (http://alge.anart.no/linux/scripts/)에 있는 예제 쉘 스크립트들.

SHELLdorado (http://www.oase-shareware.org/shell/scripts) 에 있는 예제 쉘 스크립트들.

Noah Friedman's script site (http://clri6f.gsi.de/gnu/bash-2.01/examples/scripts.noah/) 에 있는 예제 쉘 스크립트들.

SourceForge Snippet Library - shell scrips (http://sourceforge.net/snippet/browse.php?by=lang&lang=7) 에 있는 예제 쉘 스크립트들.

Giles Orr 의 Bash-Prompt HOWTO (http://www.linuxdoc.org/HOWTO/Bash-Prompt-HOWTO.html).

The sed F.A.Q. (http://www.cornerstonemag.com/sed/sedfaq.html)

Carlos Duarte 의 유익한 튜토리얼인 "Do It With Sed" (http://www.dbnet.ece.ntua.gr/~george/sed/sedtut_1.html).

GNU **gawk** 레퍼런스 매뉴얼 (http://sunsite.ualberta.ca/Documentation/Gnu/gawk-3.0.6/gawk.html)(**gawk**는 리눅스와 BSD 시스템에서 쓸 수 있는 **awk**의 확장된 GNU 버전입니다).

Trent Fisher 의 groff tutorial (http://www.cs.pdx.edu/~trent/gnu/groff/groff.html).

Mark Komarinski 의 Printing-Usage HOWTO (http://www.linuxdoc.org/HOWTO/Printing-Usage-HOWTO.html).

알버타 대학 사이트 (http://sunsite.ualberta.ca/Documentation)의 chapter 10 of the textutils documentation (http://sunsite.ualberta.ca/Documentation/Gnu/textutils-2.0/html_chapter/textutils_10.html)에서 I/O 재지향에 대한 멋진 내용을 만나 볼 수 있습니다.

Rick Hohensee은 쉘 스크립트만 써서 가상 머신과 어셈블러를 작성했습니다. H3sm site (http://linux01.gwdg.de/~rhohen/H3sm.html) 나 ftp 사이트 (ftp://linux01.gwdg.de/pub/linux/clienux/interim)를 참고하세요.

Chet Ramey와 Brian Fox가 쓴 훌륭한 "Bash 레퍼런스 매뉴얼". "bash-2-doc" 패키지중 하나로 배포됩니다(rpm으로도 가능함). 특히, 유익한 예제 스크립트를 잘 살펴보세요.

comp.os.unix.shell (news:comp.unix.shell) 뉴스그룹.

bash, **bash2**, **date**, **expect**, **expr**, **find**, **grep**, **gzip**, **ln**, **patch**, **tar**, **tr**, **bc**, **xargs** 맨 페이지. **bash**, **dd**, **m4**, **gawk**, **sed** 에 대한 texinfo 문서들.

부록 A. 여러분들이 보내준 스크립트들(Contributed Scripts)

이 스크립트들은 이 문서의 주제에 딱 들어맞지는 않지만 쉘 프로그래밍 테크닉의 재밌는 부분을 보여주고 또한 쓸만합니다. 이 스크립트들을 분석해 보고 실행해 보면서 재미를 느껴 보기 바랍니다.

예 **A-1. manview:** 포맷된 맨 페이지를 보는 스크립트

```
#!/bin/bash
# manview.sh: 맨페이지 소스를 보기 위해 형식화하기.

# 맨페이지 소스를 작성하는 도중에
# 즉시 작업 결과를 보고 싶을 때 쓰면 아주 좋습니다.

E_WRONGARGS=65

if [ -z "$1" ]
then
  echo "사용법: `basename $0` [filename]"
  exit $E_WRONGARGS
fi

groff -Tascii -man $1 | less
# 맨페이지를 groff 로 보기.

# 맨페이지에 테이블이나 방정식이 들어 있다면, 위 코드는 에러가 납니다.
# 이렇게 하면 그런 상황을 처리할 수 있습니다.
#
#   gtbl < "$1" | geqn -Tlatin1 | groff -Tlatin1 -mtty-char -man
#
#   Thanks, S.C.

exit 0
```

예 **A-2. mailformat:** 이메일 메세지를 포맷해서 보기

```
#!/bin/bash
# mail-format.sh: 이메일 메세지를 포맷.

# 캐럿(>)이나, 탭, 과도한 들여쓰기가 된 줄을 삭제.

ARGS=1
E_BADARGS=65
E_NOFILE=66
```

```
if [ $# -ne $ARGS ]  # 원하는 숫자의 인자가 넘어왔는지.
then
  echo "사용법: `basename $0` filename"
  exit $E_BADARGS
fi

if [ -f "$1" ]       # 파일이 존재하는지 확인.
then
    file_name=$1
else
    echo "\"$1\" 이란 파일은 없습니다."
    exit $E_NOFILE
fi

MAXWIDTH=70          # 긴 줄을 접을 넓이.

sed '
s/^>//
s/^  *>//
s/^  *//
s/ *//
' $1 | fold -s --width=$MAXWIDTH
          # "fold"의 -s 옵션은 가능하다면 공백문자에서 줄을 잘라줍니다.

#  이 스크립트는 아주 유명한 잡지에 기사로 소개되어 격찬을 받은
#+ 164K 짜리 윈도우즈용 유틸리티에서 영감을 얻어 작성되었습니다.

exit 0
```

예 A-3. rn: 간단한 파일이름 변경 유틸리티

이 스크립트는 예 12-15의 변경판입니다.

```
#! /bin/bash
#
# "lowercase.sh"에 기초해 작성된 아주 간단한 파일이름 "rename" 유틸리티.
#
#  Vladimir Lanin(lanin@csd2.nyu.edu)이 만든 "ren" 유틸리티가
#+ 이것보다 훨씬 더 좋습니다.


ARGS=2
E_BADARGS=65
ONE=1                    # 단수, 복수를 문법에 맞게 쓰려고(아래를 보세요).

if [ $# -ne "$ARGS" ]
then
  echo "사용법: `basename $0` old-pattern new-pattern"
  #  예를 들어 "rn gif jpg" 라고 하면 현재 디렉토리의 모든 gif 파일의 이름을
```

```
  #+ jpg 파일로 바꿔줍니다.
  exit $E_BADARGS
fi

number=0                  # 실제로 얼마나 많은 파일이름이 바뀌었는지를 담고 있
을 변수.


for filename in *$1*      # 디렉토리에서 일치하는 모든 파일을 탐색.
do
   if [ -f "$filename" ]  # 찾았다면...
   then
     fname=`basename $filename`            # 경로를 떼어내고,
     n=`echo $fname | sed -e "s/$1/$2/"`   # 새 이름으로 바꾼 다음,
     mv $fname $n                          # 이름 바꿈.
     let "number += 1"
   fi
done

if [ "$number" -eq "$ONE" ]                # 문법에 맞게 하려고
then
 echo "$number 개의 파일이름이 바뀌었습니다."
else
 echo "$number 개의 파일이름들이 바뀌었습니다."
fi

exit 0


# 독자들을 위한 연습문제:
# 어떤 종류의 파일일 경우에 이 스크립트가 제대로 동작하지 않을까요?
# 그런 파일을 처리하려면 어떻게 고치죠?
```

예 A-4. encryptedpw: 로컬에 암호화 되어 있는 비밀번호로 **ftp** 사이트에 파일을 업로드하는 스크립트

```
#!/bin/bash

# "ex72.sh" 예제를 암호화된 비밀번호를 사용하도록 수정함.

#  하지만 복호화된 비밀번호가 그대로 전송되기 때문에
#+ 아직도 안전하지는 않습니다. 주의하세요.
# 이 점이 걱정된다면 "ssh" 같은 것을 써보세요.

E_BADARGS=65

if [ -z "$1" ]
then
```

```
  echo "사용법: `basename $0` filename"
  exit $E_BADARGS
fi

Username=bozo           # 알맞게 고치세요.
pword=/home/bozo/secret/password_encrypted.file
# 암호화된 비밀번호가 들어있는 파일.

Filename=`basename $1`  # 파일이름에서 경로이름을 떼어 냅니다.

Server="XXX"
Directory="YYY"         # 실제 서버이름과 디렉토리로 바꾸세요.


Password=`cruft <$pword`          # 비밀번호 복호화.
#  저자가 만든 고전적인 1회용 암호표(onetime pad) 알고리즘에 기반한
#+ "cruft" 파일 암호화 패키지를 씁니다.
#+ "cruft"는 다음에서 얻을 수 있습니다.
#+ Primary-site:   ftp://metalab.unc.edu /pub/Linux/utils/file
#+                 cruft-0.2.tar.gz [16k]


ftp -n $Server <<End-Of-Session
user $Username $Password
binary
bell
cd $Directory
put $Filename
bye
End-Of-Session
# "ftp"의 -n 옵션은 자동 로그인을 막아줍니다.
# "bell"은 파일 전송이 일어날 때마다 '벨'을 울려줍니다.

exit 0
```

예 A-5. copy-cd: 데이타 **CD**를 복사하는 스크립트

```
#!/bin/bash
# copy-cd.sh: 데이타 CD 복사

CDROM=/dev/cdrom                          # CDROM 디바이스
OF=/home/bozo/projects/cdimage.iso        # 출력 파일
#       /xxxx/xxxxxxx/                    여러분 시스템에 맞게 고치세요.
BLOCKSIZE=2048
SPEED=2                                   # 더 빠른 CDROM 이면 거기에 맞는 속
도를 쓰세요.

echo; echo "원본 CD 를 넣고, 마운트는 하지 \"마세요.\""
```

```
echo "준비가 되면 엔터를 누르세요. "
read ready                                # 입력 대기, $ready 는 안 쓰임.

echo; echo "원본 CD 를 $OF 로 복사합니다."
echo "시간이 걸릴 수도 있으니 기다리기 바랍니다."

dd if=$CDROM of=$OF bs=$BLOCKSIZE         # 디바이스를 물리적 그대로(raw) 복
사.


echo; echo "데이타 CD 를 꺼내세요."
echo "공 CD 를 넣으세요."
echo "준비가 되면 엔터를 누르세요. "
read ready                                # 입력 대기, $ready 는 안 쓰임.

echo "$OF 를 CDR 로 복사합니다."

cdrecord -v -isosize speed=$SPEED dev=0,0 $OF
# Joerg Schilling 의 "cdrecord" 패키지 씀(해당 문서 참고).
# http://www.fokus.gmd.de/nthp/employees/schilling/cdrecord.html


echo; echo "$OF 를 $CDROM 디바이스로 복사 완료했습니다."

echo "이미지 파일을 지우고 싶으세요(y/n)? "  # 아마 꽤 큰 파일이겠죠.
read answer

case "$answer" in
[yY]) rm -f $OF
      echo "$OF 를 지웠습니다."
      ;;
*)    echo "$OF 를 안 지웠습니다.";;
esac

echo

# 독자들을 위한 연습문제:
# 위의 "case" 문을 "yes"와 "Yes"도 받아들이도록 고쳐보세요.

exit 0
```

예 **A-6. days-between:** 두 날짜 사이의 차이를 계산해 주는 스크립트

```
#!/bin/bash
# days-between.sh:    두 날짜 사이의 날 수.
# 사용법: ./days-between.sh YYYY/[M]M/[D]D YYYY/[M]M/[D]D

ARGS=2                # 명령어줄 매개변수는 두 개 필요.
```

```
E_PARAM_ERR=65        # 매개변수 에러.

REFYR=1600            # 참조 년(Reference year).
CENTURY=100
DIY=365
ADJ_DIY=367           # 윤년 + fraction 을 조절하기 위해서.
MIY=12
DIM=31
LEAPCYCLE=4

MAXRETVAL=256         # 함수에서 리턴 가능한 최대 정수

diff=       # 날짜 간격을 위한 전역 변수 선언.
value=                # 절대값을 위한 전역 변수 선언.
day=                  # 년, 월, 일을 위한 전역 변수 선언.
month=
year=


Param_Error ()        # 틀린 명령어줄 매개변수.
{
  echo "사용법: `basename $0` YYYY/[M]M/[D]D YYYY/[M]M/[D]D"
  echo "       (1600/1/3 이후의 날짜여야 합니다)"
  exit $E_PARAM_ERR
}


Parse_Date ()                 # 명령어줄 매개변수로 넘어온 날짜를 파싱.
{
#  month=${1%%/**}
#  dm=${1%/**}                 # Day and month.
#  day=${dm#*/}
# 옮긴이: YYYY/MM/DD 형태로 바꿈.
  year=${1%%/**}
  ym=${1%/**}                 # 년, 월.
  month=${ym#*/}
  let "day = `basename $1`"   # 파일이름은 아니지만 똑같이 동작합니다.
}


check_date ()                 # 날짜 범위가 맞는지 확인.
{
  [ "$day" -gt "$DIM" ] || [ "$month" -gt "$MIY" ] || [ "$year" -
lt "$REFYR" ] && Param_Error
  # 제대로 된 날짜가 아닐 경우에 스크립트를 종료.
  # "or-list / and-list" 를 썼죠.
  # 독자들을 위한 연습문제: 날짜를 좀 더 정확하게 확인하도록 구현해 보세요.
}
```

```
strip_leading_zero () # 월이나 일 앞에 0 이 있을 경우에는
{                     # Bash 가 8 진수로 해석하기 때문에(POSIX.2, 2.9.2.1 절)
  val=${1#0}          # 그 0 을 잘라낼 필요가 있습니다.
  return $val
}


day_index ()          # 가우스 공식(Gauss' Formula):
{                     # 1600년 1월 3일부터 매개변수로 넘어온 날짜까지 계산.

  day=$1
  month=$2
  year=$3

  let "month = $month - 2"
  if [ "$month" -le 0 ]
  then
    let "month += 12"
    let "year -= 1"
  fi

  let "year -= $REFYR"
  let "indexyr = $year / $CENTURY"


  let "Days = $DIY*$year + $year/$LEAPCYCLE -
$indexyr + $indexyr/$LEAPCYCLE + $ADJ_DIY*$month/$MIY + $day - $DIM"
  # 이 알고리즘에 대한 자세한 설명은 다음을 참고하세요.
  # http://home.t-online.de/home/berndt.schwerdtfeger/cal.htm


  if [ "$Days" -gt "$MAXRETVAL" ]  # 함수에서 256 보다 큰 수를 리턴하기 위해서
  then                             # 결과가 256 보다 크다면 음수로 바꿈.
    let "dindex = 0 - $Days"
  else let "dindex = $Days"
  fi

  return $dindex

}


calculate_difference ()            # 두 날짜간 차이나는 날 수를 계산.
{
  let "diff = $1 - $2"             # 전역 변수.
}
```

```
abs ()                                # 절대값
{                                     # "value" 전역 변수 사용.
  if [ "$1" -lt 0 ]                   # If 음수
  then                                # then
    let "value = 0 - $1"              # 부호 바꾸기,
  else                                # else
    let "value = $1"                  # 그대로 둠.
  fi
}



if [ $# -ne "$ARGS" ]                 # 명령어줄 매개변수가 두 개 필요.
then
  Param_Error
fi

Parse_Date $1
check_date $day $month $year         # 날짜 형식이 맞는지 확인.

strip_leading_zero $day              # 일이나 월 앞에 붙은 0 을 제거.
day=$?
strip_leading_zero $month
month=$?

day_index $day $month $year
date1=$?

abs $date1                            # 절대값을 구해 확실히 양수로 만듦
date1=$value

Parse_Date $2
check_date $day $month $year

strip_leading_zero $day
day=$?
strip_leading_zero $month
month=$?

day_index $day $month $year
date2=$?

abs $date2                            # 양수로 만듦.
date2=$value

calculate_difference $date1 $date2

abs $diff                             # 양수로 만듦.
diff=$value
```

```
echo $diff

exit 0
# http://buschencrew.hypermart.net/software/datedif
# 에 있는 가우스 공식의 C 구현과 이 스크립트를 비교해 보세요.
```

+

다음 두 스크립트는 토론토 대학의 Mark Moraes가 작성했습니다. 허가 사항과 제한 사항에 대해서는 이 문서와 같이 배포되는 “Moraes-COPYRIGHT” 파일을 참고하기 바랍니다.

예 **A-7. behead:** 메일과 뉴스 메세지 헤더를 제거해 주는 스크립트

```
#! /bin/sh
# 메일이나 뉴스 메세지에서 첫번째 빈 줄이 나올 때까지의
# 헤더를 떼어 내 주는 스크립트.
# 토론토 대학의 Mark Moraes 작성

# --> 이 주석은 HOWTO 저자가 붙인 것입니다.

if [ $# -eq 0 ]; then
# --> 명령어 줄 인자가 없다면 표준입력에서 재지향되어 들어오는 파일에 대해서 동작함.
sed -e ’1,/^$/d’ -e ’/^[  ]*$/d’
# --> 공백 문자로 시작하는 첫번째 줄이 나올 때까지
# --> 빈 줄과 모든 줄을 지웁니다.
else
# --> 명령어 줄 인자가 있다면 그 주어진 파일에 대해서 동작함
for i do
sed -e ’1,/^$/d’ -e ’/^[  ]*$/d’ $i
# --> 위와 같습니다.
done
fi

# --> 독자를 위한 연습문제: 에러 체크와 다른 옵션을 추가해 보세요.
# -->
# --> 작은 sed 스크립트가 인자를 넘기는 것만 빼고 반복되는것을 유심히 살펴 보세요.
# --> 이 부분을 함수로 빼는게 이치에 맞을까요? 그 대답에 대해서 설명해 보세요.
```

예 **A-8. ftpget: ftp**에서 파일을 다운로드 해 주는 스크립트

```
#! /bin/sh
# $Id: ftpget,v 1.2 91/05/07 21:15:43 moraes Exp $
# 익명 ftp에 배치 작업을 실행하는 스크립트. 기본적으로 명령어 라인 인자들을
# ftp의 입력으로 바꿔주는 일을 합니다.
# 간단하고 빠릅니다 - ftplist와 한 쌍이 되도록 작성됐습니다.
# -h 는 접속할 호스트를 나타냅니다(기본값은 prep.ai.mit.edu)
# -d 는 접속후 cd 로 옮겨갈 디렉토리를 나타냅니다. -d 를 여러번 쓸 수도
```

```
# 있는데, 이렇게 하면 주어진 순서대로 디렉토리를 옮겨갈 것입니다.
# 만약에 해당 디렉토리가 상대 경로라면 순서를 잘 매겨야 합니다.
# 요즘엔 너무 많은 심볼릭 링크가 존재하기 때문에 아주 조심해서 사용해야 합니다.
# (기본값은 ftp 로그인 디렉토리)
# -v 는 ftp의 verbose모드를 켜서, ftp 서버의 모든 응답을 보여줍니다.
# -f remotefile[:localfile] 은 remote 파일을 local 파일로 이름을 바꿔
# 가져옵니다.
# -m pattern 은 주어진 패턴에 해당하는 파일들을 mget으로 가져옵니다.
# 쉘 문자들을 인용(quote)해야 하는 것을 기억하세요.
# -c 는 현재 자신의 시스템에서 주어진 디렉토리로 cd 를 실행합니다.
# 예를 들면,
#  ftpget -h expo.lcs.mit.edu -d contrib -f xplaces.shar:xplaces.sh \
# -d ../pub/R3/fixes -c ~/fixes -m ’fix*’
# 는 expo.lcs.mit.edu 의 ~ftp/contrib 에서 xplaces.shar 를 현재 디렉토리에
# xplaces.sh 로 가져오고, ~ftp/pub/R3/fixes 에서 모든 수정 파일(’fix*’)들을
# 자기 시스템의 ~/fixes 디렉토리로 가져옵니다.
# ftp 에서 해당 명령어가 주어진 순서대로 실행되기 때문에 옵션 순서가
# 중요하다는 것은 아주 확실합니다.
#
# Mark Moraes (moraes@csri.toronto.edu), Feb 1, 1989
# ==> Docbook에서 처리할 수 있도록 부등호 괄호를 소괄호로 바꾸었습니다.
#

# ==> 이런 주석은 HOWTO 저자가 덧붙인 주석입니다.

# PATH=/local/bin:/usr/ucb:/usr/bin:/bin
# export PATH
# ==> 원래 스크립트에 있던 위 두 줄은 쓸데없어 보입니다.

TMPFILE=/tmp/ftp.$$
# ==> 스크립트의 프로세스 ID($$)로 임시 파일을 만듦.

SITE=`domainname`.toronto.edu
# ==> ’domainname’은 ’hostname’과 비슷합니다.
# ==> 좀 더 일반적으로 쓰려면 매개변수로 처리하도록 재작성 할 수도 있습니다.

usage="사용
법: $0 [-h remotehost] [-d remotedirectory]... [-f remfile:localfile]... \
[-c localdirectory] [-m filepattern] [-v]"
ftpflags="-i -n"
verbflag=
set -f  # -m 옵션에서 globbing을 쓰기 위해서
set x `getopt vh:d:c:m:f: $*`
if [ $? != 0 ]; then
echo $usage
exit 65
fi
shift
trap ’rm -f ${TMPFILE} ; exit’ 0 1 2 3 15
```

```
echo "user anonymous ${USER-gnu}@${SITE} > ${TMPFILE}"
# ==> 쿼우트 추가(복잡한 echo 문에서는 이렇게 하기 바랍니다).
echo binary >> ${TMPFILE}
for i in $*   # ==> 명령어줄 인자를 파싱.
do
case $i in
-v) verbflag=-v; echo hash >> ${TMPFILE}; shift;;
-h) remhost=$2; shift 2;;
-d) echo cd $2 >> ${TMPFILE};
    if [ x${verbflag} != x ]; then
        echo pwd >> ${TMPFILE};
    fi;
    shift 2;;
-c) echo lcd $2 >> ${TMPFILE}; shift 2;;
-m) echo mget "$2" >> ${TMPFILE}; shift 2;;
-f) f1=`expr "$2" : "\([^:]*\).*"`; f2=`expr "$2" : "[^:]*:\(.*\)"`;
    echo get ${f1} ${f2} >> ${TMPFILE}; shift 2;;
--) shift; break;;
esac
done
if [ $# -ne 0 ]; then
echo $usage
exit 65   # ==> 표준을 따르기 위해 "exit 2"였던 것을 수정.
fi
if [ x${verbflag} != x ]; then
ftpflags="${ftpflags} -v"
fi
if [ x${remhost} = x ]; then
remhost=prep.ai.mit.edu
# ==> 여러분이 좋아하는 ftp 싸이트로 바꾸세요.
fi
echo quit >> ${TMPFILE}
# ==> 모든 명령어는 임시 파일로 저장됩니다.

ftp ${ftpflags} ${remhost} < ${TMPFILE}
# ==> 이제 임시 파일에 저장됐던 명령어들이 ftp 에 의해 한 번에 처리됩니다.

rm -f ${TMPFILE}
# ==> 끝으로, 임시 파일 삭제(지우지 않고 로그 파일로 복사할 수도 있습니다).


# ==> 독자들을 위한 연습문제:
# ==> 1) 에러 체크를 추가하세요.
# ==> 2) 다른 편리한 기능들(bells & whistles)을 넣어보세요.
```

+

다음 스크립트는 Antek Sawicki가 보내주었는데 9.3절에서 논의된 매개변수 치환 연산자의 깔끔한 사용법을 보여줍니다.

예 A-9. password: 8 글자짜리 랜덤한 비밀번호 생성 스크립트

```
#!/bin/bash
# 오래된 시스템에서는 #!/bin/bash2 라고 바꿔야 될지도 모릅니다.
#
# 이 Bash 2.x 용 무작위 비밀번호 생성기는 Antek Sawicki <tenox@tenox.tc>
# 의 관대한 허락하에 싣습니다.
#
# ==> 본 문서의 저자가 추가한 주석 ==>


MATRIX="0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
LENGTH="8"
# ==> 더 긴 비밀번호를 원한다면 'LENGTH'를 늘리세요.


while [ "${n:=1}" -le "$LENGTH" ]
# ==> := 가 "기본값 치환"(default substitution) 연산자였던거 기억나시죠?
# ==> 그래서 만약에 'n'이 초기화 되지 않았다면 1로 세트됩니다.
do
PASS="$PASS${MATRIX:$(($RANDOM%${#MATRIX})):1}"
# ==> 아주 훌륭한데 약간 헷갈리네요.

# ==> 제일 안쪽부터 살펴 봅시다...
# ==> ${#MATRIX} 는 MATRIX 배열의 길이를 리턴합니다.

# ==> $RANDOM%${#MATRIX} 은 1 부터 MATRIX의 길이 - 1 에서
# ==> 무작위 숫자를 리턴합니다.

# ==> ${MATRIX:$(($RANDOM%${#MATRIX})):1}
# ==> 은 MATRIX 에서 그 무작위 숫자의 위치에 있는
# ==> 길이 1 짜리 문자열로 확장된 값을 리턴합니다.
# ==> 3.3.1 절의 {var:pos:len} 매개변수 치환과 해당 예제들을 참고하세요.

# ==> PASS=... 이 결과를 바로 전 PASS 에 붙입니다(연결).

# ==> PASS 가 만들어지는 과정을 좀 더 확실하게 보고 싶다면,
# ==> 다음 줄의 주석을 푸세요.
# ==>             echo "$PASS"
# ==> 루프를 한 번 돌때마다 PASS 가 한 글자씩 만들어 집니다.

let n+=1
# ==> 다음 단계를 위해 'n'을 증가.
done

echo "$PASS"      # ==> 필요하다면 파일로 재지향.

exit 0
```

+

James R. Van Zandt 가 보내준 다음 스크립트는 네임드 파이프를 사용하는데 그의 말에 따르면, "정말로 쿼우팅과 이스케이핑" 연습용이었답니다.

예 **A-10. fifo:** 네임드 파이프를 써서 매일 백업해 주는 스크립트

```
#!/bin/bash
# ==> James R. Van Zandt 가 작성한 스크립트로 그의 허락하에 이곳에 싣습니다.

# ==> 이 문서의 저자가 추가한 주석.


  HERE=`uname -n`    # ==> 호스트이름
  THERE=bilbo
  echo "`date +%r`에 $THERE 로 백업을 시작합니다"
  # ==> `date +%r` 은 시간을 12 시간 포맷("08:08:34 PM")으로 리턴합니다.

  # /pipe 가 보통 파일이 아니고 진짜 파이프인지 확인.
  rm -rf /pipe
  mkfifo /pipe       # ==> "/pipe"란 "네임드 파이프"(named pipd) 만들기.

  # ==> 'su xyz' 는 명령어를 "xyz"란 사용자로 실행시킵니다.
  # ==> 'ssh' 는 secure shell을 띄웁니다(원격지 로그인 클라이언트).
  su xyz -c "ssh $THERE \"cat >/home/xyz/backup/${HERE}-
daily.tar.gz\" < /pipe"&
  cd /
  tar -czf -
bin boot dev etc home info lib man root sbin share usr var >/pipe
  # ==> 네임드 파이프(/pipe)를 써서 프로세스끼리 통신을 함
  # ==> 'tar/gzip' 은 /pipe로 쓰고, 'ssh'은 /pipe에서 읽음

  # ==> 최종 결과는 / 이하의 메인 디렉토리들이 백업 됩니다.

  # ==> 지금 같은 상황에서 "익명의 파이프"인 | 대신 그 반대인
  # ==> "네임드 파이프"를 쓴 것이 어떤 이점이 있을까요?
  # ==> 익명 파이프도 제대로 동작할까요?


  exit 0
```

+

다음은 Stephane Chazelas가 보내준, 배열없이 소수(prime number)를 생성해 주는 스크립트입니다.

예 **A-11.** 나머지 연산자로 소수 생성하기

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

```
# primes.sh: 배열을 쓰지 않고 소수(prime number)를 생성.

#  이 스크립트는 고전적인 "에라토스테네스의 체" 알고리즘을 쓰지 *않고*
#+ "%" 나머지 연산자를 써서 소수를 구하는 좀 더 직관적인 방법을 사용합니다.
#
# Stephane Chazelas 가 보내준 스크립트.


LIMIT=1000                    # 2 - 1000 의 소수

Primes()
{
 (( n = $1 + 1 ))             # 다음 정수.
 shift                        # 목록의 다음 매개변수.
#  echo "_n=$n i=$i_"

 if (( n == LIMIT ))
 then echo $*
 return
 fi

 for i; do                    # "i"는 $n 의 이전값인 "@"로 세트됨.
#    echo "-n=$n i=$i-"
   (( i * i > n )) && break   # 최적화.
   (( n % i )) && continue    # 나머지 연산자로 소수가 아닌 수를 걸러냄.
   Primes $n $@               # 루프안에서 재귀 호출.
   return
   done

   Primes $n $@ $n            # 루프밖에서 재귀 호출.
                              # 위치 매개변수로 누적(accumulate) 성공.
         # "$@" 는 현재 찾은 소수의 목록입니다.
}

Primes 1

exit 0

# 무슨 일이 일어나는지 알고 싶으면 16줄과 24줄의 주석을 풀어보세요.

# 이 알고리즘과 에라토스테네스의 체 알고리즘(ex68.sh)의 속도를 비교해 보세요.

# 연습문제: 더 빠르게 실행시키기 위해서 재귀 호출을 쓰지 말고 다시 작성해 보세요.
```

+

Jordi Sanfeliu가 이 근사한 **tree** 스크립트를 쓰도록 허락해 주었습니다.

예 **A-12. tree:** 디렉토리 구조를 트리 형태로 보여주는 스크립트

```
#!/bin/sh
#         @(#) tree      1.1  30/11/95       by Jordi Sanfeliu
#                                         email: mikaku@arrakis.es
#
#         Initial version:  1.0  30/11/95
#         Next version   :  1.1  24/02/97   심볼릭 링크도 지원
#         Patch by       :  Ian Kjos, 찾을 수 없는 디렉토리 지원
#                           email: beth13@mail.utexas.edu
#
#         Tree 는 디렉토리 트리구조를 확실히(:-) ) 보여줍니다.
#

# ==> 'Tree' 스크립트는 원자자인 Jordi Sanfeliu 의 허락하에 여기에 싣습니다.
# ==> 이 문서의 저자가 추가한 주석.
# ==> 인자 쿼우팅 추가.


search () {
   for dir in `echo *`
   # ==> `echo *` 는 현재 디렉토리의 모든 파일을 한 줄에 표시해 줌.
   # ==> for dir in *   라고 하는 것과 비슷함.
   # ==> 하지만 "dir in `echo *`" 은 이름에 빈 칸이 들어있는 파일은 처리 못함.
   do
      if [ -d "$dir" ] ; then   # ==> 디렉토리라면(-d)...
         zz=0   # ==> 디렉토리 깊이를 갖고 있을 임시 변수.
         while [ $zz != $deep ]    # 안쪽 루프인지 계속 확인.
         do
            echo -n "|   "    # ==> 수직 연결자를 표시.
                      # ==> 들여쓰기(indent)를 위해 라인피드 없는 두 개의 빈 칸
            zz=`expr $zz + 1` # ==> zz 증가.
         done
         if [ -L "$dir" ] ; then   # ==> 디렉토리가 심볼릭 링크라면...
            echo "+---$dir" `ls -l $dir | sed 's/^.*'$dir' //'`
    # ==> 수평 연결자와 긴 목록 표시(long linting)에서 날짜/시간 부분을
# ==> 제외한 디렉토리 이름을 표시.
         else
            echo "+---$dir"      # ==> 수평 연결자와 디렉토리 이름 표시.
            if cd "$dir" ; then  # ==> 하위 디렉토리로 들어갈 수 있다면...
               deep=`expr $deep + 1`   # ==> 깊이 증가.
               search     # 재귀적으로 호출 ;-)
                  # ==> 자기 자신을 부르는 함수.
               numdirs=`expr $numdirs + 1`   # ==> 디렉토리 숫자를 증가.
            fi
         fi
      fi
   done
   cd ..   # ==> 부모 디렉토리로 올라감.
```

```
   if [ "$deep" ] ; then  # ==> depth = 0 이면(TRUE을 리턴)...
      swfi=1              # ==> 탐색이 끝났음을 알리는 플래그 세트.
   fi
   deep=`expr $deep - 1`  # ==> 깊이를 감소.
}

# - 메인 -
if [ $# = 0 ] ; then
   cd `pwd`    # ==> 인자없이 불리면 현재 디렉토리에서 시작.
else
   cd $1       # ==> 아니면 주어진 디렉토리로 이동.
fi
echo "시작 디렉토리 = `pwd`"
swfi=0      # ==> 탐색 종료 플래그.
deep=0      # ==> 보여줄 깊이.
numdirs=0
zz=0

while [ "$swfi" != 1 ]   # 플래그가 세트 안 된 동안...
do
   search   # ==> 변수를 초기화하고 함수를 부름.
done
echo "전체 디렉토리 수 = $numdirs"

exit 0
# ==> 독자들에게 도전: 이 스크립트가 정확히 어떻게 동작하는지 알아내 보세요.
```

Noah Friedman 의 허락하에 문자열 조작 C 라이브러리 함수 몇 개의 쉘 스크립트 버전을 보여 줍니다.

예 **A-13.** 문자열 함수들**:** **C** 형태의 문자열 함수

```
#!/bin/bash

# string.bash --- string(3) 라이브러리의 bash 에뮬레이션 버전
# 저자: Noah Friedman <friedman@prep.ai.mit.edu>
# ==>     저자의 허락하에 이 문서에 게재함.
# Created: 1992-07-01
# Last modified: 1993-09-29
# Public domain

# Chet Ramey 가 bash 버전 2 문법을 쓰도록 변환

# Commentary:
# Code:

#:docstring strcat:
# Usage: strcat s1 s2
#
```

```
# Strcat 은 s2 를 s1 으로 덧붙임.
#
# 예제:
#    a="foo"
#    b="bar"
#    strcat a b
#    echo $a
#    => foobar
#
#:end docstring:

###;;;autoload   ==> Autoloading of function commented out.
function strcat ()
{
    local s1_val s2_val

    s1_val=${!1}                        # 변수 간접참조 확장
    s2_val=${!2}
    eval "$1"=\'"${s1_val}${s2_val}"\'
    # ==> 두 변수중 하나라도 작은 따옴표가 들어 있을 경우에는,
    # ==> eval $1='${s1_val}${s2_val}' 이라고 하면 됩니다.
}

#:docstring strncat:
# 사용법: strncat s1 s2 $n
#
# strcat 과 비슷하지만 s2 에서 최대 n 문자만큼만 덧붙입니다. s2 가 n 보다
# 짧다면 그냥 s2 만 복사합니다. Echoes result on stdout.
#
# 예제:
#    a=foo
#    b=barbaz
#    strncat a b 3
#    echo $a
#    => foobar
#
#:end docstring:

###;;;autoload
function strncat ()
{
    local s1="$1"
    local s2="$2"
    local -i n="$3"
    local s1_val s2_val

    s1_val=${!s1}                       # ==> 변수 간접 참조 확장
    s2_val=${!s2}
```

```
    if [ ${#s2_val} -gt ${n} ]; then
       s2_val=${s2_val:0:$n}            # ==> 문자열조각(substring) 추출
    fi

    eval "$s1"=\'"${s1_val}${s2_val}"\'
    # ==> 두 변수중 하나라도 작은 따옴표가 들어 있을 경우에는,
    # ==> eval $1='${s1_val}${s2_val}' 이라고 하면 됩니다.
}

#:docstring strcmp:
# 사용법: strcmp $s1 $s2
#
# Strcmp 는 두 인자를 사전상의 순서대로 비교해서 s1 이 s2 보다 작으면 음수,
# 같으면 0, 크면 양수를 리턴합니다.
#:end docstring:

###;;;autoload
function strcmp ()
{
    [ "$1" = "$2" ] && return 0

    [ "${1}" '<' "${2}" ] > /dev/null && return -1

    return 1
}

#:docstring strncmp:
# 사용법: strncmp $s1 $s2 $n
#
# strcmp 와 비슷하나 최대 n 문자만큼만 비교합니다( n 이 0 이하면 0 을 리턴).
#:end docstring:

###;;;autoload
function strncmp ()
{
    if [ -z "${3}" -o "${3}" -le "0" ]; then
       return 0
    fi

    if [ ${3} -ge ${#1} -a ${3} -ge ${#2} ]; then
       strcmp "$1" "$2"
       return $?
    else
       s1=${1:0:$3}
       s2=${2:0:$3}
       strcmp $s1 $s2
       return $?
    fi
}
```

```
#:docstring strlen:
# 사용법: strlen s
#
# Strlen 은 s 의 길이를 리턴합니다.
#:end docstring:

###;;;autoload
function strlen ()
{
    eval echo "\${#${1}}"
    # ==> 인자로 넘어온 변수값의 길이를 리턴
}

#:docstring strspn:
# 사용법: strspn $s1 $s2
#
# Strspn 은 s2 전체를 포함하는 s1 의 최대 초기 세그먼트 길이를 리턴합니다.
#:end docstring:

###;;;autoload
function strspn ()
{
    # IFS 를 언셋해 놓으면 공백문자도 일반 문자처럼 처리할 수 있습니다.
    local IFS=
    local result="${1%%[!${2}]*}"

    echo ${#result}
}

#:docstring strcspn:
# 사용법: strcspn $s1 $s2
#
# Strcspn 은 s2 전체가 포함되지 않는 s1 의 최대 초기 세그먼트 길이를 리턴합니다.
#:end docstring:

###;;;autoload
function strcspn ()
{
    # IFS 를 언셋해 놓으면 공백문자도 일반 문자처럼 처리할 수 있습니다.
    local IFS=
    local result="${1%%[${2}]*}"

    echo ${#result}
}

#:docstring strstr:
# 사용법: strstr s1 s2
#
```

```
# Strstr 은 s1 에서 s2 가 처음 발견되는 문자열조각(substring)을 에코해 주고,
# s1 에서 s2 가 발견되지 않는다면 그냥 무시합니다.
# 만약에 s2 가 길이가 0 인 문자열을 가르킨다면 s1 을 리턴합니다.
#:end docstring:

###;;;autoload
function strstr ()
{
    # s2 가 가르키는 문자열의 길이가 0 이라면 s1 을 에코
    [ ${#2} -eq 0 ] && { echo "$1" ; return 0; }

    # s1 에서 s2 가 발견되지 않으면 그냥 리턴
    case "$1" in
    *$2*) ;;
    *) return 1;;
    esac

    # 일치하는 부분부터 끝까지를 떼어 내기 위해 패턴 매칭 사용
    first=${1/$2*/}

    # 그 다음에는 일치하지 첫번째 부분을 떼어냄
    echo "${1##$first}"
}

#:docstring strtok:
# 사용법: strtok s1 s2
#
# Strtok 은 s1 이 구분자인 s2 에 들어 있는 하나 이상의 문자열에 의해
# 0 개 이상으로 이루어져 있다고 가정합니다.
# strtok 이 처음 불리면(비어 있지 않은 s1 을 지정해서) 첫번째 토큰을 포함한
# 문자열을 표준출력으로 에코해 줍니다. 이 함수는 첫번째 인자가 빈 문자열로
# 계속 호출되도록 해서 바로 다음에 따라나오는 토큰을 처리하도록 하기 위해서
# 호출시마다 s1 에서의 자신의 위치를 계속 추적합니다.
# 이런 식으로 해서 그 다음 호출은 더 이상의 토큰이 없을 때까지 s1 을 처리합니다.
# 구분 문자열인 s2 는 호출시마다 다를 수 있습니다. s1 에 처리할 토큰이
# 남아 있지 않다면, 빈 값이 표준출력으로 에코됩니다.

###;;;autoload
function strtok ()
{
 :
}

#:docstring strtrunc:
# 사용법: strtrunc $n $s1 {$s2} {$...}
# strncmp 처럼 여러 함수에서 쓰이는데, 비교를 위해서 인자를 잘라냅니다.
# 각각의 s1, s2 ... 에서 처음 n 개의 문자를 표준출력으로 에코해 줍니다.
#:end docstring:
```

```
###;;;autoload
function strtrunc ()
{
    n=$1 ; shift
    for z; do
        echo "${z:0:$n}"
    done
}

# provide string

# string.bash 는 여기까지


# ========================================================================== #
# ==> 여기부터는 본 문서의 저자가 추가한 부분입니다.

# ==> 이 스크립트를 여러분 스크립트에서 쓰려면 여기부터 끝까지 지운다음
# ==> 여러분 스크립트에서 이 스크립트를 "source" 하면됩니다.

# strcat
string0=one
string1=two
echo
echo "\"strcat\" 함수 테스트:"
echo "원래의 \"string0\" = $string0"
echo "\"string1\" = $string1"
strcat string0 string1
echo "새 \"string0\" = $string0"
echo

# strlen
echo
echo "\"strlen\" 함수 테스트:"
str=123456789
echo "\"str\" = $str"
echo -n "\"str\" 의 길이 = "
strlen str
echo



# 독자들을 위한 연습문제:
# 여기서 소개한 문자열 함수를 모두 테스트하는 코드를 작성해 보세요.


exit 0
```

Stephane Chazelas 는 Bash 스크립트의 객체 지향적 구현을 보여줬습니다.

예 **A-14.** 객체 지향 데이타 베이스

```
#!/bin/bash
# obj-oriented.sh: 쉘 스크립트에서 객체 지향적 프로그래밍 하기.
# Stephane Chazelas 작성.


person.new()        # C++ 의 클래스 선언처럼 보입니다.
{
  local obj_name=$1 name=$2 firstname=$3 birthdate=$4

  eval "$obj_name.set_name() {
          eval \"$obj_name.get_name() {
                   echo \$1
                 }\"
        }"

  eval "$obj_name.set_firstname() {
          eval \"$obj_name.get_firstname() {
                   echo \$1
                 }\"
        }"

  eval "$obj_name.set_birthdate() {
          eval \"$obj_name.get_birthdate() {
            echo \$1
          }\"
          eval \"$obj_name.show_birthdate() {
            echo \$(date -d \"1/1/1970 0:0:\$1 GMT\")
          }\"
          eval \"$obj_name.get_age() {
            echo \$(( (\$(date +%s) - \$1) / 3600 / 24 / 365 ))
          }\"
        }"

  $obj_name.set_name $name
  $obj_name.set_firstname $firstname
  $obj_name.set_birthdate $birthdate
}

echo

person.new self Bozeman Bozo 101272413
# "person.new" 인스턴스 생성(실제로는 인자를 함수로 넘기는 것임).

self.get_firstname       #   Bozo
self.get_name            #   Bozeman
self.get_age             #   28
self.get_birthdate       #   101272413
```

```
self.show_birthdate      #   Sat Mar 17 20:13:33 MST 1973

echo

# 생성된 함수를 보려면
# typeset -f
# 라고 해 보세요(화면이 주루룩 올라가니까 조심하세요).

exit 0
```

부록 B. Sed 와 Awk 에 대한 간단한 입문서

텍스트 처리 유틸리티인 **sed**와 **awk**에 대해서 아주 간단한 소개를 해드리겠습니다. 여기서는 몇 개의 기본적인 명령어만 살펴보는데, 쉘 스크립트에서 쓰이는 sed와 awk의 동작을 이해하는데는 부족함이 없을 것입니다.

sed: 비대화형(non-interactive) 모드의 텍스트 파일 에디터

awk: C 형태의 문법을 갖는 필드 단위의 패턴(field-oriented pattern) 처리 언어

이런 차이점에도 불구하고 이 둘은 비슷한 실행 문법을 갖는데, 정규 표현식을 쓰고, 기본 입출력은 표준입력과 표준출력을 쓴다는 것입니다. 이렇게 잘 정의된 유닉스식 특징 때문에 이 둘은 같이 잘 동작합니다. 파이프를 통해서 한 쪽의 출력을 다른 쪽으로 넘길 수 있기 때문에 이들의 이런 능력을 잘 엮어서 쓰게 되면 스크립트를 펄의 능력과 거의 비슷한 수준으로 끌어올릴 수 있습니다.

> 참고**:** 한가지 중요한 차이점은 쉘 스크립트에서 sed로 인자를 전달하는 것은 쉬운데 비해 awk는 좀 더 복잡하다는 것입니다(예 34-3와 예 9-18 참고).

B.1. Sed

sed는 비대화형(non-interactive) 모드의 줄 단위 편집기입니다. 표준입력이나 파일에서 텍스트를 입력으로 받아, 주어진 줄들에 대해 한 번에 한 줄씩 어떤 처리를 한 다음 그 결과를 표준출력이나 파일로 내 보냅니다. sed는 쉘 스크립트에서 파이프에 걸어 쓸 수 있는 몇몇 도구중 하나입니다.

sed는 주어진 주소 범위(address range)에서 의해 입력의 어떤 줄을 조작할 것인지를 결정합니다. [1] 이 주소 범위는 줄 번호로 주어질 수도 있고 일치하는 패턴으로 주어질 수도 있습니다. 예를 들면, *3d*는 입력의 3번째 줄을 지우라는 신호이고 */windows/d*는 “windows”를 포함하는 모든 줄을 지우라는 뜻입니다.

sed가 할 수 있는 일 가운데, 일반적으로 가장 많이 쓰이는 **p**rinting(표준출력으로 출력), **d**eletion(삭제), **s**ubstitution(치환), 이 세 가지를 살펴보겠습니다.

표 **B-1.** 기본 **sed** 연산자

연산자	이름	뜻
`[주소-범위]/p`	print	[주어진 주소 범위] 출력
`[주소-범위]/d`	delete	[주어진 주소 범위] 삭제
`s/pattern1/pattern2/`	substitute	한 줄에서 처음 나타나는 pattern1을 pattern2로 치환

연산자	이름	뜻
`[주소-범위]/s/pattern1/pattern2/`	substitute	*주소-범위에 대해서 한 줄에 처음 나타나는 patter1을 pattern2로 치환*
`[주소-범위]/y/pattern1/pattern2/`	transform	*주소-범위에 대해서 pattern1에 나타나는 어떤 문자라도 pattern2에 나타나는 문자로 바꿈(**tr**과 동일)*
`g`	global	입력의 일치하는 각 줄에서 발생하는 모든 패턴에 대해 동작

참고**:** 치환(substitute) 명령어에 `g`(**global**) 연산자를 같이 안 쓰면 한 줄에 패턴이 여러번 일치하더라도 오직 첫 번째 패턴만 치환됩니다.

명령어 줄이나 쉘 스크립트에서 sed를 쓸 때는 쿼우팅과 몇몇 옵션이 필요할 지도 모릅니다.

```
sed -e '/^$/d'
# -e 옵션은 다음에 나오는 문자열을 편집 명령어로 해석하게 합니다.
# ("sed"에 단 하나의 명령을 내린다면 "-e"는 넣어도 되고 안 넣어도 됩니다.)
# "강한" 쿼우트(')는 sed 명령에 나오는 정규 표현식용 문자가
# 쉘에 의해 특수 문자로 재해석 되는 것을 막아줍니다.
# (이렇게 해서 sed가 명령어의 정규 표현식을 확장하도록 해 줍니다.)
```

참고**:** sed, awk 모두, `-e` 옵션을 써서 다음에 나오는 문자열이 명령이나 명령 집합이라는 것을 지정해 줍니다. 만약에 문자열이 그냥 하나짜리 명령이라면, 이 옵션은 안 적어줘도 됩니다.

```
sed -n '/xzy/p'
# -n 옵션은 패턴이 일치하는 줄만 출력하도록 합니다.
# 이 옵션이 없으면 모든 입력이 출력됩니다.
# 여기서는 단지 한 개의 편집 명령만 쓰기 때문에 -e 옵션은 필요없습니다.
```

표 **B-2.** 예제

표시	뜻
`8d`	입력의 8번째 줄을 지워라.
`/^$/d`	빈 줄을 모두 지워라.
`1,/^$/d`	첫 줄부터 처음 나타나는 빈 줄까지 지워라.

표시	뜻
`/Jones/p`	“Jones”를 포함하는 줄만 출력하라(-n 옵션을 써서).
`s/Windows/Linux/`	입력의 각 줄에서 처음 나오는 “Windows”를 “Linux”로 치환하라.
`s/BSOD/stability/g`	입력의 각 줄에서 “BSOD”가 나올 때 마다 “stability”로 치환하라.
`s/ *$//`	모든 줄의 끝에 나오는 빈 칸을 지워라.
`s/00*/0/g`	연속적인 모든 0을 하나의 0으로 압축하라.
`/GUI/d`	“GUI”를 포함하는 모든 줄을 지워라.
`s/GUI//g`	“GUI”가 나오는 줄에서 “GUI”만 지워라.

> **참고:** 어떤 단어를 길이가 0인 문자열로 치환하는 것은 그 줄에서 그 단어만 지우는 것과 똑같습니다. 이렇게 하면 그 줄의 나머지 부분들은 아무 영향도 받지 않습니다. `어느 어플리케이션이나 가장 중요한 부분은` **`GUI`**`와 음향 효과이다.`라는 줄에 **`s/GUI//`**을 걸면
>
> ```
> 어느 어플리케이션이나 가장 중요한 부분은  와 음향 효과이다.
> ```
>
> 라고 됩니다.

> **작은 정보:** 텍스트 파일의 매 줄마다 빈 줄을 넣을 때는 **`sed G filename`**이라고 하면 아주 간단합니다.

쉘 스크립트에서 sed가 어떻게 쓰이는지를 보려면 다음을 참고하세요.

1. 예 34-1
2. 예 34-2
3. 예 12-2
4. 예 A-3
5. 예 12-12
6. 예 12-20
7. 예 A-7
8. 예 A-12
9. 예 12-24
10. 예 10-8
11. 예 12-29

12. 예 A-2
13. 예 12-10
14. 예 12-9

sed의 더욱 다양한 쓰임새를 알고 싶다면 *서지사항*에서 적당한 참고 자료를 찾아서 확인해 보기 바랍니다.

B.2. Awk

Awk

awk는 **C** 문법을 연상시키는 완전한 형태의 텍스트 처리 언어입니다. **awk**는 광범위한 연산자들과 뛰어난 성능을 가지고 있지만 여기서는 쉘 스크립트에서 유용하게 쓰이는 몇 가지만 살펴 보도록 하겠습니다.

awk는 입력된 각 줄을 필드로 나눕니다. 디폴트로, 필드는 공백문자로 분리된 연속된 문자들로 이루어 집니다만 이 구분자를 바꿀수도 있습니다. awk는 이렇게 파싱된 필드를 기준으로 동작하게 됩니다. 이런 작업들로 인해 awk가 구조화된 텍스트 파일, 특히 열과 행으로 나누어진 일관된 데이타 묶음(chunk)으로 된 테이블 처리에 이상적인 도구가 되는 것입니다.

쉘 스크립트안에서는 awk 코드를 강한 쿼우트(strong quote, 작은 따옴표)와 중괄호로 묶어줍니다.

```
awk '{print $3}'
# 세번째 필드를 표준출력으로 출력.

awk '{print $1 $5 $6}'
# 첫번째, 다섯번째, 여섯번째 필드를 출력.
```

실제로 awk **print** 명령어의 사용법을 살펴 봤는데, 이제 변수에 대한 특징만 알아보고 마치겠습니다. awk는 변수를 쉘 스크립트와 비슷하게 다루는데 쉘 스크립트보다는 약간 더 융통성이 있습니다.

```
{ total += ${column_number} }
```

column_number의 값을 “total”에 계속 더합니다. 마지막으로, “total”을 출력하려면 **END**를 써서 전체 처리 과정을 끝내야 합니다.

```
END { print total }
```

END와 짝을 이루는 **BEGIN**은 awk가 입력을 처리하기 전에, 실행할 코드 블럭 앞에 써줘야 합니다.

awk가 쉘 스크립트에서 쓰이는 예제는 다음을 참고하세요.

1. 예 11-8
2. 예 16-5
3. 예 12-24
4. 예 34-3
5. 예 9-18
6. 예 11-12
7. 예 28-1
8. 예 28-2
9. 예 10-3
10. 예 12-33
11. 예 9-21
12. 예 12-3

자, 여기까지가 우리가 다룰 awk의 모든 것입니다. 하지만 배울 것이 더 많으니까 *서지사항*에서 적당한 레퍼런스를 참고하세요.

주석

1. 주소 범위가 주어지지 않는다면 디폴트로 모든 줄을 처리합니다.

부록 C. 특별한 의미를 갖는 종료 코드

표 **C-1.** "예약된" 종료 코드

종료 코드 번호	뜻	예제	비고
`1`	광범위한 일반적 에러	let "var1 = 1/0"	"divide by zero"같은 잡다한 에러
`2`	bash 문서에 명시되어 있는 쉘 내장명령어의 오사용		거의 보기 힘들고 보통은 디폴트로 1번 종료 코드로 나타남
`126`	실행 불가능한 명령어의 구동		퍼미션 문제거나 실행 허가가 없는 명령어
`127`	"command not found"		`$PATH` 문제거나 오타일 가능성 있음
`128`	exit에 잘못된 인자 넘김	exit 3.14159	**exit**는 **0**에서 **255**사이의 정수만 받음
`128+n`	치명적 에러 시그널 "n"	**kill -9** 스크립트의 **`$PPID`**	**`$?`**는 **`137 (128 + 9)`**을 리턴
`130`	스크립트가 Control-C 에 의해 종료됨		Control-C 는 치명적 에러 시그널 2번(130 = 128 + 2, 바로 위 참고)
`255`	종료 상태 범위 초과	exit -1	**exit**는 **0**에서 **255**사이의 정수만 받음

위의 테이블을 보면 1 - 2, 126 - 165, 255 번 종료 코드는 특별한 의미를 갖고 있기 때문에 사용자가 임의대로 exit 의 매개변수로 쓰면 안 됩니다. 예를 들어, 스크립트를 **exit 127**로 끝내면 나중에 문제가 생겨 해결하려고 할 때에 혼란을 가져올 수 있습니다(이 에러가 "command not found"인지 사용자가 정의한 것인지). 그렇긴 하지만 많은 스크립트들이 에러 발생시 종료를 위해 보통 **exit 1**을 씁니다. 1번 종료 코드는 생길수 있는 다양한 형태의 에러를 나타내기 때문에 사용자 정의 에러도 충분히 포함될 수 있습니다. 하지만 다른 면에서 보면 이렇게 했을 경우 유용한 정보를 나타내지 않는다는 것도 알아두세요.

종료 상태 번호(`/usr/include/sysexits.h` 참고)를 체계적으로 분류하려는 노력이 있었지만 이는 C, C++ 프로그래머들을 위한 것이었습니다. 하지만 이를 스크립트용으로 비슷하게 적용해도 괜찮을 듯 싶습니다. 저자가 제안하는 방법은 사용자 정의 종료 코드를 64 - 113(성공시 0도 포함)으로 제한해서 C/C++ 표준을 따르는 것입니다. 이렇게 해도 사용자는 여전히 50개의 코드를 쓸 수 있는 있기 때문에 나중에 스크립트의 문제를 좀 더 깔끔하게 해결할 수 있습니다.

이 문서에서 소개해 드린 예 9-2를 제외한 모든 예제들에서 쓰이는 사용자 정의 종료 코드는

지금 설명 드린 표준을 따릅니다.

> 참고**:** 쉘 스크립트가 종료한 뒤 명령어 줄에서 $?를 쳐서 나오는 결과는 위의 테이블에서 설명드린것과 일치하지만 이는 오직 bash나 **sh**에서만 적용됩니다. C 쉘이나 **tcsh**의 경우에는 몇몇 경우에 다른 결과값이 나옵니다.

부록 D. I/O와 I/O 재지향에 대한 자세한 소개

Stephane Chazelas가 작성, 본 문서의 저자가 교정

어떤 명령어든 처음 세 개의 파일 디스크립터가 사용 가능하기를 기대합니다. 첫 번째는 읽기용인 **fd 0**(표준입력)이고, 나머지 둘은 쓰기용으로 **fd 1**과 **fd 2**입니다.

표준입력(`stdin`), 표준출력(`stdout`), 표준에러(`stderr`)는 각 명령어들과 연관이 있습니다. **`ls 2>&1`** 은 **ls** 명령어의 표준에러를 임시로 쉘의 표준출력과 동일한 "리소스"로 연결시켜 줍니다.

관습적으로 한 명령어는 fd 0 (표준입력)에서 입력을 읽고, fd 1 (표준출력)으로 보통의 결과들을 출력하고, 에러 출력은 fs 2 (표준에러)로 합니다. 만약에 이들중 하나라도 열려 있지 않다면 문제가 생길지도 모릅니다.

```
bash$ cat /etc/passwd >&-
cat: standard output: Bad file descriptor
```

예를 들어, **xterm**을 실행시키면 먼저 자신을 초기화하고 사용자 쉘을 띄우기 전에 터미널 디바이스(/dev/pts/<n> 나 비슷한것)를 세 번 엽니다.

이 시점에서 Bash 는 이 세 개의 파일 디스크립터들을 상속받고 Bash 에서 실행시키는 각각의 명령어(자식 프로세스)들은 다시 차례대로 이 디스크립터들을 상속받습니다. 재지향이란 이 파일 디스크립터중의 하나를 다른 파일(혹은 파이프나 사용 가능한 모든 것)로 재할당 시키는 것입니다. 파일 디스크립터는 지역적(locally)으로 재할당 될 수도 있고(명령어, 명령어 그룹, 서브쉘, while, if, case, for 루프 등을 위해서), 쉘의 나머지 부분을 위해서 전역적(globally)으로 될 수도 있습니다(exec를 써서).

`ls > /dev/null` 은 **ls**의 fs 1 을 `/dev/null`으로 연결해서 실행하라는 뜻입니다.

```
bash$ lsof -a -p $$ -d0,1,2
COMMAND PID     USER   FD   TYPE DEVICE SIZE NODE NAME
 bash    363 bozo        0u   CHR  136,1         3 /dev/pts/1
 bash    363 bozo        1u   CHR  136,1         3 /dev/pts/1
 bash    363 bozo        2u   CHR  136,1         3 /dev/pts/1


bash$ exec 2> /dev/null
bash$ lsof -a -p $$ -d0,1,2
COMMAND PID     USER   FD   TYPE DEVICE SIZE NODE NAME
 bash    371 bozo        0u   CHR  136,1         3 /dev/pts/1
 bash    371 bozo        1u   CHR  136,1         3 /dev/pts/1
 bash    371 bozo        2w   CHR    1,3       120 /dev/null


bash$ bash -c 'lsof -a -p $$ -d0,1,2' | cat
COMMAND PID USER   FD   TYPE DEVICE SIZE NODE NAME
 lsof    379 root    0u   CHR  136,1         3 /dev/pts/1
```

```
 lsof    379 root    1w  FIFO    0,0      7118 pipe
 lsof    379 root    2u   CHR  136,1         3 /dev/pts/1


bash$ echo "$(bash -c ’lsof -a -p $$ -d0,1,2’ 2>&1)"
COMMAND PID USER   FD   TYPE DEVICE SIZE NODE NAME
 lsof    426 root    0u   CHR  136,1         3 /dev/pts/1
 lsof    426 root    1w  FIFO    0,0      7520 pipe
 lsof    426 root    2w  FIFO    0,0      7520 pipe
```

이것은 다른 형태의 재지향에서도 동작합니다.

연습문제: 다음 스크립트를 분석해 보세요.

```
#! /usr/bin/env bash

mkfifo /tmp/fifo1 /tmp/fifo2
while read a; do echo "FIFO1: $a"; done < /tmp/fifo1 &
exec 7> /tmp/fifo1
exec 8> >(while read a; do echo "FD8: $a, to fd7"; done >&7)

exec 3>&1
(
 (
  (
   while read a; do echo "FIFO2: $a"; done < /tmp/fifo2 | tee /dev/stderr | tee /dev/fd/4 |
   exec 3> /tmp/fifo2

   echo 1st, to stdout
   sleep 1
   echo 2nd, to stderr >&2
   sleep 1
   echo 3rd, to fd 3 >&3
   sleep 1
   echo 4th, to fd 4 >&4
   sleep 1
   echo 5th, to fd 5 >&5
   sleep 1
   echo 6th, through a pipe | sed ’s/.*/PIPE: &, to fd 5/’ >&5
   sleep 1
   echo 7th, to fd 6 >&6
   sleep 1
   echo 8th, to fd 7 >&7
   sleep 1
   echo 9th, to fd 8 >&8

  ) 4>&1 >&3 3>&- | while read a; do echo "FD4: $a"; done 1>&3 5>&- 6>&-
 ) 5>&1 >&3 | while read a; do echo "FD5: $a"; done 1>&3 6>&-
) 6>&1 >&3 | while read a; do echo "FD6: $a"; done 3>&-
```

```
rm -f /tmp/fifo1 /tmp/fifo2

# 각 명령어와 서브쉘에 대해서 어떤 fs 가 무엇을 가르키는지를 알아내 보세요.

exit 0
```

부록 E. 지역화(Localization)

지역화는 문서화가 안 돼 있는 Bash 의 특징입니다.

지역화된 쉘 스크립트는 시스템의 로케일에 따라 정의된 언어로 텍스트를 출력해 줍니다. 독일의 베를린에 있는 리눅스 유저는 독일어로 된 스크립트의 출력을 얻을 수 있는 반면에 미국의 메릴랜드에 있는 그의 사촌은 똑같은 스크립트임에도 불구하고 영어로 된 출력을 얻을 수 있습니다.

사용자에게 에러나 프롬프트등을 지역화된 메세지로 보여주려면 다음에 나오는 형식을 쓰면 됩니다.

```
#!/bin/bash
# localized.sh

E_CDERROR=65

error()
{
  printf "$@" >&2
  exit $E_CDERROR
}

cd $var || error $"Can't cd to %s." "$var"
read -p $"Enter the value: " var
# ...
```

```
bash$ bash -D localized.sh
"Can't cd to %s."
 "Enter the value: "
```

이 목록들은 지역화된 텍스트를 보여줍니다.(`-D` 옵션은 스크립트를 실행시키지 않고 $ 뒤에서 큰따옴표로 쿼우트된 문자열들을 보여줍니다.)

```
bash$ bash --dump-po-strings localized.sh
#: a:6
msgid "Can't cd to %s."
msgstr ""
#: a:7
msgid "Enter the value: "
msgstr ""
```

`--dump-po-strings` 옵션은 `-D`와 닮았지만 gettext의 “po” 포맷을 사용합니다.

자 이제 스크립트가 *`msgstr`*을 참고해서 특정 언어로 메세지를 번역하도록 `languge.po`를 빌드하면 됩니다. 예를 살펴 볼까요?

ko.po:

```
#: a:6
msgid "Can't cd to %s."
msgstr "%s 디렉토리로 옮겨갈 수 없습니다."
#: a:7
msgid "Enter the value: "
msgstr "값을 넣으세요: "
```

그 다음엔 **msgfmt**을 실행시키세요.

```
msgfmt -o localized.sh.mo ko.po
```

그러면 `localized.sh.mo` 파일이 생기는데 이 파일을 `/usr/local/share/locale/ko/LC_MESSAGES` 디렉토리에다 두고 스크립트의 첫 부분에 다음 두 줄을 추가해 주세요.

```
TEXTDOMAINDIR=/usr/local/share/locale
TEXTDOMAIN=localized.sh
```

이제 어떤 사용자가 이 스크립트를 한국어 환경에서 실행시킨다면 영어 메세지 대신에 한글 메세지를 보게 될 것입니다.

> 참고: Bash 예전 버전이나 다른 쉘에서 지역화를 하려면 gettext에 `-s` 옵션을 걸어서 사용해야 합니다. 이런 경우에 쉘 스크립트는 다음처럼 해 주면 됩니다.

```
#!/bin/bash
# localized.sh

E_CDERROR=65

error() {
  local format=$1
  shift
  printf "$(gettext -s "$format")" "$@" >&2
  exit $E_CDERROR
}
cd $var || error "Can't cd to %s." "$var"
read -p "$(gettext -s "Enter the value: ")" var
# ...
```

`TEXTDOMAIN` 과 `TEXTDOMAINDIR` 변수는 전체 환경으로 export 되어야 합니다.

이 부록은 Stephane Chazelas 가 작성해 주었습니다.

부록 F. 샘플 .bashrc 파일

`~/.bashrc` 파일은 대화형(interactive) 모드 쉘의 행동을 결정합니다. 이 파일을 잘 공부하면 Bash 를 더욱 잘 이해할 수 있을 것입니다.

Emmanuel Rouat (mailto:emmanuel.rouat@wanadoo.fr) 가 아주 정교한 리눅스용 `.bashrc` 스크립트를 제공해 주었습니다. 이 파일을 아주 자세히 공부해서 여기 나오는 코드 조각들이나 함수들을 여러분의 `.bashrc` 나 스크립트에서 마음껏 사용하기 바랍니다.

예 **F-1.** 샘플 **`.bashrc`** 파일

```
#===============================================================
#
# bash-2.05 이후 버전을 위한 개인적 $HOME/.bashrc 파일
#
# 아 파일은 대화모드 쉘을 위한 것입니다.
# 별칭(alias)이나 함수, 프롬프트같은
# 대화모드용 기능들을 여기에 두면 됩니다.
#
# 원래는 솔라리스를 위해 디자인 되었습니다.
# --> 리눅스용으로 수정
# 이 파일은 너무 많은 것을 포함하고 있지만
# 단지 예제라는 것을 기억하세요.
# 여러분 필요에 따라 수정해서 쓰기 바랍니다.
#
#===============================================================

# --> 본 문서 저자에 의한 주석.

#-----------------------------------
# 필요하다면 전역 정의를 source
#-----------------------------------

if [ -f /etc/bashrc ]; then
        . /etc/bashrc   # --> 있다면 /etc/bashrc 를 읽음.
fi

#-------------------------------------------------------------
# 아직 세트되지 않았다면 $DISPLAY 를 자동으로 세팅
# 이 부분은 리눅스와 솔라리스용입니다 - 필요한대로 고쳐 쓰세요...
#-------------------------------------------------------------

if [ -z ${DISPLAY:=""} ]; then
    DISPLAY=$(who am i)
    DISPLAY=${DISPLAY%%\!*}
    if [ -n "$DISPLAY" ]; then
        export DISPLAY=$DISPLAY:0.0
    else
```

```
        export DISPLAY=":0.0"  # 실패할 경우를 대비(fallback)
    fi
fi

#---------------
# 몇 가지 세팅
#---------------

set -o notify
set -o noclobber
set -o ignoreeof
set -o nounset
#set -o xtrace          # 디버깅용

shopt -s cdspell
shopt -s cdable_vars
shopt -s checkhash
shopt -s checkwinsize
shopt -s mailwarn
shopt -s sourcepath
shopt -s no_empty_cmd_completion
shopt -s histappend histreedit
shopt -s extglob        # programmable completion에 유용

#-----------------------
# 인사말, motd 등등...
#-----------------------

# 먼저 색깔을 몇 개 정의:
red='\e[0;31m'
RED='\e[1;31m'
blue='\e[0;34m'
BLUE='\e[1;34m'
cyan='\e[0;36m'
CYAN='\e[1;36m'
NC='\e[0m'              # No Color
# --> 좋군요. 도스에서 "ansi.sys"를 쓰는 것과 똑같은 효과가 있네요.

# 검정색 백그라운드에서 가장 좋게 보입니다.....
echo -e "${CYAN}This is BASH ${RED}${BASH_VERSION%.*}${CYAN} -
DISPLAY on ${RED}$DISPLAY${NC}\n"
date
if [ -x /usr/games/fortune ]; then
    /usr/games/fortune -s     # 하루를 즐겁게.... :-)
fi

function _exit()        # 쉘에서 종료시 실행할 함수
{
    echo -e "${RED}나중에 또 봐요${NC}"
```

```
}
trap _exit 0

#---------------
# 쉘 프롬프트
#---------------

function fastprompt()
{
    unset PROMPT_COMMAND
    case $TERM in
        *term | rxvt )
            PS1="[\h] \W > \[\033]0;[\u@\h] \w\007\]" ;;
        *)
            PS1="[\h] \W > " ;;
    esac
}

function powerprompt()
{
    _powerprompt()
    {
        LOAD=$(uptime|sed -e "s/.*: \([^,]*\).*/\1/" -e "s/ //g")
        TIME=$(date +%H:%M)
    }

    PROMPT_COMMAND=_powerprompt
    case $TERM in
        *term | rxvt  )
            PS1="${cyan}[\$TIME \$LOAD]$NC\n[\h \#] \W > \[\033]0;[\u@\h] \w\007\]" ;;
        linux )
            PS1="${cyan}[\$TIME - \$LOAD]$NC\n[\h \#] \w > " ;;
        * )
            PS1="[\$TIME - \$LOAD]\n[\h \#] \w > " ;;
    esac
}

powerprompt     # 좀 느릴지도 모를 기본 프롬프트입니다.
                # 너무 느리면 fastprompt 를 쓰세요....

#================================================================
#
# 별칭(alias)과 함수들
#
# 논쟁의 여지가 있지만 몇몇 함수들은 조금 덩치가 큰데(즉, 'lowercase')
# 제 워크스테이션은 램이 512메가거든요...
# 이 파일 크기를 줄이고 싶다면 이런 함수들은 스크립트로 빼도 됩니다.
#
# 많은 함수들은 bash-2.04 예제에서 거의 그대로 갖다 썼습니다.
```

```
#
#===============================================================

#-------------------------
# 개인적인 별칭들(Aliases)
#-------------------------

alias rm=’rm -i’
alias cp=’cp -i’
alias mv=’mv -i’
# -> 파일에 실수로 타격을 입히지 않게.

alias h=’history’
alias j=’jobs -l’
alias r=’rlogin’
alias which=’type -all’
alias ..=’cd ..’
alias path=’echo -e ${PATH//:/\\n}’
alias print=’/usr/bin/lp -o nobanner -d $LPDEST’   # LPDEST 가 정의되어 있다
고 가정
alias pjet=’enscript -h -G -fCourier9 -d $LPDEST’  # enscript 로 예쁜 출력하
기(Pretty-print)
alias background=’xv -root -quit -max -rmode 5’ # 백그라운드 배경 그림
alias vi=’vim’
alias du=’du -h’
alias df=’df -kh’

# ’ls’ 그룹(여러분이 GNU ls 를 쓴다고 가정)
alias ls=’ls -hF --color’ # 파일타입 인식을 위해 색깔을 추가
alias lx=’ls -lXB’ # 확장자별로 정렬
alias lk=’ls -lSr’ # 크기별로 정렬
alias la=’ls -Al’ # 숨겨진 파일 보기
alias lr=’ls -lR’ # 재귀적 ls
alias lt=’ls -ltr’ # 날짜별로 정렬
alias lm=’ls -al |more’ # ’more’로 파이프 걸기
alias tree=’tree -Cs’ # ’ls’의 멋진 대용품


# 맞춤 ’less’
alias more=’less’
export PAGER=less
export LESSCHARSET=’latin1’
export LESSOPEN=’|/usr/bin/lesspipe.sh %s 2>&-’ # lesspipe.sh 이 있다면 이
걸 쓰세요
export LESS=’-i -N -w  -z-4 -g -e -M -X -F -R -P%t?f%f \
:stdin .?pb%pb\%:?lbLine %lb:?bbByte %bb:-...’

# 스펠링 오타용 - 아주 개인적임 :-)
alias xs=’cd’
```

```
alias vf=’cd’
alias moer=’more’
alias moew=’more’
alias kk=’ll’

#----------------
# 재밌는 거 몇 개
#----------------

function xtitle ()
{
    case $TERM in
        *term | rxvt)
            echo -n -e "\033]0;$*\007" ;;
        *)  ;;
    esac
}

# 별칭들(aliases)...
alias top=’xtitle Processes on $HOST && top’
alias make=’xtitle Making $(basename $PWD) ; make’
alias ncftp="xtitle ncFTP ; ncftp"

# .. 과 함수들
function man ()
{
    xtitle The $(basename $1|tr -d .[:digit:]) manual
    man -a "$*"
}

function ll(){ ls -l "$@"| egrep "^d" ; ls -lXB "$@" 2>&-| egrep -
v "^d|total "; }
function xemacs() { { command xemacs -private $* 2>&- & } && disown ;}
function te()  # xemacs/gnuserv 래퍼
{
    if [ "$(gnuclient -batch -eval t 2>&-)" == "t" ]; then
        gnuclient -q "$@";
    else
        ( xemacs "$@" & );
    fi
}

#-----------------------------------
# 파일 & 문자열 관련 함수들:
#-----------------------------------

function ff() { find . -name ’*’$1’*’ ; }                 # 파일 찾기
function fe() { find . -name ’*’$1’*’ -exec $2 {} \; ; }  # 파일을 찾아
서 $2 의 인자로 실행
```

```
function fstr() # 여러 파일중에서 문자열 찾기
{
    if [ "$#" -gt 2 ]; then
        echo "Usage: fstr \"pattern\" [files] "
        return;
    fi
    SMSO=$(tput smso)
    RMSO=$(tput rmso)
    find . -type f -name "${2:-*}" -print | xargs grep -sin "$1" | \
sed "s/$1/$SMSO$1$RMSO/gI"
}

function cuttail() # 파일에서 끝의 n 줄을 잘라냄. 기본값은 10
{
    nlines=${2:-10}
    sed -n -e :a -e "1,${nlines}!{P;N;D;};N;ba" $1
}

function lowercase()  # 파일이름을 소문자로 변경
{
    for file ; do
        filename=${file##*/}
        case "$filename" in
        */*) dirname==${file%/*} ;;
        *) dirname=.;;
        esac
        nf=$(echo $filename | tr A-Z a-z)
        newname="${dirname}/${nf}"
        if [ "$nf" != "$filename" ]; then
            mv "$file" "$newname"
            echo "lowercase: $file --> $newname"
        else
            echo "lowercase: $file not changed."
        fi
    done
}

function swap()         # 파일이름 두개를 서로 바꿈
{
    local TMPFILE=tmp.$$
    mv $1 $TMPFILE
    mv $2 $1
    mv $TMPFILE $2
}

#-----------------------------------
# 프로세스/시스템 관련 함수들:
#-----------------------------------
```

```
function my_ps() { ps $@ -u $USER -o pid,%cpu,%mem,bsdtime,command ; }
function pp() { my_ps f | awk '!/awk/ && $0~var' var=${1:-".*"} ; }

# 이 함수는 리눅스의 'killall' 스크립트와 거의 비슷하지만
# 솔라리스에는, 제가 아는 한, 이와 비슷한 것이 없습니다.
function killps()   # 프로세스 이름으로 kill
{
    local pid pname sig="-TERM"   # 기본 시그널
    if [ "$#" -lt 1 ] || [ "$#" -gt 2 ]; then
echo "Usage: killps [-SIGNAL] pattern"
return;
    fi
    if [ $# = 2 ]; then sig=$1 ; fi
    for pid in $(my_ps| awk '!/awk/ && $0~pat { print $1 }' pat=${!#} ) ; do
pname=$(my_ps | awk '$1~var { print $5 }' var=$pid )
if ask "Kill process $pid <$pname> with signal $sig?"
    then kill $sig $pid
fi
    done
}

function my_ip() # IP 주소 알아내기
{
    MY_IP=$(/sbin/ifconfig ppp0 | awk '/inet/ { print $2 } ' | sed -
e s/addr://)
    MY_ISP=$(/sbin/ifconfig ppp0 | awk '/P-t-P/ { print $3 } ' | sed -
e s/P-t-P://)
}

function ii()   # 현재 호스트 관련 정보들 알아내기
{
    echo -e "\nYou are logged on ${RED}$HOST"
    echo -e "\nAdditionnal information:$NC " ; uname -a
    echo -e "\n${RED}Users logged on:$NC " ; w -h
    echo -e "\n${RED}Current date :$NC " ; date
    echo -e "\n${RED}Machine stats :$NC " ; uptime
    echo -e "\n${RED}Memory stats :$NC " ; free
    my_ip 2>&- ;
    echo -e "\n${RED}Local IP Address :$NC" ; echo ${MY_IP:-
"Not connected"}
    echo -e "\n${RED}ISP Address :$NC" ; echo ${MY_ISP:-"Not connected"}
    echo
}


# 기타 유틸리티:

function repeat()       # 명령어를 n 번 반복
{
```

```
    local i max
    max=$1; shift;
    for ((i=1; i <= max ; i++)); do  # --> C 형태의 문법
        eval "$@";
    done
}


function ask()
{
    echo -n "$@" '[y/n] ' ; read ans
    case "$ans" in
        y*|Y*) return 0 ;;
        *) return 1 ;;
    esac
}

#=========================================================================
#
# PROGRAMMABLE COMPLETION - 오직 BASH 2.04 이후에서만 동작
# (거의 대부분은 bash 2.05 문서에서 가져왔습니다)
# 몇 가지 기능들을 쓰려면 bash-2.05 가 필요할 겁니다.
#
#=========================================================================

if [ "${BASH_VERSION%.*}" \< "2.05" ]; then
    echo "programmable completion 을 쓰려면 bash 2.05 이상으로 업그레이드가 필요
합니다."
    return
fi

shopt -s extglob        # 꼭 필요함
set +o nounset # 이렇게 안 하면 programmable completion 몇 가지는 실패함

# 옮긴이: 이 이후는 잘 모르겠네요. :(

complete -A hostname   rsh rcp telnet rlogin r ftp ping disk
complete -A command    nohup exec eval trace gdb
complete -A command    command type which
complete -A export     printenv
complete -A variable   export local readonly unset
complete -A enabled    builtin
complete -A alias      alias unalias
complete -A function   function
complete -A user       su mail finger

complete -A helptopic  help     # currently same as builtins
complete -A shopt      shopt
complete -A stopped -P '%' bg
```

```
complete -A job -P '%'     fg jobs disown

complete -A directory  mkdir rmdir
complete -A directory   -o default cd

complete -f -d -X '*.gz'  gzip
complete -f -d -X '*.bz2' bzip2
complete -f -o default -X '!*.gz'  gunzip
complete -f -o default -X '!*.bz2' bunzip2
complete -f -o default -X '!*.pl'  perl perl5
complete -f -o default -X '!*.ps'  gs ghostview ps2pdf ps2ascii
complete -f -o default -X '!*.dvi' dvips dvipdf xdvi dviselect dvitype
complete -f -o default -X '!*.pdf' acroread pdf2ps
complete -f -o default -X '!*.+(pdf|ps)' gv
complete -f -o default -X '!*.texi*' makeinfo texi2dvi texi2html texi2pdf
complete -f -o default -X '!*.tex' tex latex slitex
complete -f -o default -X '!*.lyx' lyx
complete -f -o default -X '!*.+(jpg|gif|xpm|png|bmp)' xv gimp
complete -f -o default -X '!*.mp3' mpg123
complete -f -o default -X '!*.ogg' ogg123


# This is a 'universal' completion function - it works when commands have
# a so-called 'long options' mode , ie: 'ls --all' instead of 'ls -a'
_universal_func ()
{
    case "$2" in
-*) ;;
*) return ;;
    esac

    case "$1" in
\~*) eval cmd=$1 ;;
*) cmd="$1" ;;
    esac
    COMPREPLY=( $("$cmd" --help | sed  -e '/--/!d' -e 's/.*--\([^ ]*\).*/-
-\1/'| \
grep ^"$2" |sort -u) )
}
complete  -o default -F _universal_func ldd wget bash id info


_make_targets ()
{
    local mdef makef gcmd cur prev i

    COMPREPLY=()
    cur=${COMP_WORDS[COMP_CWORD]}
    prev=${COMP_WORDS[COMP_CWORD-1]}
```

```
    # if prev argument is -f, return possible filename completions.
    # we could be a little smarter here and return matches against
    # ‘makefile Makefile *.mk’, whatever exists
    case "$prev" in
        -*f)    COMPREPLY=( $(compgen -f $cur ) ); return 0;;
    esac

    # if we want an option, return the possible posix options
    case "$cur" in
        -)      COMPREPLY=(-e -f -i -k -n -p -q -r -S -s -t); return 0;;
    esac

    # make reads ‘makefile’ before ‘Makefile’
    if [ -f makefile ]; then
        mdef=makefile
    elif [ -f Makefile ]; then
        mdef=Makefile
    else
        mdef=*.mk               # local convention
    fi

    # before we scan for targets, see if a makefile name was specified
    # with -f
    for (( i=0; i < ${#COMP_WORDS[@]}; i++ )); do
        if [[ ${COMP_WORDS[i]} == -*f ]]; then
            eval makef=${COMP_WORDS[i+1]}       # eval for tilde expansion
            break
        fi
    done

        [ -z "$makef" ] && makef=$mdef

    # if we have a partial word to complete, restrict completions to
    # matches of that word
    if [ -n "$2" ]; then gcmd=’grep "^$2"’ ; else gcmd=cat ; fi

    # if we don’t want to use *.mk, we can take out the cat and use
    # test -f $makef and input redirection
    COMPREPLY=( $(cat $makef 2>/dev/null | awk ’BE-
GIN {FS=":"} /^[^.#   ][^=]*:/ {print $1}’ | tr -s ’ ’ ’\012’ | sort -
u | eval $gcmd ) )
}

complete -F _make_targets -X ’+($*|*.[cho])’ make gmake pmake

_configure_func ()
{
    case "$2" in
```

```
        -*)     ;;
        *)      return ;;
    esac

    case "$1" in
        \~*)    eval cmd=$1 ;;
        *)      cmd="$1" ;;
    esac

    COMPREPLY=( $("$cmd" --help | awk '{if ($1 ~ /--
.*/) print $1}' | grep ^"$2" | sort -u) )
}

complete -F _configure_func configure

# cvs(1) completion
_cvs ()
{
    local cur prev
    COMPREPLY=()
    cur=${COMP_WORDS[COMP_CWORD]}
    prev=${COMP_WORDS[COMP_CWORD-1]}

    if [ $COMP_CWORD -eq 1 ] || [ "${prev:0:1}" = "-" ]; then
COMPREPLY=( $( compgen -W 'add admin checkout commit diff \
export history import log rdiff release remove rtag status \
tag update' $cur ))
    else
COMPREPLY=( $( compgen -f $cur ))
    fi
    return 0
}
complete -F _cvs cvs


_killall ()
{
    local cur prev
    COMPREPLY=()
    cur=${COMP_WORDS[COMP_CWORD]}

    # get a list of processes (the first sed evaluation
    # takes care of swapped out processes, the second
    # takes care of getting the basename of the process)
    COMPREPLY=( $( /usr/bin/ps -u $USER -o comm  | \
        sed -e '1,1d' -e 's#[]\[]##g' -e 's#^.*/##'| \
        awk '{if ($0 ~ /^'$cur'/) print $0}' ))

    return 0
```

```
}

complete -F _killall killall killps

# Local Variables:
# mode:shell-script
# sh-shell:bash
# End:
```

부록 G. 도스(DOS) 배치 파일을 쉘 스크립트로 변환

아주 많은 프로그래머들은 PC 위에서 도는 도스에서 스크립트를 배웠습니다. 기능이 조금 떨어지는 도스 배치 파일 언어로도 꽤 강력한 스크립트나 어플리케이션을 작성할 수도 있지만 그렇게 하려면 아주 해박한 지식을 사용해 해결책을 찾거나 꽁수를 부려야 합니다. 가끔은 오래된 도스용 배치 파일을 유닉스 쉘 스크립트로 변환해서 써야될 경우가 생기지만 이렇게 하는것이 그렇게 어렵지만은 않습니다. 왜냐하면 도스 배치 파일 연산자들이 기능이 동일한 쉘 스크립트 연산자의 서브셋에 불과하기 때문입니다.

표 **G-1.** 배치 파일 키워드/변수/연산자 와 그에 해당하는 쉘 동의어

배치 파일 연산자	쉘 스크립트 동의어	뜻
`%`	$	명령어줄 매개변수 접두사
`/`	-	명령어 옵션 플래그
`\`	/	디렉토리 패스 구분자
`==`	=	(같음) 문자열 비교 테스트
`!==!`	!=	(다름) 문자열 비교 테스트
`|`	|	파이프
`@`	set +v	현재 명령어를 에코하지 말 것
`*`	*	파일명 “와일드 카드”
`>`	>	파일 재지향(덮어 쓰기)
`>>`	>>	파일 재지향(덧붙여 쓰기)
`<`	<	표준입력 재지향
`%VAR%`	$VAR	환경 변수
`REM`	#	주석
`NOT`	!	뒤에 나오는 테스트 부정
`NUL`	`/dev/null`	명령어 출력을 없애기 위한 “블랙홀”
`ECHO`	echo	에코 (Bash 에는 옵션이 많이 있음)
`ECHO.`	echo	빈 줄 에코
`ECHO OFF`	set +v	다음에 나오는 명령어를 에코하지 말 것
`FOR %%VAR IN (LIST) DO`	for var in [list]; do	“for” 루프
`:LABEL`	없음 (필요치 않음)	라벨

배치 파일 연산자	쉘 스크립트 동의어	뜻
GOTO	없음 (대신 함수를 씀)	스크립트의 다른 곳으로 건너뜀
PAUSE	sleep	일정 간격을 두고 잠시 대기
CHOICE	case 나 select	메뉴 선택
IF	if	if-test
IF EXIST *FILENAME*	if [-e filename]	파일이 존재하는지 확인
IF !%N==!	if [-z "$N"]	변경가능한 매개변수인 "N"이 없다면
CALL	source 나 . (도트 연산자)	다른 스크립트를 "포함"
COMMAND /C	source 나 . (도트 연산자)	다른 스크립트를 "포함"(**CALL**과 동일)
SET	export	환경 변수를 세트
SHIFT	shift	명령어줄 변수 목록을 왼쪽으로 이동(shift)
SGN	-lt or -gt	(정수) 부호(sign)
ERRORLEVEL	$?	종료 상태
CON	stdin	"콘솔"(표준입력)
PRN	/dev/lp0	(일반적인) 프린터 디바이스
LP1	/dev/lp0	첫번째 프린터 디바이스
COM1	/dev/ttyS0	첫번째 시리얼 포트

배치 파일은 대개 도스 명령어를 갖고 있습니다. 도스용 배치 파일이 쉘 스크립트로 변환되기 위해서는 이 명령어들은 꼭 동일한 유닉스 명령어로 변환되어야 합니다.

표 **G-2.** 도스 명령어와 동일한 유닉스 명령어

도스 명령어	동일한 유닉스 명령어	효과
ASSIGN	ln	파일이나 디렉토리를 링크
ATTRIB	chmod	파일 퍼미션 변경
CD	cd	디렉토리 변경
CHDIR	cd	디렉토리 변경
CLS	clear	스크린 지우기
COMP	cmp or diff	파일 비교
COPY	cp	파일 복사
Ctl-C	Ctl-C	정지(시그널)
Ctl-Z	Ctl-D	EOF (end-of-file)

도스 명령어	동일한 유닉스 명령어	효과
`DEL`	rm	파일 삭제
`DELTREE`	rm -rf	디렉토리의 하위 디렉토리까지 포함해서 삭제
`DIR`	ls -l	디렉토리 보이기
`ERASE`	rm	파일 삭제
`EXIT`	exit	현재 프로세스 종료
`FC`	comm, cmp	파일 비교
`FIND`	grep	파일안에서 문자열 찾기
`MD`	mkdir	디렉토리 생성
`MKDIR`	mkdir	디렉토리 생성
`MORE`	more	텍스트 파일 쪽단위(paging) 필터
`MOVE`	mv	이동
`PATH`	$PATH	실행파일들의 경로
`REN`	mv	이름 바꾸기(이동)
`RENAME`	mv	이름 바꾸기(이동)
`RD`	rmdir	디렉토리 삭제
`RMDIR`	rmdir	디렉토리 삭제
`SORT`	sort	파일 정렬
`TIME`	date	시스템 시간 보여주기
`TYPE`	cat	파일을 표준출력으로 출력
`XCOPY`	cp	(확장) 파일 복사

> **참고:** 사실 모든 유닉스, 쉘 연산자, 명령어들은 그들과 동일한 도스용보다 많은 옵션과 강력한 기능을 갖고 있습니다. 많은 배치 파일 스크립트들은 read의 불완전한 버전인 **ask.com**같은 외부 유틸리티에 의존합니다.
>
> 도스는 파일명 와일드 카드 확장에 대해서 오직 *과 ? 문자만을 인식하는 제한되고 부족한 서브셋을 지원합니다.

도스 배치 파일을 쉘 스크립트로 변환하는 것은 일반적으로 매우 간단하고 가끔은 변환된 쉘 스크립트가 원래 도스 배치 파일보다 더 이해하기 쉬운 경우도 있습니다.

예 **G-1. VIEWDATA.BAT:** 도스용 배치 파일

```
REM VIEWDATA

REM PAUL SOMERSON의 "DOS POWERTOOLS"의 예제에서 영감을 받아 작성
```

```
@ECHO OFF

IF !%1==! GOTO VIEWDATA
REM  명령어줄 인자가 없다면...
FIND "%1" C:\BOZO\BOOKLIST.TXT
GOTO EXIT0
REM  문자열이 일치하는 줄을 출력후 종료.

:VIEWDATA
TYPE C:\BOZO\BOOKLIST.TXT | MORE
REM  한 번에 한 쪽씩 전체 파일을 보여줌.

:EXIT0
```

스크립트 변환을 하면 기능이 다소 개량됩니다.

예 G-2. viewdata.sh: VIEWDATA.BAT 의 스크립트 버전

```
#!/bin/bash
# VIEWDATA.BAT 를 쉘 스크립트로 변환.

DATAFILE=/home/bozo/datafiles/book-collection.data
ARGNO=1

# @ECHO OFF       여기서 이 명령어는 필요없습니다.

if [ $# -lt "$ARGNO" ]     # IF !%1==! GOTO VIEWDATA
then
  less $DATAFILE           # TYPE C:\MYDIR\BOOKLIST.TXT | MORE
else
  grep "$1" $DATAFILE      # FIND "%1" C:\MYDIR\BOOKLIST.TXT
fi

exit 0                     # :EXIT0

# GOTO, 라벨, 속임수, 엉터리 구문등이 필요없어졌죠.
# 원래 배치 파일보다 더 짧고, 더 쉽고, 더 깔끔합니다.
```

Ted Davis 의 Shell Scripts on the PC (http://www.maem.umr.edu/~batch/) 사이트에서는 구식 배치 파일 프로그래밍에 대한 포괄적인 튜토리얼을 소개하고 있습니다. 생각건대, 그의 독창적인 몇몇 테크닉들은 쉘 스크립트와 연관성이 있습니다.

부록 H. 연습문제

다음에 주어진 일들을 수행하는 스크립트를 짜 보세요.

초급

홈디렉토리 목록

사용자의 홈디렉토리 이하 모든 디렉토리의 목록을 나열하고 그 정보를 파일로 저장하세요. 그 다음 그 파일을 압축한 다음에 사용자가 플로피 디스켓을 준비하고 **ENTER**를 누르게 하세요. 끝으로, 파일을 디스켓으로 저장하세요.

for 루프를 **while** 과 **until** 루프로 바꾸기

예 10-1의 **for** 루프를 **while** 루프로 바꿔보세요. 힌트: 데이타를 배열에 저장하고 각 요소들에 대해 하나씩 처리하세요.

다 했나요? 그럼 이번에는 **until** 루프로 바꿔보세요.

소수(**Primes**)

60000 에서 63000 사이에 들어 있는 모든 소수를 표준출력으로 출력하세요. 결과가 깔끔한 형태로 나오도록 해야 합니다(힌트: printf 를 써보세요).

유일한 시스템 **ID**

여러분의 컴퓨터를 위한 "유일한" 6 자리 16진수 ID를 만들어 보세요. 결함이 많은 hostid 명령어를 쓰지 말고 **md5sum `/etc/passwd`**라고 한 다음 그 출력의 첫 6 자리를 쓰면 됩니다.

백업

여러분의 홈디렉토리(`/home/your-name`)에 들어 있는 파일중에 최근 24시간 동안 변경된 모든 파일을 "타르볼"(`*.tar.gz`)로 묶으세요. 힌트: find를 쓰세요.

안전한 삭제

이름을 `srm.sh`로 해서 "안전한" 삭제 명령어를 짜 보세요. 이 스크립트에 명령어줄 인자로 파일명을 적어 주면 그 파일은 지워지지 않고 gzip으로 묶여 `/home/username/trash` 디렉토리로 이동됩니다. 구동시에는 "trash" 디렉토리를 확인해서 48시간보다 오래된 파일들을 지우도록 하세요.

중급

디스크 공간 관리

`/home/username` 디렉토리에 들어 있는 파일중에서 100K 보다 큰 모든 파일을 한 번에 하나씩 나열하고 사용자가 그 파일을 지울것인지 압축할 것인지를 물어보게 하세요. 삭제된 모든 파일과 삭제 시간을 로그 파일로 쓰도록 하세요.

잔돈 계산

보통의 상황에서 단지 동전(25 센트까지)만 있다고 할 때, 1.68 달러를 잔돈으로 주려고 한다면 어떻게 해야 잘 줬다고 할 수 있을까요? 25 센트 6개, 10 센트 1개, 5 센트 1개, 3 센트면 되겠죠.

명령어줄에서 임의의 달러와 센트를 입력받아($*.??) 최소한의 동전을 사용해서 잔돈을 계산해 보세요. 여러분이 미국에 살지 않는다면 대신 여러분 나라에서 쓰이는 통화 단위를 쓰세요. 스크립트는 명령어줄 입력을 파싱해야 하고 그것을 최소 화폐 단위의 배수가 되도록 해야 합니다. 힌트: 예 23-4을 보세요.

행운의 숫자

"행운의 숫자"란 각 자리수를 계속 더해서 최종적으로 7 이 되는 숫자를 말합니다. 예를 들어 62431 은 6 + 2 + 4 + 3 + 1 = 16, 1 + 6 = 7 이 되기 때문에 행운의 숫자입니다. 1000 에서 10000 사이에 존재하는 모든 "행운의 숫자"를 찾아보세요.

문자열을 알페벳 순으로 표시하기

명령어줄에서 입력받은 임의의 문자열을 알파벳 순(아스키 순서로)으로 표시하세요.

파싱

`/etc/passwd` 을 파싱해서 탭이 들어간 보기 좋은 형태로 출력하세요.

데이타 파일을 예쁘게 출력하기

몇몇 데이타베이스와 스프레드시트들은 파일을 콤마로 구분된 값(**comma-separated values, CSVs**)으로 저장해 줍니다. 다른 어플리케이션에서 종종 이 파일을 파싱할 필요가 있습니다.

필드가 콤마로 구분된 데이타 파일이 다음과 같이 주어졌을때:

```
Jones,Bill,235 S. Williams St.,Denver,CO,80221,(303) 244-7989
Smith,Tom,404 Polk Ave.,Los Angeles,CA,90003,(213) 879-5612
...
```

이 데이타에 라벨을 붙이고 동일한 컬럼으로 재정렬해서 표준출력으로 출력해 보세요.

고급

파일 억세스 로깅하기

`/etc` 디렉토리에 있는 모든 파일에 대한 하루동안의 모든 접근을 로깅하세요. 로그에는 파일이름, 사용자 이름, 접근 시간이 포함되어야 합니다. 파일에 대해서 일어나는 어떤 변경사항이라도 모두 체크되어야 합니다. 이 모든 정보들을 하나의 로그파일에 깔끔한 포맷으로 저장하세요.

주석 지우기

명령어줄에서 주어진 쉘 스크립트에 들어 있는 모든 주석을 지우세요. “#! 이 들어 있는 줄”은 지워지면 안 된다는 것에 주의하세요.

HTML 변환

주어진 텍스트 파일을 HTML로 변환하세요. 이 비대화형 모드 스크립트는 주어진 파일에 대해 자동으로 적절한 HTML 태그를 삽입해야 합니다.

HTML 태그 지우기

주어진 HTML 파일에서 모든 HTML 태그를 지우고 각 줄에 60 에서 75 글자 정도만 들어가도록 하세요. 문단과 블럭 간격을 적당하게 재세팅하고 HTML 테이블은 최대한 비슷한 형태의 텍스트로 변환하세요.

헥사 덤프

주어진 이진 파일에 대해 헥사 덤프를 뜨세요. 출력의 첫번째 필드에는 주소가 오고 다음의 8개 필드에는 4 바이트 헥사값이 오고 마지막 필드에는 앞의 8개 필드에 해당하는 아스키값으 오도록 탭으로 구분하세요.

행렬식

4 x 4 행렬식을 계산하세요.

숨겨진 낱말

"낱말 찾기" 퍼즐(word-find puzzle) 생성기를 짜 보세요. 이 스크립트는 입력된 10개의 낱말을 랜덤한 글자들로 가득찬 10 x 10 행렬에 집어 넣어야 합니다. 이 낱말들은 가로나, 세로, 대각선으로 숨겨져 있을 수 있습니다.

철자 바꾸기**(Anagramming)**

4 글자짜리 입력을 받아 철자를 바꾸세요. 예를 들어, 입력이 **word** 라고 하면 **do or rod row word** 처럼 될 수 있겠죠. 참고 리스트가 필요하다면 `/usr/share/dict/linux.words` 를 써도 될 겁니다.

여러분이 생각한 해결책을 저자에게 보내지 말아 주세요. 여러분이 영리하다는 것을 저자에게 감동시키려면 차라리 이 책의 버그나 여러 제안들을 보내서 이 책을 더 발전시키는 것이 더 좋은 방법입니다.

부록 I. Copyright

The “Advanced Bash-Scripting Guide” is copyright, (c) 2000, by Mendel Cooper. This document may only be distributed subject to the terms and conditions set forth in the LDP License (http://www.linuxdoc.org/manifesto.html) These are very liberal terms, and they should not hinder any legitimate distribution or use of this book. The author especially encourages the use of this book, or portions thereof, for instructional purposes.

Hyun Jin Cha has done a Korean translation (http://kldp.org/HOWTO/html/Adv-Bash-Scr-HOWTO/index.html) of an earlier version of this book. Spanish and Portuguese translations are underway. If you wish to translate this document into another language, please feel free to do so, subject to the terms stated above. The author would appreciate being notified of such efforts.

If this document is printed as a hard-copy book, the author requests a courtesy copy. This is a request, not a requirement.