Unix & Linux 셸 스크립팅 튜토리얼

이 책은 Steve Parker의 Unix & Linux Shell Scripting Tutorial을 개인적으로 번역한 것입니다.

2. Philosophy

셸 스크립트 프로그래밍은 일부 Unix 시스템 관리자들 사이에서 좋지 않은 평가를 받고 있습니다. 이는 보통 두 가지 이유 중 하나 때문입니다:

- 해석된 프로그램이 실행되는 속도를 C 프로그램이나 심지어 해석된 Perl 프로그램과 비교하여 나타냅니다.
- 간단한 배치 작업 유형의 셸 스크립트는 쉽게 작성할 수 있기 때문에 품질이 떨어지는 셸 스크립트가 많이 있습니다.

이 때문에 좋은 셸 스크립트를 만드는 것과 관련된 특정 마초주의가 존재합니다. 예를 들어, Perl에 비해 속도 면에서 크게 뒤지지 않으면서도 CGI 프로그램으로 사용할 수 있는 스크립트 (많은 경우 속도만이 유일한 기준이기는 하지만, 둘 다 C에 비해서는 떨어집니다). 훌륭하고 깔끔하며 빠른 셸 스크립트에는 여러 가지 요소가 있습니다.

- 가장 중요한 기준은 명확하고 가독성 있는 레이아웃이어야 합니다.
- 두 번째는 불필요한 명령어를 피하는 것입니다.

명확한 레이아웃은 "흑마법"처럼 보이는 셸 스크립트와 쉽게 유지 관리하고 이해할 수 있는 스크립트의 차이를 만듭니다.
간단한 스크립트라면 큰 문제가 되지 않는다고 생각할 수도 있지만, 다음 두 가지 사항을 염두에 두어야 합니다.

1. 첫째, 간단한 스크립트는 예상보다 더 자주 크고 복잡한 스크립트로 커질 수 있습니다.
2. 둘째, 스크립트가 어떻게 작동하는지 아무도 이해하지 못하면 남은 기간 동안 남은 평생 동안 직접 유지 관리해야 합니다!

셸 스크립트는 특히 들여쓰기가 잘 안 되는 경우가 많은데, 주요 제어 구조가 if/then/else와 루프이기 때문에 스크립트의 동작을 이해하려면 들여쓰기가 매우 중요합니다.

많은 셸 스크립트의 주요 약점 중 하나는 다음과 같은 줄입니다:

```
cat /tmp/myfile | grep "mystring"
```

로 훨씬 빠르게 실행됩니다:

```
grep "mystring" /tmp/myfile
```

제 시스템에서는 75600바이트의 비교적 작은 /bin/grep 실행 파일을 로드하고, 전송을 위해 메모리에서 파이프를 열고, 이보다 더 작은 9528바이트의 /bin/cat 실행 파일을 로드하여 실행하고, 파이프의 입력에 연결하여 실행해야 합니다.

물론 이런 종류의 작업은 OS가 존재하는 이유이며 일반적으로 매우 효율적으로 수행됩니다. 하지만 이 명령이 여러 번 반복 실행되는 경우, 특히 스크립트 자체에 큰 문제가 없으면서도 다른 요인으로 인해 속도가 느려지는 CGI 환경에서는 cat 실행 파일을 찾아서 로드하고 파이프를 설정했다가 해제하지 않아도 되기 때문에 약간의 차이를 만들 수 있습니다. 일부 유니티는 프로세스를 로드하고 실행한 후 다시 지우는 이른바 '프로세스 구축 및 해체'에 다른 유니티보다 더 효율적입니다. 하지만 이 작업을 아무리 잘 수행하는 유닉스라고 해도 이 작업을 전혀 수행하지 않는 것이 좋습니다.

그 결과, 일부 커뮤니티에서는 '심각한 셸 스크립트에서 cat이라는 단어를 가장 무분별하게 사용한 사람에게 주는 상 (The Award For The Most Gratuitous Use Of The Word Cat In A Serious Shell Script)'으로도 알려진 Useless Use of Cat Award (UUoC)에 대한 언급이 comp.unix.shell 뉴스 그룹에서 가끔씩 언급되는 것을 볼 수 있습니다. 이것은 순전히 동료들이 서로를 견제하고 일이 제대로 이루어지도록 하기 위한 방법입니다.

이제 다른 이야기로 넘어가죠: 셸 스크립트는 그 특성상 소스를 닫을 수 없으므로 너무 가깝게 느끼지 마세요. 고객에게 셸 스크립트를 제공하면 고객은 이를 아주 쉽게 검사할 수 있습니다. 따라서 셸 스크립트를 넘겨주는 사람이 누구든 검사할 수 있다는 것을 받아들이는 것이 좋으며, GPL2를 활용하여 사람들이 무료로 피드백과 버그 수정을 제공하도록 장려하세요!

3. A First Script

파일을 실행 가능한 파일로 만들려면 실행 가능 비트를 설정해야 하며, 셸 스크립트의 경우 읽기 가능 비트도 설정해야 합니다:

```
$ chmod a+rx first.sh


$ ./first.sh
```

첫 번째 셸 스크립트로는 "Hello World"라는 스크립트를 작성하겠습니다. 그런 다음 지금까지 읽은 그 어떤 튜토리얼보다 헬로 월드 프로그램에서 더 많은 것을 얻으려고 노력할 것입니다 :-).

다음과 같이 파일 (first.sh)을 만듭니다:

```
#!/bin/sh
# This is a comment!
echo Hello World # This is a comment, too!
```

첫 번째 줄은 파일이 /bin/sh에서 실행되도록 Unix에 지시합니다. 이 위치는 거의 모든 유닉스 시스템에서 본 셸의 표준 위치입니다. GNU/Linux를 사용하는 경우 /bin/sh는 일반적으로 bash(또는 최근에는 dash)에 대한 심볼릭 링크입니다.

두 번째 줄은 특수 기호로 시작합니다: #. 이 줄은 주석으로 표시되며 셸에서 완전히 무시됩니다. 유일한 예외는 파일의 첫 번째 줄이 #!로 시작하는 경우입니다. 이것은 유닉스에서 특별히 취급하는 특수 지시어입니다. 즉, 대화형 셸로 csh, ksh 등을 사용하더라도 그 뒤에 오는 내용은 본 셸이 해석해야 한다는 뜻입니다. 마찬가지로, Perl 스크립트는 인터랙티브 셸에 #!/usr/bin/perl 줄로 시작하여 뒤에 오는 프로그램이 Perl로 실행되어야 한다는 의미입니다. 본 셸 프로그래밍의 경우 #!/bin/sh를 사용합니다.

세 번째 줄은 두 개의 매개 변수 또는 인수가 있는 echo 명령을 실행합니다. 첫 번째 줄은 "Hello", 두 번째 줄은 "World"입니다.
echo는 매개변수 사이에 자동으로 공백을 하나씩 넣는다는 점에 유의하세요.

기호는 여전히 주석을 표시하며, #과 그 뒤에 오는 모든 것은 셸에서 무시됩니다.

이제 chmod 755 first.sh를 실행하여 텍스트 파일을 실행 가능하게 만들고 ./first.sh를 실행합니다. 그러면 화면이 다음과 같이 표시됩니다:

```
$ chmod 755 first.sh $ ./first.sh
Hello World
$
```

아마 예상하셨을 겁니다 ! 그냥 실행할 수도 있습니다 :

```
$ echo Hello World
Hello World
$
```

이제 몇 가지를 변경해 보겠습니다 .
먼저 , echo는 매개 변수 사이에 공백을 하나 넣습니다 . "Hello"와 "World" 사이에 공백을 몇 개 넣습니다 .
어떤 결과가 나올까요 ? 그 사이에 탭 문자를 넣으면 어떨까요 ?
셸 프로그래밍을 할 때 항상 그렇듯이 직접 시도해 보세요 .
결과는 똑같습니다 ! 우리는 두 개의 인수를 사용하여 echo 프로그램을 호출하고 있으며 , 이 프로그램은 두 인수 사이의 간격에 대해서는 cp와 마찬가지로 신경 쓰지 않습니다 . 이제 코드를 다시 수정합니다 :

```
#!/bin/sh
# This is a comment!
echo "Hello   World"      # This is a comment, too!
```

이번에는 작동합니다 . 다른 프로그래밍 언어에 대한 경험이 있다면 이 역시 예상하셨을 것입니다 . 하지만 더 복잡한 명령어와 셸 스크립트에서 무슨 일이 일어나고 있는지 이해하기 위해서는 이를 이해하고 설명할 수 있어야 합니다 : 왜 ?
이제 **echo**는 단 하나의 인자 , 즉 "Hello World"라는 문자열로 호출되었습니다 . 이 문자열을 정확하게 출력합니다 .
여기서 이해해야 할 점은 셸이 호출되는 프로그램에 인수를 전달하기 전에 인수를 구문 분석한다는 것입니다 .
이 경우 따옴표는 제거하지만 문자열은 하나의 인수로 전달합니다 .
마지막 예로 다음 스크립트를 입력해 보세요 . 실행하기 전에 결과를 예측해 보세요 :

```
#!/bin/sh
# This is a comment!
echo "Hello   World"      # This is a comment, too!
echo "Hello World"
echo "Hello * World"
echo Hello * World
echo Hello   World
echo "Hello" World
echo Hello "   " World
echo "Hello \"*\" World"
echo `hello` world
echo 'hello' world
```

모든 것이 예상대로 되었나요 ? 그렇지 않더라도 걱정하지 마세요 ! 이 튜토리얼에서 다룰 내용 중 일부에 불과하며 .... 예 , echo보다 더 강력한 명령을 사용하게 될 것입니다 !

4. Variables - Part I

현존하는 거의 모든 프로그래밍 언어에는 값을 할당하고 그 내용을 읽고 조작할 수 있는 메모리 덩어리의 상징적인 이름인 변수라는 개념이 있습니다. 본 셸(Bourne shell)도 예외는 아니며, 이 섹션에서는 그 개념을 소개합니다. 환경에 의해 설정되는 변수를 살펴보는 변수 - 2부에서는 이에 대해 더 자세히 설명합니다.

첫 번째 Hello World 예제를 다시 살펴봅시다. 변수를 사용하여 이 작업을 수행할 수 있습니다(너무 간단한 예제라서 변수가 필요하지는 않지만!).

"=" 기호 주위에 공백이 없어야 한다는 점에 유의하세요: **VAR=**값은 작동하고, **VAR =** 값은 작동하지 않습니다. 첫 번째 경우 셸은 "=" 기호를 보고 명령을 변수 할당으로 처리합니다. 두 번째 경우 셸은 VAR이 명령의 이름이어야 한다고 가정하고 실행을 시도합니다.

첫 번째 인수가 "="이고 두 번째 인수가 "값"인 VAR 명령을 어떻게 실행하라고 지시할 수 있을까요?

var1.sh에 다음 코드를 입력합니다:

```
#!/bin/sh
MY_MESSAGE="Hello World"
echo $MY_MESSAGE
```

이렇게 하면 문자열 "Hello World"가 **MY_MESSAGE** 변수에 할당된 다음 변수 값이 에코됩니다.

Hello World 문자열 주위를 따옴표로 묶어야 한다는 점에 유의하세요. echo는 매개 변수를 얼마든지 사용할 수 있기 때문에 Hello World를 에코할 수 있지만, 변수는 하나의 값만 저장할 수 있으므로 공백이 있는 문자열은 셸이 모두 하나로 취급하도록 따옴표로 묶어야 합니다. 그렇지 않으면 셸은 **MY_MESSAGE=Hello**를 지정한 후 World 명령을 실행하려고 시도합니다.

셸은 변수의 유형을 신경 쓰지 않으며 문자열, 정수, 실수 등 원하는 모든 것을 저장할 수 있습니다. Perl에 익숙한 사람이라면 이 점이 상당히 만족스러울 수 있지만, C나 파스칼, 심지어 Ada에 익숙하지 않은 사람이라면 이 점이 상당히 이상하게 느껴질 수 있습니다. 실제로는 모두 문자열로 저장되지만 숫자를 기대하는 루틴은 이를 숫자로 취급할 수 있습니다. 변수에 문자열을 할당하고 거기에 1을 더하려고 하면 실패합니다:

```
$ x="hello"
$ expr $x + 1
expr: non-numeric argument
$
```

이는 외부 프로그램 expr이 숫자만 기대하기 때문입니다. 그러나 둘 사이에는 구문상의 차이가 없습니다:

```
MY_MESSAGE="Hello World"
MY_SHORT_MESSAGE=hi
MY_NUMBER=1
MY_PI=3.142
```

```
MY_OTHER_PI="3.142"
MY_MIXED=123abc
```

특수 문자는 셸에서 해석되지 않도록 적절하게 이스케이프 처리해야 합니다. 이에 대해서는 6장 "이스케이프 문자"에서 자세히 설명합니다.

다음 스크립트는 이름을 물어본 다음 직접 인사하는 방식으로 변수 이름을 대화형으로 설정할 수 있습니다:

```
#!/bin/sh
echo What is your name?
read MY_NAME
echo "Hello $MY_NAME - hope you're well."
```

마리오 바친스키가 친절하게도 제가 3행에서 큰따옴표를 놓쳐서 "you're"라는 단어의 작은따옴표가 일치하지 않아서 오류가 발생했다고 지적해 주었습니다. 셸 프로그래머를 미치게 만드는 이런 종류의 오류를 조심하세요!

이것은 표준 입력에서 제공된 변수로 한 줄을 읽는 셸 내장 명령 read를 사용하는 것입니다. 전체 이름을 입력하고 echo 명령 주위에 큰따옴표를 사용하지 않더라도 올바르게 출력된다는 점에 유의하세요. 어떻게 하나요? 앞서 **MY_MESSAGE** 변수를 설정하기 위해 변수 주위를 큰따옴표로 묶어야 했습니다.

이제 read 명령은 입력 주위에 자동으로 따옴표를 넣어 공백이 올바르게 처리되도록 합니다. 물론 출력도 따옴표로 묶어야 합니다 (예: echo "$MY_MESSAGE").

변수의 범위

본 셸의 변수는 C와 같은 언어에서처럼 선언할 필요가 없습니다. 선언되지 않은 변수를 읽으려고 하면 결과는 빈 문자열입니다. 경고나 오류가 발생합니다. 이로 인해 몇 가지 미묘한 버그가 발생할 수 있습니다 - 만약 다음과 같이

```
MY_OBFUSCATED_VARIABLE=Hello
```

할당하고

```
echo $MY_OSFUCATED_VARIABLE
```

그러면 아무것도 표시되지 않습니다 (두 번째 OBFUSCATED의 철자가 틀렸기 때문입니다).

변수의 범위에 근본적인 영향을 미치는 export라는 명령이 있습니다. 변수에 어떤 일이 일어나는지 제대로 알기 위해서는 이 명령이 어떻게 사용되는지 이해해야 합니다.

작은 셸 스크립트인 myvar2.sh를 만듭니다:

```
#!/bin/sh
echo "MYVAR is: $MYVAR"
MYVAR="hi there"
echo "MYVAR is: $MYVAR"
```

이제 스크립트를 실행합니다 :

```
$ ./myvar2.sh
MYVAR is:
MYVAR is: hi there
```

MYVAR에 어떤 값도 설정되지 않았으므로 비어 있습니다 . 그런 다음 값을 지정하면 예상한 결과가 나옵니다 .
이제 실행합니다 :

```
$ MYVAR=hello
$ ./myvar2.sh
MYVAR is:
MYVAR is: hi there
```

아직 설정되지 않았습니다 ! 무슨 일이죠 ?
대화형 셸에서 myvar2.sh를 호출하면 스크립트를 실행하기 위한 새 셸이 생성됩니다 . 이는 앞서 설명한
스크립트 시작 부분의 #!/bin/sh 줄 때문입니다 .
셸 스크립트를 포함한 다른 프로그램에서 변수를 상속하려면 변수를 내보내야 합니다 . 다음을 입력합니다 :

```
$ export MYVAR
$ ./myvar2.sh
MYVAR is: hello
MYVAR is: hi there
```

이제 스크립트 3줄을 보세요 . MYVAR의 값을 변경하고 있습니다 . 하지만 이것이 대화형 셸에 다시 전달될
방법은 없습니다 . MYVAR의 값을 읽어보세요 :

```
$ echo $MYVAR
hello
$
```

셸 스크립트가 종료되면 해당 환경은 파괴됩니다 . 하지만 MYVAR은 대화형 셸 내에서 hello 값을 유지합니다 .
스크립트에서 환경 변경 사항을 다시 받으려면 스크립트를 소싱해야 하는데 , 이렇게 하면 스크립트를
실행하기 위해 다른 셸을 생성하는 대신 자체 대화형 셸 내에서 스크립트를 효과적으로 실행할 수 있습니다 .
"."(점) 명령을 통해 스크립트를 소스화할 수 있습니다 :

```
$ MYVAR=hello
$ echo $MYVAR
hello
$ . ./myvar2.sh
MYVAR is: hello
MYVAR is: hi there
$ echo $MYVAR
hi there
```

이제 변경 사항이 다시 셸에 적용되었습니다! 예를 들어 .profile 또는 .bash_profile 파일은 이렇게 작동합니다.

이 경우 MYVAR을 내보낼 필요가 없다는 점에 유의하세요.

위에서 echo MYVAR라고 했지, $MYVAR을 echo하라고 하지 않았다는 점을 지적해 주신 sway님께 감사드립니다. 셸 스크립트에서 저지르기 쉬운 실수의 또 다른 예입니다. 이 시점에서 변수에 대해 한 가지 더 언급할 가치가 있는 것은 다음 셸 스크립트를 고려하는 것입니다:

```
#!/bin/sh
echo "What is your name?"
read USER_NAME
echo "Hello $USER_NAME"
echo "I will create you a file called $USER_NAME_file"
touch $USER_NAME_file
```

어떤 결과를 기대할 수 있는지 생각해 보세요. 예를 들어 **USER_NAME**에 "steve"를 입력하면 스크립트에서 steve_file을 만들어야 할까요?

사실, 아니죠. USER_NAME_file이라는 변수가 없으면 오류가 발생합니다. 셸은 변수가 어디에서 끝나고 나머지가 시작되는지 알지 못합니다. 이를 어떻게 정의할 수 있을까요? 답은 변수 자체를 중괄호({})로 묶는 것입니다:

```
#!/bin/sh
echo "What is your name?"
read USER_NAME
echo "Hello $USER_NAME"
echo "I will create you a file called ${USER_NAME}_file"
touch "${USER_NAME}_file"
```

이제 셸은 우리가 **USER_NAME** 변수를 참조하고 있으며 이 변수에 "_file"이라는 접미사를 붙이기를 원한다는 것을 알고 있습니다. 문제의 원인을 추적하기 어려울 수 있기 때문에 많은 초보 셸 스크립트 프로그래머가 이런 오류를 범할 수 있습니다.

또한 **"${USER_NAME}_file"** 주위의 따옴표에 주목하세요. 사용자가 "Steve Parker"(공백에 유의)를 입력한 경우 따옴표가 없으면 touch에 전달되는 인수는 Steve와 Parker_file이 됩니다. 즉, 터치할 파일이 하나가 아니라 두 개인 Steve Parker_file을 터치하라는 뜻이 됩니다. 따옴표는 이를 방지합니다. 이 점을

강조해준 Chris에게 감사드립니다 .

5. 와일드카드 (Wildcards)

와일드카드는 이전에 유닉스를 사용해 본 적이 있다면 전혀 새로운 것이 아닙니다.

하지만 셸 스크립트에서 와일드카드가 어떻게 유용한지는 분명하지 않습니다. 이 섹션은 셸 스크립트에서 다양한 구문을 사용할 때 어떤 효과가 나타날지 예측하는 등, 오래된 회색 세포가 어떻게 보이는지 생각해보도록 하기 위한 것입니다. 이것은 나중에 특히 루프 섹션에서 사용될 것입니다.

먼저 /tmp/a의 모든 파일을 /tmp/b로 어떻게 복사할지 생각해 보세요. 모든 .txt 파일? 모든 .html 파일? 여러분이 생각해내셨기를 바랍니다:

```
$ cp /tmp/a/* /tmp/b/
$ cp /tmp/a/*.txt /tmp/b/
$ cp /tmp/a/*.html /tmp/b/
```

이제 ls /tmp/a/를 사용하지 않고 /tmp/a/에 있는 파일을 나열하려면 어떻게 해야 할까요? echo /tmp/a/*는 어떨까요? 이것과 ls 출력의 두 가지 주요 차이점은 무엇일까요? 이것이 어떻게 유용할까요? 아니면 방해가 될까요?

모든 .txt 파일의 이름을 어떻게 .bak으로 바꿀 수 있을까요? 다음 사항에 유의하세요.

```
$ mv *.txt *.bak
```

를 사용하면 원하는 효과를 얻을 수 없으므로, 셸에서 어떻게 확장되어 mv로 전달되는지 생각해 보세요. 도움이 된다면 mv 대신 echo를 사용하여 시도해 보세요. 아직 다루지 않은 몇 가지 개념을 사용하기 때문에 나중에 더 자세히 살펴보겠습니다.

6. 이스케이프 문자

예를 들어 큰따옴표(") 문자를 사용하면 공백 및 탭 문자가 처리되는 방식에 영향을 주는 등 특정 문자는 셸에 중요한 역할을 합니다:

```
$ echo Hello        World
Hello World
$ echo "Hello        World"
Hello        World
```

그렇다면 Hello "World"라고 출력하려면 어떻게 해야할까요?

```
$ echo "Hello    \"World\""
```

첫 번째와 마지막 " 문자는 두 단어 사이의 간격이 그대로 유지되도록 echo에 전달된 하나의 매개 변수로 전체를 래핑합니다. 하지만 코드:

```
$ echo "Hello    " World ""
```

는 세 개의 매개 변수로 해석됩니다:

- "Hello   "
- World
- ""

따라서 출력은 다음과 같습니다.

```
Hello World
```

따옴표가 완전히 손실된다는 점에 유의하세요. 첫 번째와 두 번째 따옴표는 Hello와 그 뒤에 오는 공백을 표시하고, 두 번째 인수는 따옴표로 묶이지 않은 "World"이고, 세 번째 인수는 빈 문자열("")이기 때문입니다.

이 점을 지적해 주신 Patrick에게 감사드립니다:

```
$ echo "Hello    "World""
```

가 실제로 하나의 매개변수(따옴표로 묶인 매개변수 사이에 공백이 없음)이며, 다음과 같이 echo 명령을 (예를 들어) ls로 대체하여 이를 테스트할 수 있습니다.

대부분의 문자(*, ', 등)는 큰따옴표("") 안에 넣어도 해석되지 않습니다(즉, 문자 그대로 받아들여집니다). 이러한 문자는 있는 그대로 호출되는 명령에 전달됩니다. 별표(*)를 사용한 예는 다음과 같습니다:

```
$ echo *
case.shtml escape.shtml first.shtml
functions.shtml hints.shtml index.shtml
ip-primer.txt raid1+0.txt
$ echo *txt
ip-primer.txt raid1+0.txt
$ echo "*"
*
$ echo "*txt"
*txt
```

첫 번째 예에서 *는 확장되어 현재 디렉터리에 있는 모든 파일을 의미합니다. 두 번째 예에서 *txt는 txt로 끝나는 모든 파일을 의미합니다. 세 번째 예에서는 *를 큰따옴표로 묶어 문자 그대로 해석합니다. 네 번째 예에서는 동일하게 적용되지만 문자열에 txt를 추가했습니다.

그러나 ", $, ` 및 \는 큰따옴표 안에 있더라도 셸에서 해석됩니다. 백슬래시 (\) 문자는 이러한 특수 문자를 표시하는 데 사용되므로 셸에서 해석되지 않고 실행 중인 명령 (예: echo)에 전달됩니다. 따라서 문자열을 다음과 같이 출력하려면: ($X의 값이 5라고 가정)

> A quote is ", backslash is \, backtick is `.
> A few spaces are and dollar is $. $X is 5.

다음과 같이 입력해야 합니다:

```
$ echo "A quote is \", backslash is \\, backtick is \`."
A quote is ", backslash is \, backtick is `.
$ echo "A few spaces are    ; dollar is \$. \$X is ${X}."
A few spaces are    ; dollar is $. $X is 5.
```

" 가 공백을 유지하는 데 특별한 이유를 살펴보았습니다. 달러 ($)는 변수를 표시하기 때문에 특별한데, $X는 변수 X의 내용으로 셸을 대체합니다. 백슬래시 (\)는 그 자체로 다른 문자를 표시하는 데 사용되기 때문에 특별한데, 완전한 셸을 위해서는 다음 옵션이 필요합니다:

```
$ echo "This is \\ a backslash"
This is \ a backslash
$ echo "This is \" a quote and this is \\ a backslash"
This is " a quote and this is \ a backslash
```

따라서 백슬래시 자체를 이스케이프 처리하여 문자 그대로 받아들여야 함을 표시해야 합니다. 다른 특수 문자, 백틱에 대해서는 12장 "외부 프로그램"의 뒷부분에서 설명합니다.

7. 루프

대부분의 언어에는 루프라는 개념이 있습니다 : 어떤 작업을 20번 반복하려면 매번 약간의 변경을 가하면서 코드를 20번 입력할 필요는 없습니다 . 그 결과 본 셸에는 for 루프와 while 루프가 있습니다 . 다른 언어에 비해 다소 적은 기능이지만 , 셸 프로그래밍이 C의 힘을 가지고 있다고 주장하는 사람은 아무도 없습니다 .

For 루프

"for" 루프는 목록이 모두 소진될 때까지 값 집합을 반복합니다 :

```
#!/bin/sh
for i in 1 2 3 4 5
do
  echo "Looping ... number $i"
done
```

이 코드를 사용해 어떤 기능을 하는지 확인해 보세요 . 값은 무엇이든 될 수 있다는 점에 유의하세요 :

```
#!/bin/sh
for i in hello 1 * 2 goodbye
do
  echo "Looping ... i is set to $i"
done
```

시도해 볼 만한 가치가 있습니다 . 여기서 무슨 일이 일어나고 있는지 이해했는지 확인하세요 . "*" 없이 시도해 보고 아이디어를 파악한 다음 와일드카드 섹션을 다시 읽고 "*"를 제자리에 두고 다시 시도해 보세요 . 다른 디렉토리에서도 "*"를 큰따옴표로 둘러싸고 그 앞에 백슬래시 (*)를 넣어 시도해 보세요 .

현재 셸에 액세스할 수 없는 경우 (이 튜토리얼을 읽는 동안 셸이 있으면 매우 유용합니다), 위의 두 스크립트의 결과는 다음과 같습니다 :

```
Looping .... number 1
Looping .... number 2
Looping .... number 3
Looping .... number 4
Looping .... number 5
```

그리고 두 번째 예시에서는

```
Looping ... i is set to hello
Looping ... i is set to 1
Looping ... i is set to (name of first file in current directory)
... etc ...
Looping ... i is set to (name of last file in current directory) Looping ... i is set to 2
Looping ... i is set to goodbye
```

보시다시피, 주어진 입력이 무엇이든 입력이 다 떨어질 때까지 단순히 반복합니다.

While 루프

"while" 루프가 훨씬 더 재미있을 수 있습니다! (재미에 대한 여러분의 생각과 얼마나 자주 집을 나가느냐에 따라 다르겠지만 ...)

```
#!/bin/sh
INPUT_STRING=hello
while [ "$INPUT_STRING" != "bye" ]
do
  echo "Please type something in (bye to quit)"
  read INPUT_STRING
  echo "You typed: $INPUT_STRING"
done
```

여기서 일어나는 일은 메시지가 표시될 때 **"bye"**를 입력할 때까지 에코 및 읽기 문이 무한정 실행된다는 것입니다. 테스트하기 전에 '변수 - 1부 '(4장)를 검토하여 **INPUT_STRING=hello**를 설정한 이유를 알아보세요. 이렇게 하면 기존의 동안 루프가 아닌 반복 루프가 됩니다.

콜론 (:)은 항상 참으로 평가됩니다. 콜론을 사용하는 것이 필요할 때도 있지만, 실제 종료 조건을 사용하는 것이 더 바람직할 때가 많습니다. 위의 루프를 종료하는 것과 아래의 루프를 종료하는 것을 비교하여 어느 것이 더 우아한지 살펴보세요. 또한 각각이 다른 것보다 더 유용할 수 있는 몇 가지 상황을 생각해 보세요:

```
#!/bin/sh
while :
do
  echo "Please type something in (^C to quit)"
  read INPUT_STRING
  echo "You typed: $INPUT_STRING"
done
```

또 다른 유용한 트릭은 **"while read"** 루프입니다. 이 예제에서는 나중에 다룰 case 문을 사용합니다. 이 문은 파일 에서 읽고 각 줄에 대해 어떤 언어가 사용되고 있다고 생각하는지 알려줍니다.

(참고: 각 줄은 LF(개행)로 끝나야 합니다. cat myfile.txt가 빈 줄로 끝나지 않으면 마지막 줄은 처리되지 않습니다.).

이렇게 하면 "myfile.txt" 파일을 한 번에 한 줄씩 "$input_text" 변수로 읽습니다. 그런 다음 case 문은 $input_text의 값을 확인합니다. 읽은 단어가 "hello"인 경우 "English"라는 단어를 에코합니다. "gday"였다면 "Australian"가 echo로 출력됩니다. myfile.txt의 한 줄에서 읽은 단어가 제공된 패턴 중 어느 것과도 일치하지 않으면 포괄적인 "*" 기본값으로 "Unknown Language: $input_text"라는 메시지가 표시됩니다. 여기서 "$input_text"는 물론 myfile.txt에서 읽은 줄의 값입니다.

```
#!/bin/sh
while read input_text
do
  case $input_text in
        hello)              echo English    ;;
        howdy)              echo American   ;;
        gday)               echo Australian ;;
        bonjour)            echo French     ;;
        "guten tag")        echo German
        *)                  echo Unknown Language: $input_text ;;
   esac
done < myfile.txt
```

"myfile.txt" 파일에 다음 다섯 줄이 포함되어 있다고 가정해 보겠습니다:

```
this file is called myfile.txt. It is an example text file.
hello
gday
bonjour
hola
```

이 스크립트를 샘플로 실행하면 다음과 같습니다:

```
Unknown Language: this file is called myfile.txt. It is an example text file.
English
Australian
French
Unknown Language: hola
```

리눅스 프롬 스크래치 13 프로젝트에서 배운 편리한 Bash(본 셸은 아님) 팁은 다음과 같습니다:

```
mkdir rc{0,1,2,3,4,5,6,S}.d
```

대신에 더 번거로운 방법을 사용합니다:

```
for runlevel in 0 1 2 3 4 5 6 S
do
  mkdir rc${runlevel}.d
done
```

이 작업도 재귀적으로 수행할 수 있습니다 :

```
$ cd /
$ ls -ld {,usr,usr/local}/{bin,sbin,lib}
drwxr-xr-x   2 root    root    4096 Oct 26 01:00 /bin
drwxr-xr-x   6 root    root    4096 Jan 16 17:09 /lib
drwxr-xr-x   2 root    root    4096 Oct 27 00:02 /sbin
drwxr-xr-x   2 root    root   40960 Jan 16 19:35 usr/bin
drwxr-xr-x  83 root     root   49152 Jan 16 17:23 usr/lib
drwxr-xr-x   2 root    root    4096 Jan 16 22:22 usr/local/bin
drwxr-xr-x   3 root    root    4096 Jan 16 19:17 usr/local/lib
drwxr-xr-x   2 root    root    4096 Dec 28 00:44 usr/local/sbin
drwxr-xr-x   2 root    root    8192 Dec 27 02:10 usr/sbin
```

Test 및 Case 장에서 while 루프에 대해 자세히 살펴보겠습니다 .

8. Test

test는 거의 모든 셸 스크립트에서 사용됩니다 . 테스트가 직접 호출되는 경우가 많지 않기 때문에 그렇게 보이지 않을 수도 있습니다 . test는 []로 더 자주 호출됩니다 . []는 셸 프로그램을 더 읽기 쉽게 만들기 위해 테스트에 대한 기호적 링크입니다 . 또한 일반적으로 셸에 내장되어 있습니다 (즉 , Unix 환경이 다르게 설정되어 있더라도 셸 자체에서 [를 테스트의 의미로 해석합니다):

```
$ type [
[ is a shell builtin
$ which [
/usr/bin/[
$ ls -l /usr/bin/[
lrwxrwxrwx 1 root root 4 Mar 27 2000 /usr/bin/[ -> test
```

즉 , '['는 실제로는 ls 및 다른 프로그램과 마찬가지로 프로그램이므로 공백으로 둘러싸여 있어야 합니다 :

```
if [$foo = "bar" ]
```

는 작동하지 않으며 , 시작 '['가 없는 ']'인 test$foo = "bar"]처럼 해석됩니다 . 모든 연산자 주위에 공백을 넣으세요 . 필수 공백을 'SPACE'라는 단어로 강조 표시했는데 , 공백이 없으면 작동하지 않으므로 'SPACE'를 실제 공백으로 바꾸세요 :

```
if SPACE [ SPACE "$foo" SPACE = SPACE "bar" SPACE ]
```

참고 : 일부 셸은 문자열 비교에 "=="도 허용하지만 , 이는 이식성이 없으므로 문자열에는 단일 "="를 사용하거나 정수의 경우 "-eq"를 사용해야 합니다 .

test는 간단하지만 강력한 비교 유틸리티입니다 . 자세한 내용은 시스템에서 "man test"를 실행하세요 . 하지만 여기서는 몇 가지 사용법과 일반적인 예제를 소개합니다 .

test는 if 및 while 문을 통해 간접적으로 호출되는 경우가 가장 많습니다 . test라는 프로그램을 만들어서 실행하려고 하면 프로그램 대신 이 셸 내장 함수가 호출되기 때문에 어려움을 겪게 되는 이유이기도 합니다 ! if...then...else...의 구문은 다음과 같습니다 :

```
if [ ... ] then
  # if-code
else
  # else-code
fi
```

fi는 거꾸로 된 경우라는 점에 유의하세요! 나중에 대소문자 및 esac과 함께 다시 사용됩니다. 또한 구문에 유의하세요. "if [...]"와 "then" 명령은 서로 다른 줄에 있어야 합니다. 또는 세미콜론 ";"으로 구분할 수도 있습니다:

```
if [ ... ]; then
  # do something
fi
```

다음과 같이 elif를 사용할 수도 있습니다:

```
if  [ something ]; then
 echo "Something"
 elif [ something_else ]; then
   echo "Something else"
 else
   echo "None of the above"
fi
```

[something] 테스트가 성공하면 "Something"을 에코하고, 그렇지 않으면 [something_else]를 테스트합니다.
[something_else]를 테스트하고 성공하면 “Something else"를 에코합니다. 다른 모든 테스트가 실패하면 "None of the above"를 에코합니다.

다음 코드 스니펫을 실행하기 전에 변수 X를 다양한 값으로 설정해 보세요 (-1, 0, 1, hello, bye 등 시도해 보세요). 다음과 같이 하면 됩니다 (변수 - 1부에서 변수를 내보내야 한다는 점을 지적해 주신 Dave에게 감사드립니다):

```
$ X=5
$ export X
$ ./test.sh
  ... output of test.sh ...
$ X=hello
$ ./test.sh
  ... output of test.sh ...
$ X=test.sh
$ ./test.sh
  ... output of test.sh ...
```

그런 다음 $X를 기존 파일 이름 (예: /etc/hosts)으로 사용하여 다시 시도합니다.

```
#!/bin/sh
if [ "$X" -lt "0" ]
```

```
then
  echo "X is less than zero"
fi
if [ "$X" -gt "0" ]; then
  echo "X is more than zero"
fi
[ "$X" -le "0" ] && \
      echo "X is less than or equal to  zero"
[ "$X" -ge "0" ] && \
      echo "X is more than or equal to zero"
[ "$X" = "0" ] && \
      echo "X is the string or number \"0\""
[ "$X" = "hello" ] && \
      echo "X matches the string \"hello\""
[ "$X" != "hello" ] && \
      echo "X is not the string \"hello\""
[ -n "$X" ] && \
      echo "X is of nonzero length"
[ -f "$X" ] && \
      echo "X is the path of a real file" || \
      echo "No such file: $X"
[ -x "$X" ] && \
      echo "X is the path of an executable file"
[ "$X" -nt "/etc/passwd" ] && \
echo "X is a file which is newer than /etc/passwd"
```

세미콜론 (;)을 사용하여 두 줄을 연결할 수 있다는 점에 유의하세요. 이는 간단한 if 문에서 약간의 공간을 절약하기 위해 종종 수행됩니다. 백슬래시는 단순히 셸에 이것이 줄의 끝이 아니라 두 줄 (또는 그 이상)을 하나로 취급해야 함을 알려줍니다. 이는 가독성을 높이는 데 유용합니다. 다음 줄은 들여쓰는 것이 일반적입니다.

이 예제에서 볼 수 있듯이 test는 숫자, 문자열 및 파일 이름에 대해 많은 테스트를 수행할 수 있습니다.

-a, -e(둘 다 "파일이 존재함"을 의미), -S(파일이 소켓), -nt(파일이 보다 최신 파일), -ot(파일이 보다 오래된 파일), -ef(경로가 동일한 파일을 참조) 및 -O(파일이 내 사용자 소유)는 기존 본 셸 (예: Solaris, AIX, HPUX 등의 /bin/sh)에서 사용할 수 없다는 점을 지적해 주신 Aaron에게 감사드립니다.

if 문을 작성하는 더 간단한 방법이 있습니다: && 및 || 명령은 각각 결과가 참 또는 거짓일 때 실행할 코드를 지정합니다.

```
#!/bin/sh
[ $X -ne 0 ] && echo "X isn't zero" || echo "X is zero"
```

```
[ -f  $X ] && echo "X is a file" || echo "X is not a file"
[ -n  $X ] && echo "X is of non-zero length" || \
      echo "X is of zero length"
```

이 구문이 가능한 이유는 [라는 파일 (또는 셸 빌트인)이 있기 때문이며 , 이 파일은 test에 연결됩니다 . 하지만 이 구문을 과도하게 사용하면 읽기 어려운 코드가 될 수 있으므로 주의하세요 . if...then...else... 구조가 훨씬 더 가독성이 높습니다 . 독자의 주의를 지나치게 분산시키지 않으려는 동안 루프와 사소한 상태 점검에는 [...] 구조를 사용하는 것이 좋습니다 .

X를 숫자가 아닌 값으로 설정하면 처음 몇 번 비교하면 메시지가 표시됩니다 :

```
test.sh: [: integer expression expected before -lt
test.sh: [: integer expression expected before -gt
test.sh: [: integer expression expected before -le
test.sh: [: integer expression expected before -ge
```

이는 -lt, -gt, -le 및 -ge 비교가 정수 전용으로 설계되어 문자열에서는 작동하지 않기 때문입니다 . !=와 같은 문자열 비교는 "5"를 기꺼이 문자열로 취급하지만 "Hello"를 정수로 취급하는 합리적인 방법이 없으므로 정수 비교가 불만을 제기합니다 . 셸 스크립트가 더 우아하게 작동하도록 하려면 테스트하기 전에 다음과 같이 변수의 내용을 확인해야 합니다 :

```
echo -en "Please guess the magic number: "
read X
echo $X | grep "[^0-9]" > /dev/null 2>&1
if [ "$?" -eq "0" ]; then
  # If the grep found something other than 0-9
  # then it's not an integer.
  echo "Sorry, wanted a number"
else
  # The grep found only 0-9, so it's an integer.
  # We can safely do a test on it.
  if [ "$X" = "7" ]; then
    echo "You entered the magic number!"
  fi
fi
```

이렇게 하면 사용자에게 더 의미 있는 메시지를 전달하고 우아하게 종료할 수 있습니다 . 변수에 대해서는 '변수 - 2부 '(10장)에서 설명했으며 , grep은 복잡한 변수이므로 여기서는 숫자 (0~9)와 다른 문자가 포함된 텍스트 줄을 찾으므로 grep [^0~9]의 캐럿 (^)은 숫자로만 구성되지 않은 줄만 찾습니다 . 그러면 그 반대의 경우 (성공이 아닌 실패에 따라 작동하는 경우)를 취할 수 있습니다 . 알겠죠 ? >/dev/ null 2>&1은 모든 출력이나 오류를 사용자 화면으로 보내지 않고 특수한 "null" 장치로 보냅니다 . 이 페이지에서는 grep -v [0-9]가 작동한다고 주장했지만 이는 너무 단순합니다 .

다음과 같이 while 루프에서 test를 사용할 수 있습니다:

```
#!/bin/sh
X=0
while [ -n "$X" ]
do
  echo "Enter some text (RETURN to quit)"
  read X
  echo "You said: $X"
done
```

이 코드는 RETURN을 누를 때까지 계속 입력을 요청합니다 (X는 길이가 0입니다). 스크립트가 작동하지 않는다고 지적해 주신 Justin Heath에게 감사드립니다. 그 동안 [-n "$X"]에서 $X 주위의 따옴표를 놓쳤기 때문입니다. 이 따옴표가 없으면 $X가 비어 있을 때 테스트할 것이 없습니다. 알렉산더 웨버는 이 스크립트를 실행하면 어수선하게 종료된다는 점을 지적했습니다:

```
$ ./test2.sh
Enter some text (RETURN to quit)
fred
You said: fred
Enter some text (RETURN to quit)
wilma
You said: wilma
Enter some text (RETURN to quit)
```

당신이 말했죠:

```
$
```

루프 내의 다른 테스트로 이 문제를 해결할 수 있습니다:

```
#!/bin/sh
X=0
while [ -n "$X" ]
do
  echo "Enter some text (RETURN to quit)"
  read X
  if [ -n "$X" ]; then
    echo "You said: $X"
  fi
done
```

이 페이지에서는 if 문에 대해 두 가지 다른 구문을 사용했다는 점도 참고하세요. 다음과 같습니다:

```
if [ "$X" -lt "0" ]
then
  echo "X is less than zero"
fi
```

.... 그리고 ....

```
if [ ! -n "$X" ]; then
  echo "You said: $X"
fi
```

if 문과 then 구문 사이에는 반드시 공백이 있어야 합니다. 세미콜론이나 개행 중 어느 것이든 상관없지만 if 문과 then 문 사이에는 둘 중 하나가 반드시 있어야 합니다. 그냥 이렇게 말하는 것이 좋습니다:

```
if [ ! -n "$X" ]
  echo "You said: $X"
```

하지만 then과 fi는 반드시 필요합니다.

9. Case

case 문을 사용하면 if .. then .. else 문 전체를 살펴보는 수고를 덜 수 있습니다 . 구문은 정말 간단합니다 :

```
#!/bin/sh
echo "Please talk to me ..."
while :
do
  read INPUT_STRING
  case $INPUT_STRING in
        hello)
                echo "Hello yourself!"
                ;;
        bye)
                echo "See you again!"
                break
                ;;
        *)
                echo "Sorry, I don't understand"
                ;;

  esac
done
echo
echo "That's all folks!"
```

좋아요 , 세계 최고의 대화 상대는 아니며 단지 예시일 뿐입니다 !

실행해보고 어떻게 작동하는지 확인해 보세요 ...

```
$ ./talk.sh
Please talk to me ...
hello
Hello yourself!
What do you think of politics?
Sorry, I don't understand
bye
See you again!
```

```
That's all folks!
$
```

구문은 매우 간단합니다: case 구분 줄 자체는 항상 같은 형식이며, 이는 변수 INPUT_STRING의 값을 테스트하고 있음을 의미합니다.

그런 다음 우리가 이해하는 옵션이 나열되고 그 뒤에 대괄호로 hello) 및 bye)가 이어집니다. 즉, INPUT_STRING이 hello와 일치하면 해당 코드 섹션이 이중 세미콜론까지 실행되고, INPUT_STRING이 "bye"와 일치하면 "goodbye"메시지가 인쇄되고 루프가 종료됩니다. 스크립트를 완전히 종료하려면 break 대신 exit 명령을 사용해야 합니다. 여기서 세 번째 옵션인 *)는 기본 catch-all 조건으로, 필수는 아니지만 test 변수의 값을 알고 있다고 생각되는 경우에도 디버깅 목적으로 유용할 때가 많습니다.

전체 case 문은 esac(대소문자를 거꾸로 !)으로 끝내고, done으로 while 루프를 종료합니다.

Case 조건은 복잡하기는 하지만 매우 유용하고 강력한 도구가 될 수 있습니다. case 조건은 셸 스크립트에 전달된 매개변수를 구문 분석하는 데 자주 사용되며, 다른 용도로도 사용됩니다.

10. Variables - Part II

이미 설정된 변수 집합이 있으며, 이러한 변수에는 대부분 값을 할당할 수 없습니다. 이러한 변수에는 스크립트가 실행 중인 환경에 대해 파악하는 데 사용할 수 있는 유용한 정보가 포함될 수 있습니다.

첫 번째로 살펴볼 변수 집합은 $0 ... $9 및 $#입니다. 변수 $0은 호출된 프로그램의 기본 이름입니다. $1 ... $9는 스크립트가 호출된 처음 9개의 추가 매개변수입니다. 변수 $@는 모든 매개변수 $1 .. 뭐든지입니다. 변수 $*는 비슷하지만 공백을 보존하지 않고 따옴표를 사용하지 않으므로 " File with spaces"는 "File" "with" "spaces"가 됩니다. 이는 첫 번째 스크립트에서 살펴본 echo와 유사합니다. 일반적으로 $@을 사용하고 $*은 피하세요. 여기서 $#은 스크립트가 호출된 매개변수 수입니다. 예제 스크립트를 살펴보겠습니다:

```
#!/bin/sh
echo "I was called with $# parameters"
echo "My name is $0"
echo "My first parameter is $1"
echo "My second parameter is $2"
echo "All parameters are $@"
```

이 코드를 실행하여 출력을 확인해 보겠습니다:

```
$ /home/steve/var3.sh
I was called with 0 parameters
My name is /home/steve/var3.sh
My first parameter is
My second parameter is
All parameters are
$
$ ./var3.sh hello world earth
I was called with 3 parameters
My name is ./var3.sh
My first parameter is hello
My second parameter is world
All parameters are hello world earth
```

$0의 값은 스크립트가 호출된 방식에 따라 달라집니다. 외부 유틸리티 베이스네임으로 이 문제를 해결할 수 있습니다:

```
echo "My name is `basename $0`"
```

$#과 $1 ... $2는 셸에 의해 자동으로 설정됩니다 . 아래 스크립트를 보면 shift 명령을 사용하여 9개 이상의 매개변수를 사용할 수 있습니다 :

```
#!/bin/sh
while [ "$#" -gt "0" ]
do
  echo "\$1 is $1"
  shift
done
```

이 스크립트는 $#이 0이 될 때까지 , 즉 목록이 비어 있을 때까지 계속 shift를 사용합니다 .

또 다른 특수 변수는 $? 여기에는 마지막 실행 명령의 종료 값이 포함됩니다 . 따라서 코드 :

```
#!/bin/sh
/usr/local/bin/my-command
if [ "$?" -ne "0" ]; then
  echo "Sorry, we had a problem there!"
fi
```

는 모든 것이 정상적으로 진행되면 0 값을 반환하고 , 실패하면 0이 아닌 값으로 종료되는 /usr/local/bin/my-command를 실행하려고 시도합니다 . 그러면 명령을 호출한 후 $? 값을 확인하여 이 문제를 처리할 수 있습니다 . 이렇게 하면 스크립트를 더욱 강력하고 지능적으로 만들 수 있습니다 . 제대로 작동하는 애플리케이션은 성공 시 0을 반환해야 합니다 . 그래서 따옴표를 붙인 것입니다 :

> "로마 제국이 멸망한 주요 원인 중 하나는 0이 없었기 때문에 C 프로그램의 성공적인 종료를 알릴 방법이 없었기 때문입니다 ."
> (Robert Firth)

환경에 의해 설정되는 다른 두 가지 주요 변수는 $$ 및 $! 둘 다 프로세스 번호입니다 . 변수 $$는 현재 실행 중인 셸의 PID(프로세스 식별자)입니다 . 이 변수는 스크립트의 여러 인스턴스가 동시에 실행될 수 있고 모두 고유한 임시 파일이 필요한 경우 유용하게 사용할 수 있는 /tmp/my-script.$$와 같은 임시 파일을 만드는 데 유용할 수 있습니다 . ! 변수는 마지막으로 실행된 백그라운드 프로세스의 PID입니다 . 이는 프로세스가 작업을 진행하면서 추적하는 데 유용합니다 .

또 다른 흥미로운 변수는 IFS입니다 . 이것은 내부 필드 구분 기호입니다 . 기본값은 SPACE TAB NEWLINE 이지만 변경하는 경우 그림과 같이 복사하는 것이 더 쉽습니다 :

```
#!/bin/sh
old_IFS="$IFS"
IFS=:
echo "Please input some data separated by colons ..."
```

```
read x y z
IFS=$old_IFS
echo "x is $x y is $y z is $z"
```

이 스크립트는 다음과 같이 실행됩니다 :

```
$ ./ifs.sh
Please input some data separated by colons ...
hello:how are you:today
x is hello y is how are you z is today
```

다음과 같이 "[hello:how are you:today:my:friend"라고 입력하면 다음과 같이 출력됩니다 :

```
$ ./ifs.sh
Please input some data separated by colons ...
hello:how are you:today:my:friend
x is hello y is how are you z is today:my:friend
```

특히 IFS를 다룰 때는 공백, 개행 및 기타 "제어할 수 없는" 문자를 포함할 수 있다는 점을 인식하는 것이 중요합니다. 따라서 큰따옴표로 묶는 것이 좋습니다 (예 : old_IFS=$IFS 대신 old_IFS="$IFS").

11. Variables - Part III

4장 ('변수 - 1부 ')에서 언급했듯이 변수 주위를 괄호로 묶으면 혼동을 피할 수 있습니다 :

```
foo=sun
echo $fooshine # $fooshine is undefined
echo ${foo}shine # displays the word "sunshine"
```

하지만 이 멋진 괄호에는 이 외에도 훨씬 더 강력한 용도가 있습니다 . 변수가 정의되지 않았거나 (undefined), null인 문제를 처리할 수 있습니다 (셸에서는 정의되지 않은 것과 null인 것은 큰 차이가 없습니다).

기본값 사용하기

사용자에게 입력 메시지를 표시하지만 기본값을 허용하는 다음 코드 조각 (snippet)을 살펴보세요 :

```
#!/bin/sh
echo -en "What is your name [ `whoami` ] "
read myname
if [ -z "$myname" ]; then
  myname=`whoami`
fi
echo "Your name is : $myname"
```

echo에 "-en"을 전달하면 줄 바꿈을 추가하지 않도록 지시합니다 (bash 및 csh의 경우). Dash, Bourne 및 기타 호환 셸의 경우 , 대신 줄 끝에 "\c"를 사용합니다 . Ksh는 두 가지 형식을 모두 이해합니다 . "RETURN"을 눌러 기본값을 수락하면 이 스크립트는 다음과 같이 실행됩니다 :

```
steve$ ./name.sh
What is your name [ steve ] RETURN
Your name is : steve
```

... 또는 사용자 입력으로 :

```
steve$ ./name.sh
What is your name [ steve ] foo
Your name is : foo
```

셸 변수 기능을 사용하면 이 작업을 더 잘 수행할 수 있습니다 . 중괄호와 특수 ":-" 사용법을 사용하면 변수가 설정되지 않은 경우 사용할 기본값을 지정할 수 있습니다 :

```
echo -en "What is your name [ `whoami` ] "
read myname
echo "Your name is : ${myname:-`whoami`}"
```

이것은 특별한 경우로 간주될 수 있는데 , 로그인 이름 (UID)을 인쇄하는 whoami 명령의 출력을 사용하고 있습니다 . 보다 일반적인 예는 다음과 같이 고정 텍스트를 사용하는 것입니다 :

```
echo "Your name is : ${myname:-John Doe}"
```

백틱 (`)의 다른 사용법과 마찬가지로 `whoami`는 서브셸에서 실행되므로 백틱 내에서 cd 명령을 실행하거나 다른 변수를 설정해도 현재 실행 중인 셸에 영향을 미치지 않습니다 .

Using and Setting Default Values

변수가 정의되지 않은 경우 변수를 기본값으로 설정하는 또 다른 구문인 ":="도 있습니다 :

```
echo "Your name is : ${myname:=John Doe}"
```

이 기술은 $myname 변수에 대한 이후의 모든 액세스가 항상 사용자가 입력한 값을 가져오거나 그렇지 않으면 "(John Doe)"를 가져온다는 것을 의미합니다 .

12. External Programs

셸 스크립트 내에서 외부 프로그램이 자주 사용되는데, 몇 가지 기본 제공 명령 (**echo**, **which**, **test**가 일반적으로 기본 제공됨)이 있지만 실제로 유용한 명령은 **tr**, **grep**, **expr**, **cut** 등 유닉스 유틸리티가 많습니다.

백틱 (`)은 종종 외부 명령과 연관되기도 합니다. 따라서 백틱에 대해 먼저 설명하겠습니다. 백틱은 묶은 텍스트가 명령으로 실행될 것임을 나타내는 데 사용됩니다. 이것은 이해하기 매우 간단합니다. 먼저 대화형 셸을 사용하여 /etc/passwd에서 전체 이름을 읽습니다:

```
$ grep "^${USER}:" /etc/passwd | cut -d: -f5
Steve Parker
```

이제 이 출력을 더 쉽게 조작할 수 있는 변수로 가져올 것입니다:

```
$ MYNAME=`grep "^${USER}:" /etc/passwd | cut -d: -f5`
$ echo $MYNAME
Steve Parker
```

따라서 백틱은 단순히 실행하기로 선택한 명령 또는 명령 집합의 표준 출력을 잡는다는 것을 알 수 있습니다. 또한 느린 명령 또는 명령 집합을 실행하고 출력의 다양한 비트를 파싱하려는 경우 성능을 향상시킬 수 있습니다:

```
#!/bin/sh
find / -name "*.html" -print | grep "/index.html$"
find / -name "*.html" -print | grep "/contents.html$"
```

이 코드는 실행하는 데 시간이 오래 걸릴 수 있으며, 두 번 실행하고 있습니다! 더 나은 해결책은:

```
#!/bin/sh
HTML_FILES=`find / -name "*.html" -print`
echo "$HTML_FILES" | grep "/index.html$"
echo "$HTML_FILES" | grep "/contents.html$"
```

참고: 나열된 각 파일 사이의 줄 바꿈을 유지하기 위해 **$HTML_FILES** 주위의 따옴표는 필수입니다. 그렇지 않으면 **grep**에 파일당 한 줄이 아닌 긴 텍스트 한 줄이 표시됩니다.

이렇게 하면 느린 찾기를 한 번만 실행하여 스크립트의 실행 시간을 대략 절반으로 줄일 수 있습니다. 스크립트의 실행 시간을 대략 절반으로 줄일 수 있습니다.

구체적인 예제는 이 튜토리얼의 14장, 힌트 및 팁 섹션에서 자세히 설명합니다.

13. Functions

[illegible]

```
. ./library.sh
```

[illegible]

[illegible]

- [illegible]
- [illegible]
- **return** [illegible]
- **echo** [illegible] **stdout**[illegible] , [illegible] c=`expr $a + $b`[illegible]

[illegible]

[illegible]

```
#!/bin/sh
# A simple script with a function...


add_a_user()
{
USER=$1
PASSWORD=$2
shift; shift;
# Having shifted twice, the rest is now comments ...
COMMENTS=$@
echo "Adding user $USER ..."
```

```
echo useradd -c "$COMMENTS" $USER
echo passwd $USER $PASSWORD
echo "Added user $USER ($COMMENTS) with pass $PASSWORD"
}

###
# Main body of script starts here
###
echo "Start of script..."
add_a_user bob letmein Bob Holness the presenter
add_a_user fred badpassword Fred Durst the singer
add_a_user bilko worsepassword Sgt. Bilko the role model
echo "End of script..."
```

4행은 ()로 끝나는 함수 선언으로 식별됩니다 . 그 뒤에는 {가 오고 , 일치하는 } 다음에 오는 모든 것이 해당 함수의 코드로 간주됩니다 . 이 코드는 함수가 호출될 때까지 실행되지 않습니다 . 함수는 읽혀지지만 실제로 호출될 때까지는 기본적으로 무시됩니다 .

이 예제에서는 사용자 추가 및 패스워드 명령 앞에 echo가 붙었는데 , 이는 올바른 명령이 실행되는지 확인하는 데 유용한 디버깅 기법입니다 . 또한 루트가 되거나 시스템에 수상한 사용자 계정을 추가하지 않고도 스크립트를 실행할 수 있다는 의미이기도 합니다 !

우리는 셸 스크립트가 순차적으로 실행된다는 생각에 익숙해져 있습니다 . 하지만 함수는 그렇지 않습니다 . 이 경우 add_a_user 함수는 읽혀서 구문을 확인하지만 명시적으로 호출될 때까지 실행되지 않습니다 . 실행은 "Start of script..."라는 echo 문으로 시작됩니다 . 다음 줄인 add_a_user bob letmein Bob Holness는 함수 호출로 인식되어 add_a_user 함수가 입력되고 환경에 특정 추가 사항과 함께 실행이 시작됩니다 :

```
$1=bob
$2=letmein
$3=Bob
$4=Holness
$5=the
$6=presenter
```

따라서 해당 함수 내에서 $1은 함수 외부에서 $1이 무엇으로 설정되어 있든 상관없이 bob으로 설정됩니다 . 따라서 함수 내에서 '원본 ' $1을 참조하려면 함수를 호출하기 전에 다음과 같이 이름을 지정해야 합니다 : A=$1과 같이 함수를 호출하기 전에 이름을 지정해야 합니다 . 그런 다음 함수 내에서 $A를 참조할 수 있습니다 . 시프트 명령을 다시 사용하여 $3 이후의 매개변수를 $@로 가져옵니다 . 그런 다음 이 함수는 사용자를 추가하고 비밀번호를 설정합니다 . 이 함수는 해당 효과에 대한 주석을 표시하고 메인 코드의 다음 줄로 제어권을 반환합니다 .

변수의 범위

다른 언어에 익숙한 프로그래머라면 셸 함수의 범위 규칙에 놀랄 수도 있습니다. 기본적으로 매개변수 ($1, $2, $@ 등)를 제외하고는 범위 지정이 없습니다. 다음의 간단한 코드 세그먼트를 예로 들어보겠습니다:

```
#!/bin/sh

myfunc()
{
  echo "I was called as : $@"
  x=2 }

### Main script starts here

echo "Script was called with $@"
x=1
echo "x is $x"
myfunc 1 2 3
echo "x is $x"
```

이 스크립트를 scope.sh a b c로 호출하면 다음과 같은 출력을 제공합니다:

```
Script was called with a b c
x is 1
I was called as : 1 2 3
x is 2
```

함수 내에서 $@ 매개변수는 함수가 호출된 방식을 반영하기 위해 변경됩니다. 그러나 변수 x는 사실상 전역 변수 (global)로, myfunc가 이를 변경했으며 제어가 메인 스크립트로 돌아갈 때에도 이 변경 사항은 여전히 유효합니다.

함수의 출력이 다른 곳으로 파이프되는 경우 함수는 하위 셸에서 호출됩니다. 즉, "myfunc 1 2 3 | tee out.log"는 두 번째에도 여전히 "x는 1"이라고 표시됩니다. 이는 새로운 셸 프로세스가 호출되어 myfunc()를 파이프하기 때문입니다. 이것은 디버깅을 매우 불편하게 만들 수 있습니다. Astrid는 "| tee"가 추가되었을 때 갑자기 실패하는 스크립트가 있었는데, 왜 그래야 하는지 즉시 알 수 없었습니다. 간단한 예로 "ls | grep foo"의 경우, grep이 먼저 시작되어야 하고, ls가 시작되면 그 stdin이 ls의 stdout에 연결되어야 합니다. 셸 스크립트에서는 tee를 통해 파이프할 것이라는 사실을 알기도 전에 셸이 이미 시작되었으므로 운영 체제에서 tee를 시작한 다음 새 셸을 시작하여 myfunc()를 호출해야 합니다. 이것은 실망스럽지만 알아둘 가치가 있습니다.
함수는 호출된 값도 변경할 수 없으며, 스크립트에 전달된 매개 변수가 아닌 변수 자체를 변경해야 합니다. 예시를 통해 이를 보다 명확하게 확인할 수 있습니다:

```
#!/bin/sh

myfunc()
{
  echo "\$1 is $1"
  echo "\$2 is $2"
  # cannot change $1 - we'd have to say:
  # 1="Goodbye Cruel"
  # which is not a valid syntax. However, we can # change $a:
  a="Goodbye Cruel"
}

### Main script starts here

a=Hello
b=World
myfunc $a $b
echo "a is $a"
echo "b is $b"
```

이 다소 냉소적인 함수는 $a를 변경하여 "Hello World"라는 메시지를 "Goodbye Cruel World"로 바꿉니다 .

재귀 (Recursion)

함수는 재귀적일 수 있습니다 . 다음은 팩토리얼 함수의 간단한 예입니다 :

```
#!/bin/sh

factorial()
{
  if [ "$1" -gt "1" ]; then
    i=`expr $1 - 1`
    j=`factorial $i`
    k=`expr $1 \* $j`
    echo $k
  else
    echo 1
  fi
}
```

```
while :
do
  echo "Enter a number:"
  read x
  factorial $x
done
```

약속한 대로 이제 셸 스크립트 간에 라이브러리를 사용하는 방법에 대해 간략하게 설명하겠습니다 . 나중에 살펴보겠지만 공통 변수를 정의하는 데에도 사용할 수 있습니다 .

common.lib

```
# common.lib
# Note no #!/bin/sh as this should not spawn
# an extra shell. It's not the end of the world # to have one, but clearer not to.
#
STD_MSG="About to rename some files..."

rename()
{
  # expects to be called as: rename .txt .bak
  FROM=$1
  TO=$2

  for i in *$FROM
  do
    j=`basename $i $FROM`
    mv $i ${j}$TO
  done
}
```

function2.sh

```
#!/bin/sh
# function2.sh
. ./common.lib
echo $STD_MSG
rename txt bak
```

function3.sh

```
#!/bin/sh
# function3.sh
. ./common.lib
echo $STD_MSG
rename html html-bak
```

여기에서는 두 개의 사용자 셸 스크립트인 function2.sh와 function3.sh가 각각 공통 라이브러리 파일 common.lib를 소싱하고 해당 파일에 선언된 변수와 함수를 사용하는 것을 볼 수 있습니다. 이는 셸 프로그래밍에서 코드 재사용이 어떻게 이루어지는지 보여주는 예시일 뿐입니다.

Exit Codes

종료 코드에 대한 자세한 내용은 튜토리얼의 힌트 및 팁 섹션(14장)의 종료 코드 부분을 참조하세요. 지금은 리턴 콜에 대해 간략히 살펴보겠습니다.

```
#!/bin/sh

adduser()
{
  USER=$1
  PASSWORD=$2
  shift ; shift
  COMMENTS=$@
  useradd -c "${COMMENTS}" $USER
  if [ "$?" -ne "0" ]; then
    echo "Useradd failed"
    return 1
  fi
  passwd $USER $PASSWORD
  if [ "$?" -ne "0" ]; then
    echo "Setting password failed"
    return 2
  fi
  echo "Added user $USER ($COMMENTS) with pass $PASSWORD"
}

## Main script starts here
```

```
adduser bob letmein Bob Holness from Blockbusters
if [ "$?" -eq "1" ]; then
  echo "Something went wrong with useradd"
elif [ "$?" -eq "2" ]; then
   echo "Something went wrong with passwd"
else
  echo "Bob Holness added to the system."
fi
```

이 스크립트는 두 개의 외부 호출 (useradd 및 passwd)을 검사하고 실패할 경우 사용자에게 알려줍니다. 그런 다음 이 함수는 사용자 추가에 문제가 있는 경우 반환 코드 1을, 패스워드에 문제가 있는 경우 반환 코드 2를 정의합니다. 이렇게 하면 호출 스크립트는 문제가 어디에 있는지 알 수 있습니다.

14. Hints and Tips

참고 : 더 많은 힌트와 팁은 https://www.shellscript.sh/tips 에 자주 게시됩니다. 더 흥미롭고 최신의 힌트가 있는지 확인해 보세요. CGI 스크립팅과 같이 다소 학술적인 내용도 있습니다.

유닉스에는 텍스트를 조작하는 유틸리티가 가득하며, 이 튜토리얼의 이 섹션에서는 그 중 몇 가지 강력한 유틸리티에 대해 설명합니다. 여기서 중요한 점은 유닉스에서는 거의 모든 것이 텍스트라는 점입니다. 여러분이 생각할 수 있는 거의 모든 것이 텍스트 파일이나 명령줄 인터페이스 (CLI)로 제어됩니다. 셸 스크립트를 사용하여 자동화할 수 없는 유일한 것은 GUI 전용 유틸리티나 기능입니다. 그리고 유닉스에서는 그 수가 그리 많지 않습니다!

*닉스를 사용하면 "모든 것이 파일이다"라는 말을 들어보셨을 것입니다. - 사실입니다.

여기에는 몇 가지 하위 섹션이 있습니다... 다음은 일반적인 조언, 힌트 및 팁입니다.

CGI Scripting

CGI 프로그래밍을 할 때 주의해야 할 몇 가지 추가 변수와 그 과정에서 얻은 몇 가지 팁이 있습니다. 셸은 CGI 프로그래밍에 적합한 언어가 아닌 것처럼 보일 수 있지만, 작성 속도가 빠르고 디버깅이 간단합니다. 따라서 CGI 스크립트를 위한 이상적인 프로토타이핑 언어이며, 단순하거나 거의 사용하지 않는 CGI 스크립트를 영구적으로 사용하기에도 좋습니다. fortune.cgi를 호출하는 cookie.cgi

Exit Codes

종료 코드는 0에서 255 사이의 숫자로, 모든 Unix 명령이 상위 프로세스로 제어권을 반환할 때 반환되는 숫자입니다. 다른 숫자를 사용할 수도 있지만, 이러한 숫자는 256 모듈로 처리되므로 종료 -10은 종료 246에 해당하고 종료 257은 종료 1에 해당합니다.

셸 스크립트 내에서 이러한 변수를 사용하여 실행된 명령의 성공 또는 실패에 따라 실행 흐름을 변경할 수 있습니다. 이에 대해서는 10장, "변수 - 2부"에서 간략하게 소개했습니다. 여기서는 종료 코드의 해석에 대해 좀 더 자세히 살펴보겠습니다.

성공 **(*Success*)**은 일반적으로 종료 0으로 표시되며, 실패 **(*Failure*)**는 일반적으로 0이 아닌 종료 코드로 표시됩니다. 이 값은 실패의 다양한 이유를 나타낼 수 있습니다. 예를 들어 GNU grep은 성공하면 0을, 일치하는 항목이 없으면 1을, 기타 오류 (구문 오류, 존재하지 않는 입력 파일 등)가 있으면 2를 반환합니다.

오류 상태를 확인하는 세 가지 방법을 살펴보고 각 방법의 장단점에 대해 논의해 보겠습니다.

첫째, 간단한 접근 방식입니다:

```
#!/bin/sh
# First attempt at checking return codes
USERNAME=`grep "^${1}:" /etc/passwd|cut -d":" -f1`
if [ "$?" -ne "0" ]; then
  echo "Sorry, cannot find user ${1} in /etc/passwd"
  exit 1
fi
NAME=`grep "^${1}:" /etc/passwd|cut -d":" -f5`
HOMEDIR=`grep "^${1}:" /etc/passwd|cut -d":" -f6`

echo "USERNAME: $USERNAME"
echo "NAME: $NAME"
echo "HOMEDIR: $HOMEDIR"
```

이 스크립트는 /etc/passwd에 유효한 사용자 아이디를 입력하면 정상적으로 작동합니다. 그러나 잘못된 코드를 입력하면 처음에 예상했던 대로 작동하지 않고 계속 실행되어 표시만 됩니다:

```
USERNAME:
NAME:
HOMEDIR:
```

왜 그럴까요? 앞서 언급했듯이 $? 변수는 마지막으로 실행된 명령의 반환 코드로 설정됩니다. 이 경우, 그것은 cut입니다. 제가 테스트하고 문서를 읽으면서 알 수 있는 한, cut은 보고하는 것처럼 느껴질 정도로 아무런 문제가 없었습니다. 빈 문자열이 입력되면 빈 문자열인 입력의 첫 번째 필드를 반환하는 작업을 수행했습니다. 이제 어떻게 해야 할까요? 여기에 오류가 있으면 grep은 잘라내기가 아니라 오류를 보고합니다. 따라서 cut이 아닌 grep의 반환 코드를 테스트해야 합니다.

```
#!/bin/sh
# Second attempt at checking return codes
grep "^${1}:" /etc/passwd > /dev/null 2>&1
if [ "$?" -ne "0" ]; then
  echo "Sorry, cannot find user ${1} in /etc/passwd"
  exit 1
fi
USERNAME=`grep "^${1}:" /etc/passwd|cut -d":" -f1`
NAME=`grep "^${1}:" /etc/passwd|cut -d":" -f5`
HOMEDIR=`grep "^${1}:" /etc/passwd|cut -d":" -f6`
```

```
echo "USERNAME: $USERNAME"
echo "NAME: $NAME"
echo "HOMEDIR: $HOMEDIR"
```

이렇게 하면 코드가 약간 길어지기는 하지만 문제가 해결됩니다. 이것이 교과서에 나와 있는 기본적인 방법이지만 셸 스크립트에서 오류를 검사하는 방법에 대해 알아야 할 전부는 아닙니다. 이 방법은 특정 명령 시퀀스에 가장 적합하지 않을 수도 있고 유지 관리가 불가능할 수도 있습니다. 아래에서는 두 가지 대체 접근 방식을 살펴보겠습니다.

두 번째 접근 방식으로, 코드를 4줄짜리 테스트로 가득 채우는 대신 테스트를 별도의 함수에 넣음으로써 이 문제를 어느 정도 해결할 수 있습니다:

```
#!/bin/sh
# A Tidier approach

check_errs()
{
  # Function. Parameter 1 is the return code
  # Para. 2 is text to display on failure.
  if [ "${1}" -ne "0" ]; then
    echo "ERROR # ${1} : ${2}"
    # as a bonus, make our script exit with the right error code. exit ${1}
  fi
}

### main script starts here ###

grep "^${1}:" /etc/passwd > /dev/null 2>&1
check_errs $? "User ${1} not found in /etc/passwd"
USERNAME=`grep "^${1}:" /etc/passwd|cut -d":" -f1`
check_errs $? "Cut returned an error"
echo "USERNAME: $USERNAME"
check_errs $? "echo returned an error - very strange!"
```

이를 통해 3개의 개별 테스트를 작성할 필요 없이 사용자 정의 오류 메시지를 사용하여 오류를 3번 테스트할 수 있습니다. 테스트 루틴을 한 번만 작성하면 원하는 만큼 호출할 수 있으므로 프로그래머가 거의 비용을 들이지 않고도 더욱 지능적인 스크립트를 만들 수 있습니다. Perl 프로그래머라면 이 기능이 Perl의 die 명령과 유사하다는 것을 알 수 있습니다.

세 번째 접근 방식으로는 더 간단하고 조잡한 방법을 살펴보겠습니다. 저는 리눅스 커널을 구축할 때 이 방법을 사용하는 편인데, 잘 진행되면 그냥 넘어가면 되지만 문제가 발생하면 운영자가 지능적인 작업(즉, 스크립트로는 할 수 없는 작업!)을 해야 하는 간단한 자동화 방식입니다:

```
#!/bin/sh
cd /usr/src/linux && \
     make dep && make bzImage && make modules && \
     make modules_install && \
     cp arch/i386/boot/bzImage /boot/my-new-kernel && \ cp System.map /boot && \
     echo "Your new kernel awaits, m'lord."
```

이 스크립트는 리눅스 커널 빌드와 관련된 다양한 작업 (시간이 꽤 걸릴 수 있음)을 실행하고 && 연산자를 사용하여 성공 여부를 확인합니다. 이 작업을 수행하려면 if가 필요합니다:

```
#!/bin/sh
cd /usr/src/linux
if [ "$?" -eq "0" ]; then
  make dep
    if [ "$?" -eq "0" ]; then
      make bzImage
      if [ "$?" -eq "0" ]; then
        make modules
        if [ "$?" -eq "0" ]; then
          make modules_install
          if [ "$?" -eq "0" ]; then
            cp arch/i386/boot/bzImage /boot/my-new-kernel
            if [ "$?" -eq "0" ]; then
              cp System.map /boot/
              if [ "$?" -eq "0" ]; then
                echo "Your new kernel awaits, m'lord."
              fi
            fi
          fi
        fi
      fi
    fi
  fi
fi
```

... 개인적으로 따라가기가 꽤 어렵다고 생각합니다.

연산자 && 및 ||는 셸에서 AND 및 OR 테스트에 해당하는 연산자입니다. 위와 같이 함께 사용할 수도 있고, 또는 :

```
#!/bin/sh
cp /foo /bar && echo Success || echo Failed
```

이 코드는 다음과 같이 echo합니다 .

```
Success
```

또는

```
Failed
```

이것은 cp 명령이 성공하느냐 실패하느냐에 따라 다릅니다 . 다음 구조를 자세히 보세요 :

```
command && command-to-execute-on-success \
  || command-to-execute-on-failure
```

각 부분에는 하나의 명령만 포함될 수 있습니다 . 이 방법은 간단한 성공 /실패 시나리오에는 편리하지만 , 에코 명령 자체의 상태를 확인하려는 경우 어떤 &&와 ||가 어떤 명령에 적용되는지 금방 헷갈리기 쉽습니다 . 또한 유지 관리도 매우 어렵습니다 . 따라서 이 구조는 간단한 명령 시퀀싱에만 사용하는 것이 좋습니다 .

이전 버전에서는 cp 명령의 성공 또는 실패 여부에 따라 서브셸을 사용하여 여러 명령을 실행할 수 있다고 제안한 적이 있습니다 :

```
cp /foo /bar && \
  ( echo Success ; echo Success part II; ) || \
  ( echo Failed ; echo Failed part II )
```

그러나 실제로 Marcel은 이것이 제대로 작동하지 않는다는 것을 발견했습니다 . 서브셸의 구문은 다음과 같습니다 :

```
( command1 ; command2; command3 )
```

서브셸의 반환 코드는 최종 명령 (이 예에서는 command3)의 반환 코드입니다 . 이 반환 코드는 전체 명령에 영향을 미칩니다 . 따라서 이 스크립트의 출력은 다음과 같습니다 :

```
cp /foo /bar && \
  ( echo Success ; echo Success part II; /bin/false ) ||\
  ( echo Failed ; echo Failed part II )
```

cp가 성공했기 때문에 성공 부분을 실행하고 , /bin/false가 실패를 반환하기 때문에 실패 부분도 실행한다는 것입니다 :

```
Success
Success part II
Failed
Failed part II
```

따라서 다른 조건의 상태에 따라 여러 명령을 실행해야 하는 경우 표준 if, then, else 구문을 사용하는 것이 더 좋고 훨씬 명확합니다 .

Simple Expect Replacement

다음은 expect를 간단하게 대체할 수 있는 방법입니다 . 많은 분들이 이 작업을 수행하는 방법에 대해 문의해 주셨고 , 제가 Sun Microsystems의 Explorer 유틸리티에 사용되는 힌트 및 팁에서 보여드린 예제에서 영감을 받아 기대의 매우 간단한 버전을 소개해 드립니다 .

expect.txt의 구문은 매우 간단합니다 :

```
S command E[delay] expected_text
```

따라서 명령은 "S"(Send의 경우)로 시작하여 표시되고 , 예상 결과는 "E"로 표시됩니다 . 일부 명령은 완료하는 데 시간이 걸릴 수 있으므로 결과를 예상하기 전에 지연 시간을 지정할 수 있습니다 : "E10 $"는 10초 동안 기다린 후 달러 프롬프트를 표시합니다 . 예상 텍스트를 찾지 못하면 스크립트는 1초를 기다린 후 다시 시도하고 , 2초 , 3초를 기다린 후 예상 텍스트가 발견되거나 MAX_WAITS에 정의된 최대값에 도달할 때까지 기다립니다 . 지연은 선택 사항이므로 "E $"는 즉시 프롬프트가 표시될 것으로 예상됩니다 .

MAX_WAITS=5인 경우 최대 지연 시간은 5초가 아니라 1+2+3+4+5=15초가 됩니다 .

```
#!/bin/sh
# expect.sh | telnet > file1
host=127.0.0.1
port=23
file=file1
MAX_WAITS=5

echo open ${host} ${port}

while read l
do
c=`echo ${l}|cut -c1`
 if [ "${c}" = "E" ]; then
   expected=`echo ${l}|cut -d" " -f2-`
```

```
  delay=`echo ${l}|cut -d" " -f1|cut -c2-`
  if [ -z "${delay}" ]; then
    sleep ${delay}
  fi
  res=1
  i=0
  while [ "${res}" -ne "0" ]
  do
   tail -1 "${file}" 2>/dev/null | grep "${expected}" > /dev/null
   res=$?
   sleep $i
   i=`expr $i + 1`
   if [ "${i}" -gt "${MAX_WAITS}" ]; then
     echo "ERROR : Waiting for ${expected}" >> ${file}
     exit 1
    fi
  done
 else
  echo ${l} |cut -d" " -f2-
 fi
done < expect.txt
```

이렇게 실행됩니다 :

```
$ expect.sh | telnet > file1
```

이렇게 하면 세션의 기록이 포함된 file1이라는 파일이 생성됩니다 . 이 경우 로그인 프로세스 , /tmp의 ls, cal의 출력이 됩니다 . 예를 들어 :

```
telnet> Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.

declan login: steve
Password:
Last login: Thu May 30 23:52:50 +0100 2002 on pts/3 from localhost.
No mail.
steve:~$ ls /tmp
API.txt                 cgihtml-1.69.tar.gz          orbit-root
cal
a.txt                   cmd.txt                      orbit-steve
```

```
apache_1.3.23.tar.gz   defaults.cgi           parser.c
b.txt                  diary.c                patchdiag.xref
background.jpg         drops.jpg              sh-thd-1013541438
blocks.jpg             fortune-mod-9708.tar.gz  stone-dark.jpg
blue3.jpg              grey2.jpg              water.jpg
c.txt                  jpsock.131.1249
steve:~$ cal
   May 2002
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
steve:~$ exit
logout
```

Trap

Trap은 간단하지만 매우 유용한 유틸리티입니다. 현재 디렉터리의 모든 파일에서 FOO를 BAR로 바꾸는 이 간단한 스크립트와 같이 스크립트가 임시 파일을 생성하는 경우, 스크립트가 종료될 때 /tmp는 깨끗해집니다. 하지만 스크립트가 도중에 중단되면 /tmp에 파일이 남아있을 수 있습니다:

```
#!/bin/sh

trap cleanup 1 2 3 6

cleanup()
{
  echo "Caught Signal ... cleaning up."
  rm -rf /tmp/temp_*.$$
  echo "Done cleanup ... quitting."
  exit 1
}

### main script
for i in *
do
```

```
  sed s/FOO/BAR/g $i > /tmp/temp_${i}.$$ && mv /tmp/temp_${i}.$$ $i
done
```

trap 문은 스크립트에서 signal 1, 2, 3 또는 6에 대해 cleanup()을 실행하도록 지시합니다. 가장 일반적인 것(CTRL-C)은 signal 2(SIGINT)입니다. 이것은 매우 흥미로운 용도로도 사용될 수 있습니다:

```
#!/bin/sh

trap 'increment' 2

increment()
{
  echo "Caught SIGINT ..."
  X=`expr ${X} + 500`
  if [ "${X}" -gt "2000" ]
  then
    echo "Okay, I'll quit ..."
    exit 1
  fi
}

### main script
X=0
while :
do
  echo "X=$X"
  X=`expr ${X} + 1`
  sleep 1
done
```

위의 스크립트는 꽤 재미있습니다. CTRL-C를 잡아서 종료하지 않고 실행 방식만 변경합니다. 이것이 어떤 긍정적인 효과와 부정적인 효과를 가져올 수 있는지는 독자의 몫으로 남겨두겠습니다.) 이 특정 예제에서는 4번의 인터럽트 (또는 2000초) 후에 종료하도록 허용합니다. 처리할 기회를 얻지 못한 채 kill -9 <PID>에 의해 모든 것이 종료된다는 점에 유의하세요.

다음은 몇 가지 일반적인 인터럽트를 정리한 표입니다:

Number	SIG	뜻
0	0	셸에서 종료할 때
1	SIGHUP	깔끔한 정리
2	SIGINT	인터럽트

3	SIGQUIT	종료 (Quit)
6	SIGABRT	취소 (Abort)
9	SIGKILL	바로 Kill함 (덫에 걸리지 않음)
14	SIGALRM	알람 시계
15	SIGTERM	종료 (Terminate)

스크립트가 자체적으로 신호를 무시하는 환경(예: **nohup** 제어 상태)에서 시작된 경우 스크립트는 해당 신호도 무시합니다.

echo: -n vs \c

이미 눈치채셨겠지만, echo 문을 사용하면 명령의 끝에 줄 바꿈이 추가됩니다. 이에 대한 수정 사항이 있습니다... 아니, 더 정확하게는 두 가지 수정 사항이 있습니다.

일부 Unix 시스템에서는 **echo -n message**를 사용하여 echo에 줄 바꿈을 추가하지 말라고 지시하고, 다른 시스템에서는 **echo message \c**를 사용하여 동일한 작업을 수행합니다:

```
echo -n "Enter your name:"
read name
echo "Hello, $name"
```

이는 일부 시스템에서 작동하며 다음과 같이 표시됩니다:

```
Enter your name: Steve
Hello, Steve
```

그러나 다른 시스템에서는 다음과 같이 코드를 작성해야 합니다:

```
echo "Enter your name: \c"
read name
echo "Hello, $name"
```

그러면 해당 시스템에서 동일한 결과를 얻을 수 있습니다.

골치 아픈 일이죠. 다음은 두 가지 모두에서 작동하는 해결 방법입니다:

```
if [ "`echo -n`" = "-n" ]; then
   n=""
   c="\c"
```

```
else
    n="-n"
    c=""
fi

echo $n Enter your name: $c
read name
echo "Hello, $name"
```

echo -n이 제대로 해석되지 않으면 -n이라는 텍스트만 echo 하게되며, 이 경우 $n은 빈 문자열로 설정되고 $c는 \c로 설정됩니다. 그렇지 않으면 그 반대의 결과가 나오므로 $n은 -n으로 설정되고 $c는 빈 문자열로 설정됩니다.

위에서 이미 간단하지만 효과적인 cut 명령의 사용법을 보여드렸습니다. 여기서는 좀 더 일반적으로 사용되는 외부 프로그램 몇 가지를 예로 들어 설명하겠습니다.

grep은 셸 스크립트 프로그래머에게 매우 유용한 유틸리티입니다. grep의 예는 다음과 같습니다:

```
#!/bin/sh
steves=`grep -i steve /etc/passwd | cut -d: -f1`
echo "All users with the word \"steve\" in their passwd"
echo "Entries are: $steves"
```

일치하는 항목이 하나만 있으면 이 스크립트는 정상적으로 보입니다. 그러나 /etc/passwd에 "steve"라는 단어가 포함된 두 줄이 있으면 대화형 셸이 표시됩니다:

```
$> grep -i steve /etc/passwd
steve:x:5062:509:Steve Parker:/home/steve:/bin/bash
fred:x:5068:512:Fred Stevens:/home/fred:/bin/bash
$> grep -i steve /etc/passwd |cut -d: -f1
steve
fred
```

하지만 스크립트가 표시됩니다:

```
Entries are: steve fred
```

결과를 변수에 넣음으로써 NEWLINE을 공백으로 변경했습니다. sh 매뉴얼 페이지에서는 $IFS의 첫 번째 문자가 이 용도로 사용된다고 알려줍니다. IFS는 기본적으로 <space><tab><cr>입니다. 아마도 우리는 NEWLINE을 유지하고 싶었을 것입니다: 공백을 NEWLINEs.... 으로 만들면 더 보기 좋을 것입니다. 이것은 tr을 위한 작업입니다:

```
#!/bin/sh
steves=`grep -i steve /etc/passwd | cut -d: -f1`
echo "All users with the word \"steve\" in their passwd"
echo "Entries are: "
echo "$steves" | tr ' ' '\012'
```

tr은 공백을 8진수 문자 012(NEWLINE)로 변환한다는 점에 유의하세요. tr의 또 다른 일반적인 용도는 범위를 사용하는 것입니다. 예를 들어 텍스트를 대문자 또는 소문자로 변환할 수 있습니다:

```
#!/bin/sh
steves=`grep -i steve /etc/passwd | cut -d: -f1`
echo "All users with the word \"steve\" in their passwd"
echo "Entries are: "
echo "$steves" | tr ' ' '\012' | tr '[a-z]' '[A-Z]'
```

여기서는 [a-z]를 [A-Z]로 번역했습니다. a-z 범위에는 A-Z와 정확히 같은 수의 값이 있다는 점에 유의하세요. 그러면 ASCII 범위 a-z에 속하는 모든 문자를 A-Z로 번역할 수 있습니다. 즉, 소문자를 대문자로 변환하는 것입니다. tr은 실제로 이보다 더 똑똑합니다. tr [:lower:] [:upper:]이 더 잘 작동하고 가독성도 더 높을 수 있습니다. 또한 이식성이 떨어지므로 모든 tr이 이 작업을 수행할 수 있는 것은 아닙니다.

Cheating

속임수를 쓸 수 없는 사람들

치팅은 잘못된 것이 아닙니다! 셸이 잘하지 못하는 것도 있습니다. 두 가지 유용한 도구는 sed와 awk입니다. 이 두 가지 유틸리티는 그 자체로 미니 프로그래밍 언어로 사용할 수 있는 매우 강력한 유틸리티이지만, 셸 스크립트에서는 매우 간단하고 특정한 이유로 사용되는 경우가 많습니다.

이는 시스템에서 대용량 실행 파일(sed의 경우 52k, awk의 경우 110k)을 로드해야 한다는 것을 의미하며, 이는 끔찍한 일이지만, 훌륭한 작업자가 도구를 탓하지 않는 이유는 훌륭한 작업자는 처음부터 올바른 도구를 사용하기 때문입니다. 그래서 이 두 가지 도구를 아주 간단하게 소개해드리겠습니다.

Cheating with awk

텍스트 파일에 있는 문자, 줄, 단어의 개수를 계산하는 wc를 예로 들어 보겠습니다. 출력은 다음과 같습니다:

```
$ wc hex2env.c
       102     189    2306  hex2env.c
```

줄 수를 변수로 가져오고 싶다면 다음을 사용하면 됩니다:

```
NO_LINES=`wc -l file`
```

를 입력하면 전체 줄을 읽을 수 있습니다 . 출력에 공백이 추가되어 있기 때문에 숫자 102를 문자열에 안정적으로 넣을 수 없습니다 . 대신 , awk가 C의 scanf와 유사하게 작동한다는 사실을 이용해 원치 않는 공백을 제거합니다 . 이를 $1 $2 $3 등의 변수에 넣습니다 . 그래서 이 구조를 사용합니다 :

```
NO_LINES=`wc -l file | awk '{ print $1 }'`
```

이제 NO_LINES 변수는 102입니다 .

Cheating with sed

또 다른 편리한 유틸리티는 stream editor인 sed입니다 . Perl은 정규식 처리에 매우 능숙하지만 셸은 그렇지 않습니다 . 따라서 예를 들어 sed를 호출하여 s/from/to/g 구문을 빠르게 사용할 수 있습니다 :

```
sed s/eth0/eth1/g file1 >  file2
```

는 파일 1에 있는 eth0의 모든 인스턴스를 파일 2에 있는 eth1로 변경합니다 . 단일 문자만 변경하는 경우 , 크기가 작고 로드 속도가 빠른 tr을 사용할 수 있습니다 . tr이 할 수 없는 또 다른 일은 파일에서 문자를 제거하는 것입니다 :

```
echo ${SOMETHING} | sed s/"bad word"//g
```

이렇게 하면 ${SOMETHING} 변수에서 "'bad word'"라는 문구가 제거됩니다 . 이렇게 말하고 싶을 수도 있습니다 .
"하지만 grep은 그렇게 할 수 있습니다 !"라고 말하고 싶을 수도 있습니다 . - grep은 전체 줄만 처리합니다 . 파일을 생각해 봅시다 :

```
This line is okay.
This line contains a bad word. Treat with care.
This line is fine, too.
```

grep은 두 번째 줄 전체를 제거하여 두 줄짜리 파일만 남기고 , sed는 파일을 읽기로 변경합니다 :

```
This line is okay.
This line contains a . Treat with care.
This line is fine, too.
```

Telnet hint

이것은 제가 Sun의 Explorer 유틸리티에서 배운 유용한 기술입니다 . 텔넷은 더 이상 서버에서 사용되지 않지만 터미널 집중 장치 등과 같은 일부 네트워크 장치에서는 여전히 사용됩니다 . 이와 같은 스크립트를 직접 만들거나 명령줄에서 실행할 수 있습니다 :

```
$ ./telnet1.sh | telnet
```

몇몇 사람들이 이 문제에 대해 물어본 적이 있는데, 저는 꽤 복잡하고 부피가 큰 코드 모음인 기대 코드를 추천하는 경향이 있습니다. 이 코드는 시스템 간에 이식성이 높아야 합니다 (egrep가 있는 한). 시스템에서 작동하지 않는 경우 -q 스위치와 함께 GNU grep을 사용하거나 전용 grep을 사용하여 /dev/null로 직접 실행해 보세요. 그래도 기대 설치보다는 훨씬 쉽습니다.

```
#!/bin/sh
host=127.0.0.1
port=23
login=steve
passwd=hellothere
cmd="ls /tmp"

echo open ${host} ${port}
sleep 1
echo ${login}
sleep 1
echo ${passwd}
sleep 1
echo ${cmd}
sleep 1
echo exit
```

하지만 Sun은 몇 가지 영리한 오류 검사 코드를 추가했습니다 (현재 셸 또는 셸 스크립트에서 변수를 설정하고 내보내어 읽을 수 있는 파일에 비밀번호가 저장되는 것을 방지할 수 있습니다):

```
$ ./telnet2.sh | telnet > file1
```

```
#!/bin/sh
# telnet2.sh | telnet > FILE1
host=127.0.0.1
port=23
login=steve
passwd=hellothere
cmd="ls /tmp"
timeout=3
file=file1
prompt="$"

echo open ${host} ${port}
```

```
sleep 1
tout=${timeout}
while [ "${tout}" -ge 0 ]
do
    if tail -1 "${file}" 2>/dev/null | \
      egrep -e "login:" > /dev/null
    then
        echo "${login}"
        sleep 1
        tout=-5
        continue
    else
        sleep 1
        tout=`expr ${tout} - 1`
    fi
done

if [ "${tout}" -ne "-5" ]; then
  exit 1
fi

tout=${timeout}
while [ "${tout}" -ge 0 ]
do
    if tail -1 "${file}" 2>/dev/null | \
       egrep -e "Password:" > /dev/null
    then
        echo "${passwd}"
        sleep 1
        tout=-5
        continue
    else
      if tail -1 "${file}" 2>/dev/null | \
     egrep -e "${prompt}" > /dev/null
      then
        tout=-5
      else
        sleep 1
        tout=`expr ${tout} - 1`
      fi
```

```
    fi
done

if [ "${tout}" -ne "-5" ]; then
  exit 1
fi

> ${file}

echo ${cmd}
sleep 1
echo exit
```

이 버전에서는 출력이 file1에 저장되며, 이 파일은 실제로 스크립트에서 진행 상황을 확인하는 데 사용됩니다. "> ${file}"을 추가하여 파일로 수신되는 출력이 로그인 프로세스가 아닌 명령의 출력만 되도록 했습니다.

15. Quick Reference

쉽게 추측할 수 없는 일부 명령어와 코드의 의미에 대한 빠른 참조 가이드입니다 .

명령 / 변수 / 구문 구조	설명	예시
&	백그라운드에서 이전 명령 실행	ls &
&&	논리 AND	if ["$foo" -ge "0"] && ["$foo" -le "9"]
||	논리 OR	if ["$foo" -lt "0"] \|\| ["$foo" -gt "9"] (not in Bourne shell)
^	라인의 시작	grep "^foo"
$	라인 끝	grep "foo$"
=	문자열 등호 (cf. -eq)	if ["$foo" = "bar"]
!	논리 NOT	if ["$foo" != "bar"]
$$	현재 셸의 PID	echo "my PID = $$"
$!	마지막 백그라운드 명령의 PID	ls & echo "PID of ls = $!"
$?	마지막 종료 상태	ls ;
	명령어	echo "ls returned code $?"
$0	현재 명령의 이름 (호출 시)	echo "I am $0"
$1	현재 명령의 첫 번째 매개변수 이름	echo "My first argument is $1"
$9	현재 명령의 아홉 번째 매개변수 이름	echo "My ninth argument is $9"
$@	현재 명령의 모든 매개변수 (공백 및 따옴표 유지)	echo "My arguments are $@"
$*	현재 명령의 모든 매개변수 (공백 및 따옴표 미보존)	echo "My arguments are $*"

-eq	숫자 동일	if ["$foo" -eq "9"]
-ne	숫자 비 동일	if ["$foo" -ne "9"]
-lt	미만	if ["$foo" -lt "9"]
-le	보다 작거나 같음	if ["$foo" -le "9"]
-gt	보다 큰	if ["$foo" -gt "9"]
-ge	보다 크거나 같음	if ["$foo" -ge "9"]
-z	문자열 길이가 0	if [-z "$foo"]
-n	문자열 길이가 0이 아닌 경우	if [-n "$foo"]
-nt	다음보다 최신	if ["$filea" -nt"$fileb"]
-d	디렉터리인가	if [-d /bin]
-f	파일인가	if [-f /bin/ls]
-r	읽을 수 있는 파일인가	if [-r /bin/ls]
-w	쓰기 가능한 파일인가	if [-w /bin/ls]
-x	실행 파일	if [-x /bin/ls]
괄호 : (...)	함수 정의	function myfunc() { echo hello }

16. Interactive Shell

다음은 UNIX 또는 Linux 셸을 대화형으로 사용하기 위한 몇 가지 간단한 힌트입니다. 개인적으로 대부분의 대화형 셸은 거의 모든 *nix 버전에서 사용할 수 있으며 로그인 셸로 사용하기에 매우 편리하기 때문에 bash 셸을 추천합니다. 그러나 루트 셸은 bash 또는 본 셸을 가리키든 항상 /bin/sh여야 합니다.

bash

bash에는 위쪽 및 아래쪽 화살표 키로 이전 명령의 기록을 스크롤할 수 있는 매우 편리한 기록 검색 도구가 있습니다. 더 유용하게는 Ctrl+r을 누르면 명령줄의 어떤 부분과도 일치하는 역방향 검색이 수행됩니다. ESC를 누르면 선택한 명령이 현재 셸에 붙여넣어져 필요에 따라 편집할 수 있습니다.

이전에 실행한 명령을 반복하고 싶은데 어떤 문자로 시작했는지 알고 있다면 이렇게 하면 됩니다:

```
bash$ ls /tmp
(list of files in /tmp)
bash$ touch /tmp/foo
bash$ !l
ls /tmp
(list of files in /tmp, now including /tmp/foo)
```

화살표 키뿐만 아니라 PageUp 및 PageDn을 사용하여 명령줄의 시작과 끝으로 이동할 수 있습니다.

ksh

vi 또는 emacs 모드에서 히스토리 명령을 추가하여 ksh를 더 유용하게 사용할 수 있습니다. 이 작업을 수행하는 방법은 정확한 상황에 따라 여러 가지가 있습니다. set -o vi, ksh -o vi 또는 exec ksh -o vi(이맥스 모드를 선호하는 경우 "vi"를 "emacs"로 대체할 수 있습니다).

다른 대화형 셸에서 ksh 세션을 시작하려면 다음과 같이 ksh를 호출하면 됩니다:

```
csh% # oh no, it's csh!
csh% ksh
ksh$ # phew, that's better ksh$ # do some stuff under ksh
ksh$ # then leave it back at the csh prompt: ksh$ exit
csh%
```

그러면 새 ksh 세션이 시작되며, 이 세션에서 종료한 후 이전 셸로 돌아갈 수 있습니다. 또는 실행 명령을 사용하여 csh(또는 다른 셸)를 ksh 셸로 바꿀 수도 있습니다:

```
csh% # oh no, it's csh!
csh% exec ksh
ksh$ # do some stuff under ksh ksh$ exit


login:
```

여기서 차이점은 csh 세션을 되돌릴 수 없다는 것입니다.

좋은 점은 기록입니다:

```
csh% ksh
ksh$ set -o vi
ksh$ # You can now edit the history with vi-like commands,
  # and use ESC-k to access the history.
```

ESC를 누른 다음 k를 누르면 명령 기록을 거꾸로 스크롤할 수 있습니다. 다음과 같이 vi 명령 모드 및 입력 모드 명령을 사용하여 명령을 편집할 수 있습니다:

```
ksh$ touch foo
  ESC-k (enter vi mode, brings up the previous command)
  w (skip to the next word, to go from "touch" to "foo"
  cw (change word) bar (change "foo" to "bar")
ksh$ touch bar
```