Using Regular Expression

Seong-Kook Cin

1999년 12월 14일

목 록

1 파일 매칭과 정규식을 혼동하지 말자

우선 주의해야 할 것이 있어 먼저 말해 둡니다. Shell이 파일 패턴 매칭에 쓰는 와일드카드(wildcard)와 정규식을 혼동해서는 안됩니다. 또한 와일드 카드와 정규식은 공통적으로 다음과 같은 문자들을 특별하게 취급합니다: '`*`'와 '?', '( )', '`[ ]`', '`|`'.

Shell과 *find*, *cpio*와 같은 프로그램들은 정규식이 아닌 파일 패턴 매칭을 위해 와일드카드 문자를 쓴다는 것을 먼저 기억해 두어야 합니다.

또한 이러한 특수문자들은 shell에 의해 확장되어 버리기 때문에 미리 따옴표로 둘러싸야 합니다. 아래의 명령을 입력하면:

```
$ grep [A-Z]*.c chap[12]
```

Shell은 다음과 같이 확장할 것입니다:

```
grep Array.c Bug.c Comp.c chap1 chap2
```

가장 간단한 방법은 의심스러운 부분은 모두 작은 따옴표인 ‘’’으로 둘러싸는 것입니다.

2 Using Metacharacters in Regular Expression

정규식을 쓸 때에는 다음 세가지를 고려해야 합니다:

1. *Anchor*는 텍스트 한 줄에서 패턴이 위치하는 곳을 지정하며,
2. *Character*들은 그 위치에서 하나 이상의 문자에 매치(match)되며,
3. *Modifier*는 앞에 나온 문자 집합의 반복 횟수를 지정합니다.

Caret (∧) 문자는 줄의 첫 부분이라는 것을 나타내는 anchor이며, hash mark (#)는 단순히 문자 자체를 의미합니다. Asterisk (*)는 modifier이며, 이 의미는 앞에 나온 문자 집합이 0개 이상 있다는 뜻입니다. 1개 이상이 아닌 0개 이상인 것에 주의하기 바랍니다.

정규식은 크게 단순 정규식[1]과 확장 정규식[2]으로 나눌 수 있습니다. *awk*, *egrep*과 같은 몇몇의 프로그램은 확장 정규식을 지원하며, 나머지 대부분의 프로그램은 단순 정규식 표현만 지원합니다. 이 프로그램들에는 *vi*, *sed*, *grep*, *csplit*, *dbx*, *more*, *ed*, *expr*, *lex*등이 있습니다.

2.1 Anchors

패턴의 위치를 지정하는 anchor에는 ∧와 $가 있습니다. 전자는 start anchor라고 하며, 후자는 end anchor라고 합니다. 각각은 텍스트 한 줄에서 첫 부분, 끝 부분을 의미합니다. 따라서 정규식 “∧A”는 문자 ‘A’이되, 문장의 처음에 나오는 ‘A’를 의미합니다. 마찬가지로 “$A”는 문자 ‘A’이되, 문장의 마지막에 나오는 ‘A’만을 의미합니다. 두 anchor 모두 제 위치에 있을 때에만 그 역할을 발휘합니다. 예를 들어 정규식 “1∧”나 “$1”에서의 ∧, $는 그 역할을 하지 못하고, 단순히 일반 문자로 해석됩니다. 만약 실제로 문장의 첫 부분에 오는 ∧이나, 문장의 마지막에 오는 $를 원한다면 이들 문자 앞에 \를 붙이면 됩니다.

이 문자들은 단지 정규식이 아닌 비슷한 의미로도 많이 쓰입니다. 예를 들어 *vi*의 명령 모드에서 ‘$’는 문장의 끝으로 커서를 이동하는 데에 쓰이며, ‘∧’는 문장의 처음으로 이동시키는 데에 쓰입니다. 또한 C shell에서 ‘!∧’는 바로 전 줄의 첫 번째 인자를 지정하는데에 쓰이고, ‘!$’는 마지막 인자를 지정하는데에 쓰입니다.

다른 프로그램에서도 비슷한 명령들을 볼 수 있습니다. *ed*와 *sed*에서 ‘$’는 파일의 마지막 줄을 의미하며, ‘cat -v -e’ 명령은 각 줄의 끝을 ‘$’로 나타내게 해 줍니다.

[1] simple regular expression
[2] extended regular expression

그림 1: Regular Expression Anchor Character Examples

Pattern	Matches
∧A	문장의 처음에 나오는 'A'
A$	문장의 마지막에 나오는 'A'
A∧	아무데나 올 수 있는 'A∧'
$A	아무데나 올 수 있는 '$A'
∧\∧	문장의 처음에 나오는 '∧'
∧∧	'∧\∧'와 동일
\$$	문장의 마지막에 나오는 '$'

2.2 Matching a Character with a Character Set

가장 간단한 문자 집합은 한 문자로 나타내는 것입니다. 예를 들어 정규식 'the'는 't', 'h', 'e'의 세 문자로 이루어진 것이며, 문자열 'the'에만 match됩니다. 그러나 이 정규식은 'the' 뿐만 아니라 'other'에도 match될 수 있습니다. 이를 방지하려면 정규식의 앞에 (또는 뒷 부분, 또는 앞뒤 모두에) 공백을 주어 ' the'로 쓰면 'other'와 같은 단어에 match되는 것을 막을 수 있습니다.

앞에서 배운 anchor와 조합하면 여러가지로 응용할 수 있습니다. 예를 들어 자신에게 온 편지 중 보낸이의 목록을 알고 싶다면 다음과 같이 할 수 있습니다. e-mail 형식상 보낸이는 "From: xxx"의 형태로 기록됩니다. 따라서 정규식 'From'을 *grep* 명령에 쓰면 보낸이의 목록을 뽑을 수 있습니다. 하지만 단어 "From"은 매우 빈번하게 쓰이는 것이라 필요없는 줄까지 출력할 경우가 예상됩니다. 이 때, 앞에서 배운 anchor인 '∧'를 써서 다음과 같이 하면 - 100% 보장할 순 없지만 - 보낸이의 목록만을 얻을 수 있습니다:

```
$ grep '^From: ' /usr/spool/mail/$USER
```

앞으로 설명할 몇몇 문자들은 정규식에서 특별한 뜻을 가집니다. 이러한 문자들을 글자 그대로의 의미로 쓰고 싶다면 앞에다 '\'를 붙이면 됩니다.

2.3 Match any Character with '.' (Dot)

Dot (.) 문자는 정규식에서 한 글자에 해당합니다. 글자의 종류는 상관없습니다. 따라서 정규식 '...'은 세 글자로 구성된 단어에 해당합니다. 또한 한 글자로만 된 줄은 '∧.$'로 나타낼 수 있습니다.

2.4 Specifying a Range of Characters with [...]

특정 범위에 해당하는 글자를 나타내고 싶다면 [] 사이에 넣으면 됩니다. 예를 들어 'A', 'B', 'C' 세 글자 중 한 글자에 해당하는 정규식은 '[ABC]'입니다. 또한 한 숫자로만 이루어진 줄은 '∧[0123456789]'로 쓸 수 있습니다.

Hyphen ('-') 문자를 써서 범위로 지정할 수 있습니다. 예를 들어 위에서 한 숫자만으로 이루어진 줄은 간단히 '∧[0-9]'로 나타낼 수 있습니다. 알파벳, 숫자, 밑줄 문자 중 한 글자에 해당하는 정규식은 '[A-Za-z0-9_]입니다.

좀 더 복잡한 예를 들어 보면:

- 줄의 첫 문자가 'T'이고,

그림 2: Regular Expression Character Set Examples

Regular Expression	**Matches**
[0-9]	숫자 한 글자
[∧0-9]	숫자가 아닌 한 글자
[-0-9]	숫자나 '-' 글자
[0-9-]	숫자나 '-' 글자
[∧-0-9]	숫자도 아니고 '-'도 아닌 글자
[]0-9]	숫자나 ']' 글자
[0-9]]	숫자 한 글자 뒤에 ']'가 오는 문자열
[0-99-z]	숫자나 문자 코드가 '9'-'z' 사이인 글자
[]0-9-]	숫자 또는 '-', 또는 ']' 글자

- 다음으로 소문자 한 글자가 나오고,
- 다음으로 알파벳 소문자 모음(vowel)이 나오고,
- 세 글자로 이루어진 (네 번째 문자가 ' ')

정규식은 '∧T[a-z][aeiou] '입니다.

2.5 Exceptions in a Character Set

[]을 쓰면 주어진 범위에 해당하는 문자를 찾을 수 있습니다. 반대로 '∧'을 함께 쓰면 주어진 범위에 해당하지 않은 문자를 찾습니다. 예를 들어 '[∧aeiou]는 소문자 모음이 아닌 문자에 해당하는 정규식입니다. 이 때 '∧' 문자는 반드시 '[' 다음에 와야 합니다.

다른 anchor들과 마찬가지로 제 자리에 오지 않은 문자들은 특별한 뜻을 가지지 않습니다. 예를 들어 '[' 문자 바로 뒤에 나오는 ']' 문자나 '[' 뒤에 바로 나오는 '-'은 일반 문자로 취급됩니다. 그림 2를 참고하기 바랍니다.

2.6 Repeating Characteer Sets with *

정규식의 마지막 부분인 'modifier'는 바로 앞의 패턴의 반복 횟수를 지정합니다. '*' 문자는 바로 앞 패턴이 0번 이상 나온다는 것을 의미합니다. 따라서 '0*'는 문자 '0'이 0개 이상 나올 수 있다는 것을 의미합니다. 따라서 간단한 10진 수치는 '[0-9]*'으로 표현할 수 있을 것입니다.

조심할 것은 1개 이상이 아닌 0개 이상이라는 것입니다. 예를 들어 문장의 첫 부분에서 '#'로 시작하는 단어를 찾기 위해 '∧#*'를 쓴다면 '#'로 시작하는 줄은 당연히 포함되고, - 0번 반복일 경우 - '#'로 시작하지 않는 줄로 포함됩니다. 따라서 앞에서 다룬 10진 수치의 경우 좀 더 정확한 표현은 '[0-9][0-9]'이 됩니다.

2.7 Matching a Specific Number of Sets with \{ and \}

'*'만으로는 최대 횟수를 지정할 수 없습니다. 몇몇 프로그램들에선 다음과 같은 횟수를 지정하는 정규식을 쓸 수 있습니다. 이는 두 개의 쌍으로 이루어진 특수 표현을 사용합니다: '\{'와 '\}'.

예를 들어 소문자 4 개에서 8개로 이루어진 문자열은 [a-z]\{4,8\}에 해당합니다. 횟수를 지정할 때 쓰이는 수치는 0에서 255 사이의 수치면 됩니다. 만약 두

그림 3: Regular Expression Pattern Repetition Examples

Regular Expression	Matches
*	*로 시작하는 줄
*	*로 시작하는 줄
\\	\로 시작하는 줄
^*	*로 시작하는 줄
^A*	그냥 줄 (아무 줄이나 해당)
^A*	A*로 시작하는 줄
^AA*	하나 이상의 A로 시작하는 줄
^AA*B	하나 이상의 A로 시작하고 B로 끝나는 줄
^A\{4,8\}B	4개에서 8개 사이의 A가 나오고 B로 끝나는 줄
^A\{4,\}B	4개 이상의 A와 B로 끝나는 줄
^A\{4\}B	AAAAB로 시작하는 줄
\{4,8\}	{4,8}로 시작하는 줄
A{4,8}	A{4,8}로 시작하는 줄

번째 수치가 생략되면 *와 비슷하게 상위 경계에 적용받지 않습니다. 따라서 특수 문자 는 \{0,\}와 같은 뜻입니다. 와 마찬가지로 이는 앞 패턴의 횟수를 지정하기 위해 쓰입니다. 따라서 이 표현이 문장의 처음에 나온다면 modifier의 기능을 잃고 일반 글자로 쓰이게 됩니다.

2.8 Matching words with \<with \>

단어를 찾는 것은 때때로 복잡한 일이 될 수 있습니다. 예를 들어 'the'를 찾기 위해 the를 쓰더라도 'other'에 매치되는 경우가 있습니다. 이를 방지하기 위해 앞뒤에 공백을 넣는다 하더라도 'the'라는 단어가 줄의 맨 앞 부분이나 뒷부분에 있다면 이 방법으로는 찾을 수 없습니다. *ed*, *ex*, *vi*와 같은 프로그램은 이런 경우 단어(word) 단위로 검색할 수 있게 \<와 \>를 지원합니다. 즉 정규식 \<the\>는 'the'에는 매치되지만 'other'에는 매치되지 않습니다.

2.9 Remembering Patterns with \(, \), and \1

반복되는 단어를 찾을 때, 쓸 수 있는 유용한 기능입니다. 예를 들어 같은 글자가 반복되는, 두 글자로 이루어진 패턴을 찾을 때, [a-z][a-z]는 거의 쓸모가 없습니다. \(와 \)는 이 사이의 패턴을 기억해 줍니다. 실제 이 패턴은 \1로 쓸 수 있습니다. 예를 들어 같은 글자가 반복되는, 두 글자로 이루어진 패턴은 \([a-z]\)\1로 쓸 수 있습니다. 한 정규식에 여러 쌍의 \(와 \)가 있다면 \1, \2, \3등으로 쓸 수 있습니다. 최대 9까지 기억할 수 있습니다.

'Palindrome'이라는 것이 있습니다. 간단히 말해 앞으로 쓰나 뒤로 쓰나 같은 단어를 말합니다. 예를 들면 'radar'가 있습니다. 5글자로 된 palindrome을 찾으려면 정규식 \([a-z]\)\([a-z]\)[a-z]\2\1을 씁니다.

이는 특히 *sed*를 써서 문자열을 치환할 때에도 유용하게 쓰입니다.

2.10 Potential Problem

확장 정규식을 다루기 전에 먼저 알고 넘어갈 것이 있습니다. \<, \> 문자는 *vi*에서 처음 사용된 것입니다. 따라서 다른 프로그램에서는 지원하지 않을 수도 있습

니다. 또한 \{, \}도 새로 등장한 것입니다. 오래된 프로그램에서는 지원하지 않을 가능성이 높습니다. 초보자가 흥미를 잃는 가장 큰 이유는 바로 이런 것 때문입니다. 즉 정규식은 그 종류가 다양하고 또, 100% 지원하는 프로그램이 드물기 때문입니다. 그러나 최근에 만들어진 대부분의 프로그램들은 대부분의 정규식을 지원하기 때문에 사용자의 시스템이 특별히 오래된 것이 아니면 무난하게 쓸 수 있을 것입니다.

또 하나의 문제는 조금 골치가 아픈 것입니다. 예를 들어 `A.*B`는 'AAB'에도 해당하지만 'AAAABBBBABCCCCBBBAAAB'에도 해당합니다. 얼핏 보면 문제가 없을 것 같지만 심각합니다. 예를 들어 위의 긴 문자열은 자체가 `A.*B`에 매치될 수도 있지만 작게 쪼개면 'AAAAB'도 정규식에 해당하고 'AB'도 해당하고, 마지막 'AAAB'도 해당하는 등, 여러 가지 경우가 나옵니다. *grep*과 같은 프로그램은 단순히 주어진 정규식에 해당하는 줄을 출력해 주는 역할을 하므로 별 문제가 되지 않지만 *sed*와 같은 프로그램을 써서 정규식에 해당하는 표현을 다른 것으로 바꾼다고 할 때 문제가 발생합니다. 따라서 정규식을 써서 무언가를 바꾸거나 제거할 때에는 주의해야 합니다. 참고로 이런 경우에 정규식은 항상 가장 긴 문자열에 매치되려고[3]합니다.

2.11 Extended Regular Expressions

*egrep*과 *awk*는 확장 정규식을 쓸 수 있는 프로그램입니다. *perl*은 더욱 확장 정규식을 쓸 수 있습니다. 확장 정규식에서는 \{, \}, \<, \>, \(\), \1등은 쓸 수 없습니다. 그 이유는 이 단원 마지막에 설명합니다.

먼저 두 개의 새 특수 문자를 쓸 수 있습니다. ?는 앞 패턴이 0 또는 1번 나온다는 뜻입니다. 단순 정규식을 쓰면 `\{0,1\}`에 해당합니다. +는 앞 패턴이 1 번 이상 나온다는 뜻입니다. 따라서 `\{1,\}`에 해당합니다.

지금까지 설명만으로는 확장 정규식이 위의 두 가지 ?와 +이 추가된 것만 빼고는 전혀 쓸모없어 보일 것입니다. 아래의 예를 보면 생각이 바뀔지도 모릅니다.

확장 정규식에서 변화된 것은 바로 (, |,) 문자입니다. 이들 문자는 어떤 집합을 정의할 수 있습니다. 즉 정규식 `(A|B|C)`는 A, B, C에 매치되는 표현입니다. 얼핏 보면 `[ABC]`과 같아 보이지만 엄연히 다릅니다. 괄호와 |를 쓸 때의 A, B, C는 단순히 문자가 아니라 A, B, C 자체도 다른 정규식일 될 수 있습니다. 예를 들어 'From: ' 또는 'Subject: '라는 단어가 들어간 줄을 찾으려면 다음과 같이 *egrep*을 쓸 수 있습니다:

```
% egrep '^(From|Subject): ' /usr/spool/mail/$USER
```

만약 단순 정규식을 쓴다면 쉽게 할 방법이 없습니다. 아마도

```
^[FS][ru][ob][mj]e*c*t*:
```

처럼 해야 할 것입니다. 그러나 불안전한 방법입니다. 예를 들어 이 방법을 쓸 경우 'Sromeet: '에도 매치되어버릴 수 있습니다. 확장 정규식에서는 \<, \>도 쓸 수 없습니다. 그러나 단어(word) 단위로 찾기 위해 다른 방법을 쓸 수 있습니다. 단어 'the'만을 찾으려면 `(^| )the([^a-z]|%)`를 쓸 수 있습니다.

예를 들어 다음의 세 가지 경우 모두 매치되는, 확장 정규식을 만들어 봅시다:

- `"a simple problem"`
- `"an easy problem"`

[3] A regular expression tries to match the longest string possible

- "a problem"

이는 다음의 두 가지로 만들어 볼 수 있습니다.

- "a[n]? (simple|easy)? ?problem"
- "a[n]? ((simple|easy))?problem"

후자의 경우가 좀 더 정확한 표현입니다.

단순 정규식에서 \{, \}, \<, \>, \(\), \1가 확장 정규식에서 동작하지 않는 이유를 설명하겠습니다. 사실 \{...\}나 \<...\>는 확장 정규식에 추가될 수도 있지만 \(...\)...는 불가능합니다.

이유는 간단합니다. 만약 (가 특수 문자라면 \(는 일반 문자가 되어야 하기 때문입니다. 이는 단순 정규식과 반대지만 의미는 통합니다. 예를 들어 단순 정규식에서는 (는 일반 문자지만 \(는 특수 문자가 됩니다. 따라서 단순 정규식의 규칙을 지키면서 기능을 추가할 수 없습니다.

따라서 GNU Emacs와 같은 프로그램은 두 기능을 한번에 쓰지 않는다는 가정 아래에 단순 정규식과 확장 정규식을 동시에 지원합니다.

3 Getting Regular Expression Right

아래는 *troff* 문서의 한 부분입니다.

```
.Se "Appendix" "Full Program Listing"
```

이 줄에서 첫번째 인자('"'로 둘러싸인 부분)를 얻어내려면 어떤 정규 표현식을 써야 할까요? 즉 '큰 따옴표로 둘러 싸인 문자열'에 해당하는 정규식을 말하는 것입니다. 정규식을 많이 써 보지 않은 사용자라면 단순하게 다음과 같이 하려 할 것입니다.

```
".*"
```

그러나 이는 매우 잘못된 것입니다. 정규식은 항상 가장 큰 문자열을 먹어 치우려는 습성이 있기 때문에 위의 정규식은 원래 문장에서 아래 부분을 먹어 치울 것입니다.

```
"Appendix" "Full Program Listing"
```

따라서 다음과 같이 써야 올바른 결과를 얻을 수 있습니다:

```
"[^"]*"
```

4 Regular Expression Examples

이 장에서는 정규식을 써서 만든 간단한 예제를 모았습니다.

- 미국 주(State) 표시: (NM)

  ```
  [A-Z][A-Z]
  ```

- 미국 주, 시(city) 표시 (Portland, OR)

```
^.*,␣[A-Z][A-Z]
```

- 미국식 날짜 표시 (JAN 05, 1993) (January 5, 1993)

```
[A-Z][A-Za-z]\{2,8\}␣[0-9]\{1,2\},␣[0-9]\{4\}
```

- 미국 주민번호 (123-45-6789)

```
[0-9]\{3\}-[0-9]\{2\}-[0-9]\{4\}
```

- 전화번호 (547-5800)

```
[0-9]\{3\}-[0-9]\{4\}
```

- 달러 ($1) ($ 1000000.00)

```
\$␣*[0-9]+(\.[0-9][0-9])?
```

- HTML 태그 (<H2>) (<UL COMPACT>)

```
<[^>]*>
```

- 빈 줄

```
^$
```

- 줄 전체

```
^.*$
```

- 하나 이상의 공백

```
␣␣*
```

5 Valid Metacharacters for Difference UNIX Programs

정규식에 쓰이는 여러가지 특수문자들은 프로그램에 따라 지원하는 것이 있고 지원하지 않는 것이 있습니다. 그림 4에 지원 여부가 있으니 참고하시기 바랍니다.

그림 4는 검색에 쓰이는 문자만 도표화한 것이며, 치환(replace)에 쓰이는 문자는 그림 5를 참고하기 바랍니다.

6 Quick Reference

이 장에서는 빠르게 참고할 수 있도록 정규식에 대한 각 표현과 예제를 요약합니다.

특수 문자의 의미에 대한 것은 그림 6를 참고하기 바라며, 치환 문자에 대한 것은 그림 7를 참고하기 바랍니다.

그림 4: Valid Metacharacters for Different Programs

Symbol	ed	ex	vi	sed	awk	grep	egrep	Action
.	√	√	√	√	√	√	√	한 글자
*	√	√	√	√	√	√	√	0개 이상의 앞 패턴
^	√	√	√	√	√	√	√	줄의 첫 부분
$	√	√	√	√	√	√	√	줄의 끝 부분
\	√	√	√	√	√	√	√	특수 기능
[]	√	√	√	√	√	√	√	글자 범위 지정
\(\)	√	√		√				패턴 저장
\{\}	√			√		√		패턴 범위 지정
\<\>	√	√	√					단어(word) 단위
+					√		√	하나 이상의 앞 패턴
?					√		√	0,1개의 앞 패턴
|					√		√	선택 (OR)
()					√		√	그룹짓기

그림 5: Valid Metacharacters for Replacement Patterns

Symbol	ex	sed	ed	Action
\	√	√	√	특수 문자 전환
\n	√	√	√	\(\)에 저장된 패턴 쓰기
&	√	√		이전 검색 패턴 쓰기
~	√			이전 치환 패턴 쓰기
\u\U	√			대문자로 전환
\l\L	√			소문자로 전환
\E	√			\U, TT\L 끄기
\	√			\u, TT\l 끄기

그림 6: Special Chartacters in Search Patterns

Pattern	What Does it Match?
.	아무런 한 글자에 해당
*	앞 패턴이 0개 이상 나옴
^	뒤따르는 패턴이 줄의 첫 부분에 해당
$	앞 패턴이 줄의 마지막에 해당
[]	집합에 속하는 한 글자에 해당, 범위를 지정하기 위해 -를 쓸 수 있고, 집합에 속하지 않는 글자를 나타내기 위해 ^를 [바로 다음에 쓸 수 있음
\{n,m\}	앞 패턴의 반복 횟수를 n 번 이상, m번 이하로 지정함. n 또는 m이 생략될 수 있음. 이 때에는 각각 최소값, 최대값이 없는 상태임. 만약 ',m'이 생략되면 정확히 n번 반복을 뜻함
\	뒤따르는 문자의 특수 기능을 제거
\(\)	두 쌍의 사이에 있는 패턴을 저장함, 나중에 \n으로 불러 쓸 수 있음, 이 때의 'n'은 1에서 9 사이의 숫자.
\<\>	두 쌍 사이에 있는 패턴을 단어(word) 단위로 간주.
+	앞 패턴의 하나 이상 반복을 의미
?	앞 패턴이 0 또는 1개인 것을 의미
|	()에서 각 패턴의 OR 상태를 나타내 줌
()	패턴 그룹을 만듬 예를 들어 패턴 (A|B)는 A 또는 B를 의미, 이 때 A, B는 각각 다른 패턴이 될 수 있음.

그림 7: Special Characters in Replacement Patterns

Pattern	What Does it Match ?
\	다음 글자의 특수 의미를 제거
\n	\(\)로 저장된 패턴을 사용, 이 때 n은 1-9의 정수
&	검색 패턴을 사용
~	바로 전 치환 패턴을 사용
\u	치환 패턴의 첫 글자를 대문자로
\U	치환 패턴을 대문자로
\l	치환 패턴의 첫 글자를 소문자로
\L	치환 패턴을 소문자로

그림 8: Search Pattern Examples

Pattern	What Does it Match?
bag	문자열 'bag'.
∧bag	첫 단어가 'bag'인 줄.
bag$	마지막 단어가 'bag'인 줄.
∧bag$	내용이 'bag'인 줄.
[Bb]ag	'Bag' 또는 'bag'.
b[aeiou]g	가운데 문자가 모음인 'bag'.
b[∧aeiou]g	가운데 문자가 모음이 아닌 'bag'.
b.g	가운데 문자가 아무 글자라도 좋은 'bag'.
∧...$	세글자로 이루어진 줄.
∧\.	Dot (.)으로 시작하는 줄.
∧\.[a-z][a-z]	Dot으로 시작, 소문자 두개로 된 단어가 있는 줄.
∧\.[a-z]\{2\}	위와 같음. *grep*과 *egrep*에서만 가능.
∧[∧.]	Dot으로 시작하지 않는 줄.
bugs*	'bug', 'bugs', 'bugsss' 등에 해당.
"word"	큰 따옴표로 둘러 싸인 'word'.
"?word"?	큰 따옴표로 둘러 싸이거나 그렇지 않은 'word'.
[A-Z][A-Z]*	대문자로 이루어진 단어.
[A-Z]+	위와 동일, *grep*과 *egrep*에서만 가능.
[A-Z].*	첫 글자가 대문자인 단어.
[A-Z]*	0개 이상의 대문자.
[a-zA-Z]	알파벳.
[∧0-9A-Za-z]	알파벳, 숫자가 아닌 글자.
[567]	5, 6, 또는 7.
grep 또는 *egrep* 패턴:	
five|six|seven	'five', 'six' 또는 'seven'.
80[23]?86	'8086', '80286' 또는 '80386'.
compan(y|ies)	'company' 또는 'companies'.
ex 또는 *vi* 패턴:	
\<the	'theater'나 'the'와 같은 단어.
the\>	'breathe'나 'the'와 같은 단어.
\<the\>	'the'.
sed 또는 *grep* 패턴:	
0\{5,\}	5개 이상의 '0'.
[0-9]\{3\}-[0-9]\{2\}-[0-9]\{4\}	미 주민 번호 (*xxx-xx-xxxx*).

그림 9: Search and Replace Commands

Command	Result
s/.*/(&)/	모든 줄을 ()로 둘러 쌈.
s/.*/mv␣&␣&.old/	모든 줄의 내용을 'mv 줄 줄.old'으로 바꿈.
/^$/d	빈 줄을 제거.
:g/^$/d	위와 동일 *ex*용.
/^[␣tab]*$/d	빈 줄, 공백 문자로만 된 줄을 삭제.
:g/^[␣tab]*$/d	위와 동일 *ex*용.
s/␣␣*/␣/g	하나 이상의 공백을 한 공백으로 바꿈.
:%s/␣␣*/␣/g	위와 동일 *ex*용.
:s/[0-9]/Item␣&:/	숫자 'n'을 'Item n'으로 변경. (현재 줄).
:s	첫 대상에 치환 적용.
o :&	위와 동일.
:sg	위와 동일 (단 줄 전체에 적용).
&g	위와 동일.
%&g	위와 동일 (단 모든 줄 전체에 적용).
:.,$s/Fortran/\U&/g	현재 줄에서 끝까지 'Fortran'을 찾아 'FORTRAN'으로 변경.
:%s/.*/\L&/	파일 전체를 소문자로 변경.
:s/\<./\u&/g	현재 줄의 각 단어의 첫글자를 대문자로.
:%s/yes/No/g	파일 전체에서 'yes'를 'No'로 변경.
:%s/Yes/~/g	파일 전체에서 'Yes'를 찾아 이전 치환 문자('No')로 변경.

6.1 Examples of Searching

정규식을 *grep*이나 *egrep* 명령과 함께 쓰면 다양한 검색 효과를 얻을 수 있습니다. Shell 와일드카드 확장을 막기 위해 모든 패턴은 ''로 둘러 싸는 것이 좋습니다. 각 패턴 예제는 그림 8를 참고하기 바랍니다.

6.2 Examlpes of Searching and Replacing

그림 9를 보면 *sed*나 *ex* 명령을 써서 검색-치환할 수 있는 예가 있습니다. ':'으로 시작하는 예가 *ex* 명령입니다. 예제에서 공백은 ␣로 표기하고 탭은 tab으로 표기합니다.