『리눅스 학당-리눅스 강좌 / 연재 (go LINUX)』 461번
제 목:정규표현식 프로그래밍 강좌 ..
올린이:엠브리오(유형목)   97/05/25 10:34   읽음:4754 관련자료 없음

하이텔 시삽님이신 한동훈님의 정규표현식 프로그래밍 강좌입니다.

GNU REGEX (정규표현식) 프로그래밍 강좌 (1)

#615   한동훈   (ddoch   )
[강좌] Regex (정규표현식) 라이브러리 (1)   05/24 01:03   401 line

글쓴이   : 한동훈 ddoch@hitel.kol.co.kr
날 짜   : 1997.5.23.
저작권   : 상업적인 용도가 아닌한 어디로든 이동 및 게재 가능
부탁사항 : 질문과 관련된 내용이나 답변을 요하는 내용은 메일로 적어주지 마시고 관련 프로그래밍 게시판을 이용해 주시면 성의껏 답변해 드리겠습니다. 제가 상당히 게으른 관계로 질문메일에 제대로 답변을 못해드리고 있는 점 죄송합니다.

목차

1. 들어가는 말

안녕하세요. ddoch 한동훈 입니다.

flex 와 bison 을 공부하던 중 regex에 대한 정리가 필요하다고 생각하여 regex 메뉴얼 (자료실/문서/2027번 문서/regex.zip)로 공부를 하던 중 그냥번역하는 것 보다는 실제 예를 들어가면서 설명하는 것이 좋을 것 같아 이렇게 강좌로 올립니다.

이 강좌 내용은 위 메뉴얼을 중심으로 설명을 드릴 예정이며 순서 또한 6. Regex 프로그래밍 중 난이도에 따라 비교적 쉬운 "BSD Regex 함수" 부터 설명을 하였다는 점을 제외하고는 같습니다. 그리고 중간중간 이해를 돕기위해 예제로 설명을 하겠으며 설명내용 중 잘못된 것이나 참고사항은 위의 주소로 보내주시면 감사하겠습니다. 그리고 설명 중 모자라는 부분이나 빠진 부분이 있을 수도 있으므로, 위에서 말씀드린 메뉴얼을 참고하시기 바라겠습니다.

그리고 6장을 제외한 나머지 내용은 일반 리눅스 및 유닉스 사용자들이 참조할 수 있는 공통적인 내용이며 6장은 프로그래머를 위한 내용입니다.

regex는 정규표현식을 이용하여 패턴 검색 및 매칭 작업을 수행하는 함수를 제공하는 일종의 라이브러리입니다. Linux 나 UNIX에서 이 라이브러리는 아주 광범위하게 사용되어 사용자 수준에서의 정규표현식을 사용하여 강력한 작업을 할 수 있도록 도와주는 역할을 합니다. 아마도 리눅스나 유닉스를 사용하시면서 쉘상에서,

```
queen:~$ xanim [a-g]*.gif
```

이런 명령을 사용해보신 분들이 많이 계실 겁니다.

표준 쉘만이 아니라, ed, ex, sed, awk, vi, emacs, grep, egrep등의 유닉스 표준 유틸리티들과 postgres, bison, flex등의 툴 들에서도 내부적으로 사용을 하며, 프로그램을 직접 설치해보신 분은 소스디렉토리안에 "regex.h, regex.c"라는 파일이 들어 있는 경우를 종종 보셨을 겁니다.

이런 정규표현식은 bison, flex에서도 사용된다고 하였는 데, 이것은 각종 문자열 파싱이나 문장, 구문해석에 사용되어 컴파일러 제작, 어문해석기 등의 프로그램을 만드는 데 사용됩니다.아직 정규표현식에서 "[가-나]"와 같은 한글을 처리하지 못하고 있는 데, 이런 문제는 한글어휘분석기 및 한글토큰분석에 난제로 등장하고 있는 관계로 이의 해결은 우리들의 몫이 아닐까 생각합니다.

설치는, 리동 자료실에 있는 regex 0.12 버젼 (자료실/2370번/regex012.tgz)을 받으셔서 root로 압축을 푸시고 "./configure; make; make install"로 설치를 하시면 됩니다. 네트웍에서 구하실려면 GNU 공식 사이트나 한국에서 미러를 하고 있는 카이스트에서 "regex" 로 검색하시면 찾을 수 있습니다.

말이 설치지, 설치되는 것은 info 파일과 texi 파일을 컴파일하여 해당디렉토리로 이동시키는 것일 뿐입니다. 압축을 푼 디렉토리에 보시면 regex.c 와 regex.h가 있는 데, 이 두개가 전부이므로 휴대를 하시면서 사용하시던지, 아니면 regex.o 파일로 링크만 시키시던지는 마음대로 하시면 될 것입니다. 참고로 슬랙 3.1에 "/usr/include"에도 "regex.h"가 있으나 본 헤더파일과는 다르므로 인클루드 하실 때 주의하시기 바랍니다. 테스트 파일은 "test/" 디렉토리에 있으므로 살펴보시면 도움이 될 것이며, 테스트 소스 컴파일은 "test/" 디렉토리에서 "make all" 로 하시면 됩니다.

"regex.h" 파일에 아주 자세한 설명이 들어 있으므로 자주 참고를 하시기 바라며, 한부 뽑아서 보셔도 좋습니다.

정규표현식을 이용하여 프로그램을 짜시려는 분들이나 정규표현식을 익히시려는 분들에게 조금이나마 도움이 되었으면 좋겠습니다.

정규표현식을 이용한 프로그램인 egrep을 이용하여 소스내에서 특정 토큰(예: int)을 찾는 경우를 예를 들어보겠습니다.

```
queen:~$ egrep int something.c
                ...
```

이런식으로 찾으면 "printf" 도 같이 검색이 되므로 요구를 채워주지 못합니다.

```
queen:~$ egrep "[^[:alnum:]_]int[^[:alnum:]_]" something.c
```

이제, 하나의 독립된 토큰으로서의 "int"만 찾아서 우리에게 보여줍니다.

만일, egrep 같은 프로그램을 짤 때, 첫번째 인자(정규표현식)를 일일이 C로 파싱하여 처리하는 것은 거의 사람의 인내성의 한계를 실험하는 것이 될 것입니다. 이럴 때 미리 짜놓은 regex 함수를 이용하여 해당 펑션에서 첫번째 인자와 해당파일을 읽은 문자열을 넘겨주면 알아서 검색 및 패턴 매칭을 해주므로 아주 간편하게 프로그래밍 할 수 있는 것입니다.

정규표현식에도 상당히 많은 형태의 문법이 있다는 것은 천천히 보여드리도록 하겠습니다. 그리고 강좌 마지막에 가능하다면, 정규표현식을 이용하는 간단한 기능의 egrep 버젼을 만들어 보도록 하겠습니다.

자, 그럼 이제 설명에 들어가볼까요..

2. 정규표현식 문법

정규표현식은 어떤 문자열의 집합을 묘사하는 텍스트 스트링입니다.
오퍼레이터는 '['나 '*'같은 한개 이상의 문자와 매칭되는 정규표현식안에 있는 문자입니다.
일반적으로 대부분의 문자는 'a'나 'z'와 같이 그 자체로서의 문자그래로의 뜻을 가집니다. 이것을 여기서는 "그냥문자(ordinary) 또는 일반문자"라고 하겠습니다. 이와는 반대로 '.'와 같이 특수한 뜻을 나타내는 문자를 "특수문자(special)"라고 부르겠습니다.
어떤 문자가 특수문자인지 또는 그냥문자인지는 다양한 정규표현식의 문법과 해당 정규표현식에서의 문맥에 따라 달라집니다. 이제, 아래에서 자세하게 이야기 하겠습니다.

2.1 문법 비트

정규표현식에서 어떤 특정한 문법은 몇몇의 문자들을 항상 특수문자로 취급하고, 다른 문법은 가끔 특수문자로 취급하며, 또다른 문법은 그러한 문자들을 일반문자로 취급할 경우가 있습니다.

주어진 정규표현식에서 Regex가 인식하는 특정한 문법은 해당 정규표현식의 패턴 버퍼의 syntax 필드에 따라 다릅니다. 이 말은 위의 예에서 정규표현식 중에서 "[:alpha:]"같은 것들이 이 패턴을 다루는 버퍼중에서 syntax 필드에 따라 틀린 문법으로 치부될 수도 있고, 그냥 무시하고 넘어갈 수도 있으며, 올바르게 작동할 수도 있다는 이야기입니다. 따라서 syntax 필드를 조정해줌으로써 정규표현식의 기능을 다양하게 제한하고 확장할 수 있다는 이야기가 되겠네요.

패턴 버퍼는 "[a-g]*"와 같은 정규표현식을 뒤에서 설명하는 정규표현식 "컴파일" 함수에 인자로 넘겨줌으로 만들수 있습니다.

(참고로, 여기서 "컴파일"이라함은, 텍스트 스트링 형태의 정규표현식을 검색,매칭 할수 있는 형태로 만들기 위해 어떤 버퍼(패턴 버퍼)에 번역을 하거나 이에 필요한 각종 값을 담아두는 역할을 하는 것을 이야기합니다.)

syntax 필드는 다양한 비트들의 조합으로 구성되며, 이러한 비트들을 보통, "문법 비트"라고 부릅니다. 이러한 문법 비트는 "어떤 문자가 어떤 오퍼레이터가 될것인가"하는 문제를 결정하게 됩니다.

이제, 문법 비트의 모든 것을 알파벳 순서로 설명을 드리겠습니다. 참고적으로, 이것은 "regex.h"에 자세히 설명되어 있는 것으로 "RE_"로 정의되어 있습니다.

언뜻 정의된 이름만으로도 그 기능을 충분히 예견할 수 있을 것입니다.

* RE_BACKSLASH_ESCAPE_IN_LISTS (리스트에서 백슬래쉬는 이스케이프)

  일반적인 리스트 오퍼레이터인 '[', ']'안에서 '\'(이스케이프)문자는 뒷글자를 이스케이프하는 탈출문자가 된다는 이야지이지요. 만일 이 비트가 세팅되지 않으면 리스트 오퍼레이터안에서의 '\'는 그냥문자(=일반문자)가 됩니다.

  보통, 리스트 오퍼레이터 안의 문자는 특수문자 성격을 상실하고 그냥문자가 되는 게 일반적입니다.

* RE_BK_PLUS_QM ('\+', '\?')

  이 비트가 설정되면 '\+'는 "하나이상을 매칭시키는 오퍼레이터(이후 하나이상 오퍼레이터)(match-one-or-more operator)"가 되며, '\?'는 "0개 이상을 매칭시키는 오퍼레이터 (이후 뻥개이상 (^^;) 오퍼레이터)"(match-zero-or-more operator)이 됩니다. 이 비트가 설정되지 않으면, 각각 '+'와 '?'가 그 역할을 대신합니다.

  일반적으로는 보통, '+', '?'가 각각 하나이상, 0개 이상을 매칭시키는 오퍼레이터로 작동을 합니다.
  만일 RE_LIMITED_OPS 가 세팅되었다면 이 비트는 세팅하지 마셔야 합니다.

* RE_CHAR_CLASSES (문자 클래스)

  이 비트가 세팅되어 있으면 리스트안에서 문자클래스 오퍼레이터를 사용할 수 있으며 그렇지 않으면 사용할 수 없습니다.

  위에서 예를 든, egrep 의 경우에는 리스트안([..])에서 문자 클래스 ([:alnum:])을 사용할 수 있었으므로 이 비트가 세팅되어 있다는 것을 미루어 짐작할 수 있습니다.

* RE_CONTEXT_INDEP_ANCHORS

  이 비트가 세팅되어 있다면, '^'와 '$'는 리스트 밖에서의 어디에서나 특수문자로 취급하며, 그렇지 않다면 확실한 문맥에서만 특수문자로 취급합니다.

* RE_CONTEXT_INDEP_OPS

이 비트가 세팅되어 있으면, 리스트 밖에서 어디서던지 "확실한 문자"들은 특수 문자로 취급됩니다. 그렇지 않으면 그러한 문자들은 단지 어떤 문맥에서만 특수 문자이고 다른 곳에서는 그냥문자로 취급됩니다. 특히, 이 비트가 세팅되지 않은 상태의 '*' 와 RE_LIMITED_OPS가 설정되지 않았을 때의 '+'와 '?'(또는 RE_BK_PLUS_QM이 설정되었을 때의 '\+', '\?')는, 정규표현식의 처음(예:*foo) 이나 오픈그룹연산자('(')나 대체 연산자('|')의 바로뒤(예: (*.., |*)에 오지 않을 때에만 이것을 반복 오퍼레이터로 취급합니다.

* RE_CONTEXT_INVALID_OPS

이 비트가 세팅되어 있다면, 반복오퍼레이터('*')와 대체오퍼레이터('|')는 정규표현식 내부에서 "확실한 위치"에는 올수 없게 됩니다. 특히, 다음과 같은 경우에는 정규표현식이 잘 못된 경우입니다.

O 반복 오퍼레이터가 다음의 위치에 올경우
  - 정규표현식의 처음에 올경우 (예: '*[a-z]')
  - 라인의 시작 오퍼레이터 ('^')나 오픈 그룹 ('(')이나 대체 오퍼레이터('|') 의 바로뒤에 오는 경우 (예: '^*', '(*..)', '|*')

o 대체 오퍼레이터가 다음의 위치에 올경우
  - 정규표현식의 처음이나 마지막에 올경우 (예: '|foo', 'foo|')
  - 라인의 끝 오퍼레이터 ('$')의 바로 전이나, 대체오퍼레이터, 오픈 그룹 오퍼레이터의 바로뒤에 올경우 (예: '|$', '||', '(|')

만일, 이 비트가 세팅되어 있지 않다면, 정규표현식의 어디에서던지 반복 오퍼레이터와 데체 오퍼레이터가 올 수 있게 됩니다.

* RE_DOT_NEWLINE (점 '.'은 뉴라인을 포함)

이 비트가 세팅되어 있다면, "아무거나한문자 오퍼레이터(match-any-character operator)" ('.')는 뉴라인문자와 매칭될 수 있습니다. 세트되어 있지 않다면 '.' 는 뉴라인문자('\n') 와 매칭될 수 없습니다.

* RE_DOT_NOT_NULL (점 '.'은 널이 될 수 없다)

이 비트가 세팅되어 있다면, 아무거나한문자 오퍼레이터는 널문자와 매칭될 수 없으며, 세트되어 있지 않다면 가능합니다.

* RE_INTERVALS (간격)

이 비트가 세트되어 있다면 Regex는 "간격오퍼레이터(interval operators)" ('{', '}')를 인식할 수 있고, 그렇지 않다면 불가능합니다.

* RE_LIMITED_OPS (오퍼레이터 제한)

이 비트가 세팅되어 있다면, Regex는 하나이상 오퍼레이터('+'또는 '\+')와 뻥개이상 오퍼레이터('*')는 인식을 하지 못하며, 세팅되어 있지 않다면, 가능합니다.

* RE_NEWLINE_ALT (뉴라인 대체)

이 비트가 세팅되어 있다면, 뉴라인은 대체 오퍼레이터로 취급되며, 그렇지 않다면 뉴라인문자는 그냥문자가 됩니다.

* RE_NO_BK_BRACES (백슬래쉬 없는 중괄호)

이 비트가 세트되어 있다면, '{'는 오픈 인터벌(open-interval)오퍼레이터가 되고, '}'는 클로즈 인터벌(close-interval) 오퍼레이터가 됩니다. 그렇지 않다면, '\{'와 '\}'가 각각 그역할을 대신합니다. 이 비트는 RE_INTERVALS가 세트되어 있을 때에만 상관있습니다.

* RE_NO_BK_PARENS (백슬래쉬 없는 소괄호)

이 비트가 세트되어 있다면 '('는 오픈 그룹 오퍼레이터가 되고, ')'는 클로즈 그룹 오퍼레이터가 됩니다. 만일 이 비트가 세트되어 있지 않다면, '\('와 '\)'가 각각 그역할을 대신합니다.

* RE_NO_BK_REFS (거꾸로참조 (^^;) 오퍼레이터 인식안함)

  이 비트가 세트되어 있다면, Regex는 '\'digit 와 같은 거꾸로참조 오퍼레이터
  를 인식하지 않습니다. 그렇지 않다면 인식합니다.

* RE_NO_BK_VBAR (백슬래쉬 막대기 ^^;를 인식안함)

  이 비트가 세트되어 있다면 '|'가 대체오퍼레이터로 되고, 세트되어 있지 않다면
  ,'\|'가 대체오퍼레이터로 됩니다. 이 비트는 RE_LIMITED_OPS 가 세트되었다면
  상관없습니다.

* RE_NO_EMPTY_RANGES (비어있지 않는 범위)

  이 비트가 세트되어 있다면, 정규표현식에서 잘못된 범위지정(예:'[z-a]')
  시에는 틀린게 됩니다. 비트가 설정되어 있지 않다면, Regex는 그 범위를 단
  지 텅비게 만듭니다.

* RE_UNMATCHED_RIGHT_PAREN_ORD (빠진 오른쪽 괄호)

  이 비트가 세트되었고, 정규표현식에서 오픈그룹 오퍼레이터('(')가 클로즈
  그룹 오퍼레이터와 짝이 맞지 않는다면 그냥 넘어가나, 다른 경우네는 ')'를
  찾게 됩니다.

휴..이제 설명을 다했군요.. 무슨 뜻인지는 짐작이 가실겁니다.

이제 이러한 문법 비트들이 모여 어떻게 표준 응용프로그램마다 조금씩 다르게
적용되는 지 살펴보지요.

2.2 미리 정의된 문법

  이번에 살펴볼 것은 "regex.h" 에서 정의된 중요 응용 프로그램의 문법 스타일
을 정의해둔 부분입니다. 여기서 기준이 되는 프로그램은, GNU Emacs, POSIX Awk,
traditional Awk, Grep, Egrep 등이며, POSIX 기본과 확장 정규표현식이 정의됩
니다.

```
#define RE_SYNTAX_EMACS 0

#define RE_SYNTAX_AWK                   \
  (RE_BACKSLASH_ESCAPE_IN_LISTS | RE_DOT_NOT_NULL     \
   | RE_NO_BK_PARENS            | RE_NO_BK_REFS       \
   | RE_NO_BK_VBAR              | RE_NO_EMPTY_RANGES     \
   | RE_UNMATCHED_RIGHT_PAREN_ORD)

#define RE_SYNTAX_POSIX_AWK             \
  (RE_SYNTAX_POSIX_EXTENDED | RE_BACKSLASH_ESCAPE_IN_LISTS)

#define RE_SYNTAX_GREP                  \
  (RE_BK_PLUS_QM              | RE_CHAR_CLASSES       \
   | RE_HAT_LISTS_NOT_NEWLINE | RE_INTERVALS        \
   | RE_NEWLINE_ALT)

#define RE_SYNTAX_EGREP                 \
  (RE_CHAR_CLASSES        | RE_CONTEXT_INDEP_ANCHORS      \
   | RE_CONTEXT_INDEP_OPS | RE_HAT_LISTS_NOT_NEWLINE      \
   | RE_NEWLINE_ALT   | RE_NO_BK_PARENS       \
   | RE_NO_BK_VBAR)

#define RE_SYNTAX_POSIX_EGREP           \
  (RE_SYNTAX_EGREP | RE_INTERVALS | RE_NO_BK_BRACES)

/* P1003.2/D11.2, section 4.20.7.1, lines 5078ff.  */
#define RE_SYNTAX_ED RE_SYNTAX_POSIX_BASIC
```

```
#define RE_SYNTAX_SED RE_SYNTAX_POSIX_BASIC

/* POSIX 기본문법과 확장문법에서 공통되는 문법 */
#define _RE_SYNTAX_POSIX_COMMON           \
  (RE_CHAR_CLASSES ¦ RE_DOT_NEWLINE      ¦ RE_DOT_NOT_NULL    \
   ¦ RE_INTERVALS  ¦ RE_NO_EMPTY_RANGES)

#defineRE_SYNTAX_POSIX_BASIC           \
  (_RE_SYNTAX_POSIX_COMMON ¦ RE_BK_PLUS_QM)

/* Differs from ..._POSIX_BASIC only in that RE_BK_PLUS_QM becomes
   RE_LIMITED_OPS, i.e., \? \+ \¦ are not recognized.  Actually, this
   isn't minimal, since other operators, such as \`, aren't disabled.  */
#define RE_SYNTAX_POSIX_MINIMAL_BASIC         \
  (_RE_SYNTAX_POSIX_COMMON ¦ RE_LIMITED_OPS)

#define RE_SYNTAX_POSIX_EXTENDED          \
  (_RE_SYNTAX_POSIX_COMMON ¦ RE_CONTEXT_INDEP_ANCHORS     \
   ¦ RE_CONTEXT_INDEP_OPS  ¦ RE_NO_BK_BRACES        \
   ¦ RE_NO_BK_PARENS       ¦ RE_NO_BK_VBAR        \
   ¦ RE_UNMATCHED_RIGHT_PAREN_ORD)

/* Differs from ..._POSIX_EXTENDED in that RE_CONTEXT_INVALID_OPS
   replaces RE_CONTEXT_INDEP_OPS and RE_NO_BK_REFS is added.  */
#define RE_SYNTAX_POSIX_MINIMAL_EXTENDED        \
  (_RE_SYNTAX_POSIX_COMMON  ¦ RE_CONTEXT_INDEP_ANCHORS      \
   ¦ RE_CONTEXT_INVALID_OPS ¦ RE_NO_BK_BRACES       \
   ¦ RE_NO_BK_PARENS        ¦ RE_NO_BK_REFS       \
   ¦ RE_NO_BK_VBAR      ¦ RE_UNMATCHED_RIGHT_PAREN_ORD)
```

2.3 백슬래쉬 문자

'\'문자는 4가지의 서로 다른 뜻을 가지고 있습니다. 그 의미는 현재의 문맥과 어떤 문법 비트가 세트되어 있는가에 따라 다릅니다. 그 뜻은 1) 그냥문자, 2) 다음문자를 인용하는 역할, 3) 오퍼레이터를 도입하는 의미, 4) 아무뜻 없음 의 의미중의 하나가 됩니다.

1) 문법 비트가 RE_BACKSLASH_ESCAPE_IN_LISTS 가 세트되지 않은 상태에서 리스트안에 있을 때는 일반문자가 됩니다. 예를 들어, '[\]'는 '\'과 매칭이 됩니다.

2) 아래에 설명하는 두가지 중의 하나로 사용될 때에는 다음 글자를 이스케이프하게 됩니다. 물론 다음글자가 특수문자이면 일반문자의 의미를 가지게 합니다.

   * 리스트의 밖에 있을 때
   * 리스트의 안에 있고 문법비트가 RE_BACKSLASH_ESCAPE_IN_LISTS가 세트되어 있을 때

3) 어떤 특정한 문법비트가 세트되고 확실한 일반문자가 뒤따라 올때 그것은 오퍼레이터를 전개하는 역할을 합니다. 위에서 설명한 RE_BK_PLUS_QM, RE_NO_BK_BRACES, RE_NO_BK_VAR, RE_NO_BK_PARENS, RE_NO_BK_REF를 참조하세요.

   *'\b' 는 단어에서의 경계를 짓는 것과 매칭되는 오퍼레이터입니다.
   * '\B' 는 단어내부와 매칭되는 오퍼레이터입니다.
   * '\<' 는 단어의 시작과 매칭되는 오퍼레이터입니다.
   * '\>' 는 단어의 끝과 매칭되는 오퍼레이터입니다.
   * '\w' 는 단어의 구성과 관련되는 오퍼레이터입니다.
   * '\W' 는 비단어 구성과 관련되는 오퍼레이터입니다.
   * '\`' 는 버퍼의 시작과 매칭되는 오퍼레이터입니다.
   * '\'' 는 버퍼의 끝과 매칭되는 오퍼레이터입니다.
   * Regex가 emacs 심볼로 정의된 상태로 전처리되어 컴파일된다면, '\sclass' 는 문법상의 클래스와 매칭되는 오퍼레이터를 나타내고, '\Sclass'는

문법상 비 클래스 오퍼레이터를 나타냅니다.

4) 다른 모든 경우에, Regex 는 '\'를 무시합니다. 예를 들자면, '\n'은 'n'
과 매칭됩니다.

(다음시간에는 우리가 일반적으로 사용하는 지금까지 설명한 오퍼레이터에
대해서 자세하게 알아보겠습니다.)

『 리눅스 학당-리눅스 강좌 / 연재 (go LINUX)』 462번
제 목:정규표현식 프로그래밍 강좌 [02]
올린이:엠브리오(유형목) 97/05/26 15:15 읽음:1981 관련자료 없음

한동훈님의 정규표현식 라이브러리 강좌입니다.
하이텔 리눅스동 에서 퍼온 것입니다.

#616 한동훈 (ddoch)
[강좌] Regex (정규표현식) 라이브러리 (2) 05/26 01:36 407 line

GNU REGEX (정규표현식) 라이브러리 강좌 (2)
---

3. 공통적인 오퍼레이터

오퍼레이터라 함은 앞서도 말씀드렸지만 정규표현식에서 사용하는 '*' 나 '[' 같은
것을 말합니다. 정규표현식을 지원하는 awk, sed, vi, emacs에서 이런 기능을 사
용해보신 분은 얼마나 편리하고 강력한 기능을 제공하는 지 충분히 경험해보셨을
겁니다. 사실 유닉스는 텍스트 처리에서 탁월한 능력을 보여주고 있고, 유닉스의
이런 장점을 따온 리눅스도 마찬가지로 지원을 하는 기능입니다. 따라서, 정규
표현식에 대한 기본적인 지식은 반드시 익혀두시는 것이 좋습니다. 한두군데의
응용프로그램이 아니라 거의 모든 텍스트 처리 프로그램들은 정규표현식을 이용
하는 텍스트 패턴 매칭을 수행하기 때문입니다.

일반적으로 vi에서 다음과 같은 명령을 많이 사용하실 것입니다.
아래와 같은 데이터베이스가 있다고 가정하겠습니다. 여기에서 앞부분의 우편번
호부분만을 문서내에서 삭제하고 싶다고 하면 다음과 같이 간단하게 할 수 있
습니다.

```
100-011 서울시 중구 충무로1가                    02 충무로1가
100-012 서울시 중구 충무로2가                    02 충무로2가

:%s/^[0-9]*-[0-9]* //
    ..............
```

밑에 '..' 된 부분이 정규표현식이고, 정규표현식은 오퍼레이터의 집합으로 구성
됩니다. 대체로, 오퍼레이터들은 하나만으로 된 것들(예: '*')과 '\'다음에 한글
자가 따라오는 형태로 되어 있습니다. 예를 들면, '('나 '\('는 오픈그룹 오퍼레
이터입니다. (물론 이것은 문법 비트가 RE_BK_PARENS가 세팅되어 있는 가에 따라
달라집니다.)

대부분의 오퍼레이터는 리스트 ('[', ']')안에서는 그 특수한 의미를 상실합니다.

그럼, 이제 각각의 오퍼레이터들을 하나씩 살펴보도록 하겠습니다.

3.1 자신을 매칭시키는 오퍼레이터 (그냥문자 또는 일반문자)

이것은 그냥 일반문자를 말합니다. 'f'는 'f'와 매칭되지 'ff'와 매칭되지는 않습
니다.

3.2 아무거나한문자 오퍼레이터 (.)

'.'은 아무런 문자 한개와 매칭됩니다. 단, 특수한 경우로 다음과 같은 경우에

해당문자는 매칭될 수 없습니다.

뉴라인문자 : 문법비트가 RE_DOT_NEWLINE이 세팅되어 있지 않을때
널   : 문법비트가 RE_DOT_NOT_NULL 이 세트되어 있을 때

예) 'a.b'는 'acb', 'a.b', 'azb'등과 매칭됩니다.

3.3 연결 오퍼레이터

이 오퍼레이터는 두개의 정규표현식, a와 b를 연결합니다. 즉, 'ab'는 'a'다음에 바로 'b'가 따라오는 것을 나타내는 것으로, 정규표현식 'ab'는 정규표현식 'a'와 'b'를 연결한 것입니다. 따라서, 사실 연결 오퍼레이터는 개념적으로만 있을 뿐이지 어떤 형태는 띄고 있지 않습니다. 굳이, 형태를 나타낸다고 하면, 'ab'중 'a'와 'b'사이의 빈문자(empty character)가 연결 오퍼레이터라고 할 수 있습니다.

3.4 반복 오퍼레이터

반복 오퍼레이터는 정규표현식 중 어떤 표현식의 형태를 반복적으로 나타내는 데 사용되는 것으로, 일반적으로 '*'(뻥개이상매칭), '+'(한개이상매칭), '?'(뻥개나 한개매칭), '{', '}'(특정한 반복 횟수 지정-간격오퍼레이터)가 있습니다.

3.4.1 뻥개이상 매칭 오퍼레이터 (*) (match-zero-or-more operator)

이 연산자는 해당 스트링을 정규표현식으로 매칭시키기 위해 가능한한 가장적은 반복횟수(0를 포함하여)를 선택합니다. 가령, 예를 들면, 'o*' 는 "0개 이상으로 구성된 o"를 매칭합니다. 'fo*'는 'fo'의 반복이 아니라 'o'의 반복을 나타냅니다. 따라서, 'fo*'는 'f', 'fo', foo'등과 매칭됩니다. 다음과 같은 경우에는 반복 오퍼레이터의 역할을 수행하지 않습니다.

* 정규표현식의 처음에 올 경우 ('*foo')
* 라인의 시작과 매칭되는 '^'나, 오픈그룹 '('나, 대체 오퍼레이터인 '|' 바로 다음에 위치할 경우 ('^*', '(*foo)', 'foo|*bar')

위의 경우에 아래의 3가지 다른 일이 일어날 수 있습니다.

* 문법비트가 RE_CONTEXT_INVALID_OPS 가 세트되었다면, 그 정규표현식은 틀린것으로 취급됩니다.
* RE_CONTEXT_INVALID_OPS 가 세트되지 않았고, RE_CONTEXT_INDEP_OPS가 세트되었다면, '*'는 반복 오퍼레이터 역할을 수행합니다.
* 다른경우는, '*'는 그냥문자(일반문자)입니다.

'*' 의 작동원리를 예로 들어보겠습니다.

'ca*ar' 이라는 정규표현식으로 'caaar' 이라는 문자를 매칭 시킨다고 한다면, 'ca*ar' 의 'a*' 는 'caaar'의 'aaa'를 매칭시킵니다. 그러나 마지막 전자의 'ar' 이 후자의 남은 'r'을 매칭 시키지 못하기 때문에 이전 'a*' 로 매칭된 'aaa'중 마지막 하나를 거꾸로 밟아 'a'를 취소함으로써 'ar'을 매칭시킵니다.

```
1)  ca*ar   =>  caaar       (match)
    ^^^         ^^^^
2)  ca*ar   =>  caaar       (not match)
       ^^           ^
3)  ca*ar   =>  caaar       (one back cancle)
    ^^^         ^^^
4)  ca*ar   =>  caaar       (match)
    ---^^       ---^^
```

3.4.2 하나이상 오퍼레이터 (+ or \+) (match-one-or-more operator)

RE_LIMITED_OPS 로 오퍼레이터 제한을 가하면, Regex 는 이 오퍼레이터를 인식하지 못합니다. 만일 RE_BK_PLUS_QM 이 세팅되어 있다면, '\+' 가 그 역할을 하고, 아니면 '+' 가 됩니다.

이것은 앞서의 뻥개이상 오퍼레이터 ('*')와 적어도 하나는 매칭시킨다는 점을 제외하고는 같습니다.

가령, '+'가 이 오퍼레이터면, 'ca+r' 은 'car', 'caaaar'과 매칭되고, 'cr'과는 매칭되지 않습니다.

3.4.3 뻥개나 한개 오퍼레이터 (? or \?)

이것도 역시 RE_LIMITED_OPS 가 설정되어 있으면, 인식하지 못합니다. 아울러, RE_BK_PLUS_QM 의 세팅여부에 따라, '\?' 나 '?'가 그 역할을 합니다.

이 오퍼레이터는 뻥개이상의 오퍼레이터와 한개나 하나도 매칭시키지 않는다는 점만 제외하면 비슷합니다. 예를 들면, 'ca?r'은 'car'나 'cr'을 매칭시키고, 다른 것들은 매칭되지 않습니다.

3.4.4 간격 오퍼레이터 ({...} 또는 \{...\}) (interval operator)

이 오퍼레이터를 사용하면, 특정 패턴의 출현빈도를 지정할 수 있습니다.

RE_INTERVALS 가 세트되어 있다면, Regex는 이것을 인식합니다. 아울러 다른 것과 마찬가지로 가능한한 가장 적은 횟수의 반복과 매칭됩니다.

RE_NO_BK_BRACES 가 세트되었다면, '{', '}'가 오퍼레이터가 되며, 그렇지 않다면, '\{'와 '\}'가 오러페이터가 됩니다.

'{' 와 '}' 가 현재의 간격 오퍼레이터라고 했을 경우에, 다음의 뜻은 다음과 같습니다.

* r{2,5} : 2개에서 5개 사이의 'r'
* r{2,}   : 2개 이상의 'r'
* r{4}   : 정확히 4개의 'r'

다음의 경우에는 틀린 것이 됩니다.

* 최소한계 갯수가 최대한계 갯수보다 클 경우
* 간격 오퍼레이터 안의 숫자가 RE_DUP_MAX 의 범위를 벗어날 경우

만약, 간격 표현식이 잘못 작성되어 있고, 문법비트가 RE_NO_BK_BRACES 가 세트되어 있을 경우에는, Regex 는 간격 오퍼레이터 안에 있는 모든 문자는 그냥문자(일반문자)로 재구성합니다. 이 비트가 세트되어 있지 않다면, 그 정규표현식은 진짜로 틀린 것이 됩니다.

또한, 정규표현식이 유효하긴 한데, 간격 오퍼레이터가 작동할 대상이 없을 경우, RE_CONTEXT_INVALID_OPS 가 세트되어 있다면, 그 정규표현식은 틀린 것이 됩니다. 비트가 세트되어 있지 않다면, Regex 는 간격 오퍼레이터 안의 모든 문자를 그냥 문자(일반문자)로 재구성하며, 백슬래쉬는 그냥 무시해버립니다.

flex 로 간단히 예를 들어보겠습니다.

```
.....................................................................
queen:~/regex$ echo -e "%%\nx{5} printf(\"only five\\n\"); " | flex
queen:~/regex$ gcc lex.yy.c -lfl
queen:~/regex$ a.out
xxxxx
only five

^D
queen:~/regex$
.....................................................................
```

3.5 대체 오퍼레이터 (| or \|) (alternation operator)

RE_LIMITED_OPS 로 오러레이터에 제한을 가한다면, Regex 는 이것을 인식하지 않

습니다. RE_NO_BK_VBAR 가 세트되어 있다면, '|'가 이것을 의미하고, 그렇지 않다면 '\|'가 이 오퍼레이터를 나타냅니다.

대체 오퍼레이터는 정규표현식 중의 하나를 매칭시킵니다. 'foo|bar|quux'는 'foo'나 'bar' 또는 'quux'와 매칭됩니다.

데체 오퍼레이터는 가장 낮은 우선순위를 가지기 때문에, 그룹 오퍼레이터를 사용하여 괄호를 묶을 수도 있습니다. 예를 들자면, '(u|li)n(i|u)x' 는 'linux', 'unix' 등과 매칭됩니다.

3.6 리스트 오퍼레이터 ([...] and [^...])

리스트 오퍼레이터는 하나 이상의 아이템의 집합으로 되어 있습니다. 하나의 아이템은 문자(예: 'a'), 문자 클래스 표현식(예: '[:digit:]'), 범위 표현식('-')이 들어갈 수 있습니다. 리스트안에 어떤 아이템을 취할 수 있는 지는 문법비트에 영향을 받습니다. 비어있는 리스트 ('[]')는 틀린 것이 됩니다.

에를 들면, '[ab]'는 'a'나 'b'를 매칭시키고, '[ad]*'는 빈문자열이나, 'a'나 'b'가 앞서는 한개이상의 문자열과 매칭됩니다.

이것과는 반대의 의미를 지니는 것이 있습니다. 위의 '[..]'가리스트 안의 하나를 매칭시키는 것이라면 '[^...]'는 리스트안의 문자가 아닌 하나의 문자와 매칭됩니다. '^'는 "라인의 처음"이라는 용도로 사용되지만, 리스트의 처음에 오면, 이후의 문자가 아닌 하나의 문자와 매칭시키는 역할을 합니다. 앞서의 예제에서도 살펴보았지만, '[^a-zA-Z]'는 알파벳 문자가 아닌 문자와 매칭됩니다. 아울러, 일반적인 경우에, 리스트안에서는 특수문자들이 그 의미를 상실한다고 앞에서 말씀드렸습니다. 따라서, '[.*]'는 보통 '.'나 '*' 문자를 매칭시킵니다.

조금의 특수한 경우가 있긴 합니다.

```
']'  : 리스트를 닫는 역할을 합니다. 다만 '[' 다음에 ']' 가 바로오면 그냥
       문자입니다.
'\'  : RE_BACKSLASH_ESCAPE_IN_LISTS 문법 비트가 세트되었다면 다음문자를
       이스케이프 시키는 역할을 합니다.
'[:' : RE_CHAR_CLASSES 가 세트되고 그뒤에 문법에 맞는 클래스 이름이 따라
       온다면 문자 클래스 오퍼레이터가 됩니다.
':]' : 문자 클래스를 닫는 역할을 합니다.
'-'  : 리스트의 처음에 오지 않고 (예: '[-.]'), 범위지정에서 끝 포인터에 오지
       않는 다면(예: '[a--]') 범위 오퍼레이터의 역할을 합니다.
```

3.6.1 문자 클래스 오퍼레이터 ([:...:]) (character class operators)

이것은, 유사한 성격의 문자들을 사용자가 알아보기 쉽게 단어로 그룹을 지어서 사용하는 것입니다. C 에서의 isdigit, isalpha 등과 같이 구성이 되어 있습니다.

가령, '[[:alnum:]]'은 '[a-zA-Z0-9]' 와 같은 의미를 가지지요.
사용할 수 있는 클래스는 다음과 같습니다.

```
alnum  : 알파벳과 숫자
alpha  : 알파벳
blank  : 스페이스나 탭 (시스템에 의존적임)
cntrl  : 아스키코드에서의 127 이상의 문자와 32 이하의 제어문자
         (한글의 첫째바이트가 127 이상이므로 제어문자로 취급됨 :()
digit  : 숫자
graph  : 스페이스는 제외되고 나머지는 'print' 항목과 같음.
lower  : 소문자
print  : 아스키코드에서 32에서 126까지의 찍을 수 있는 문자
punct  : 제어문자도 아니고 알파벳.숫자도 아닌 문자
space  : 스페이스, 케리지 리턴, 뉴라인, 수직 탭, 폼피드
upper  : 대문자
xdigit : 16진수, 0-9, a-f, A-F
```

클래스 오퍼레이터는 리스트 안에서만 (예: '[[:digit:]]') 효력을 발휘하고, 그냥 '[:digit:]' 와 같이 사용하면 다른 의미를 가지게 됩니다.

3.6.2 범위 오퍼레이터 (-) (range operator)

범위 오퍼레이터는 리스트 안에서만 작동하며, '-'를 앞뒤로 한 두문자사이의 모든 문자를 의미합니다. 가령, 'a-f'는 'a'에서 'f'사이의 모든 문자를 포함합니다.

주의) 문자 클래스는 범위에서 시작과 끝포인터에 사용될 수 없습니다. 그것은 하나의 문자가 아니라 문자그룹이기 때문에 그렇죠.

잘못된 경우 : '[[:digit:]-[:alpha:]]'

이외에, 약간의 특수한 경우가 있습니다.

RE_NO_EMPTY_RANGES가 세트되었고, 범위의 끝 포인터가 시작포인터보다 작다면, (예: '[z-a]') 그것은 틀린 것이 됩니다. 해당 문법비트가 세트되어 있지 않다면, 그 범위는 텅 비게 만듭니다. 만일 '-'문자를 원래의 문자의미로 리스트안에 넣을려면, 다음 중 한가지를 따라야 합니다.

* 리스트의 첫부분이나 마지막에 삽입한다.
* 범위의 시작포인터가 '-'보다 작게 하고, 끝포인터를 '-'와 같거나 크게 한다.

에를 들어, '[-a-z]'는 소문자나 '-'를 의미합니다.

3.7 그룹화 오퍼레이터 ((...) or \(...\)) (grouping operators)

Regex 에서는 그룹을 하나의 보조 표현식으로 처리합니다. 마치 수학연산에서 '(a*(b-c)+d)/e' 와 같이 말입니다. 여기서 바깥쪽 괄호부터 그룹1번, 안쪽 괄호('(b-c)')가 그룹2번이 됩니다. 즉, 왼쪽에서 오른쪽으로, 바깥쪽에서 안쪽으로 그룹의 순서가 매겨집니다. 이것은 잠시뒤에 설명할 "거꾸로 참조(후진참조)" 오퍼레이터에 의해 사용됩니다. 사실, 연산식 등에서 괄호가 연속으로 나올경우, C의 파싱에서도 왼쪽에서부터 괄호를 처리합니다.

따라서, 그룹을 사용하면 다음의 일을 처리할 수 있습니다.

* 대체오퍼레이터 ('|')나 반복오퍼레이터 ('+'나 '*')에서 인자의 범위를 지정합니다.
* 주어진 그룹과 매칭되는 보조문자열의 인덱스의 자취를 유지합니다.
  이 그룹오퍼레이터를 사용하면,
  * "거꾸로참조" (back-reference)오퍼레이터를 사용할 수 있습니다.
  * 레지스터를 사용할 수 있습니다.

이 부분들은 나중에 자세히 설명하겠습니다.

문법비트가 RE_NO_BK_PARENS 가 세트되어 있다면, '('와 ')'가 그 역할을 하며, 아니면, '\('와 '\)'가 그 역할을 합니다. RE_UNMATCHED_RIGHT_PAREN_ORD 가 세트되어 있고, '('는 있는 데 ')'가 없다면, ')'가 매칭된 것으로 생각하고 넘어갑니다.

3.8 거꾸로참조 오퍼레이터 (\숫자) (back-reference operator)

이 오퍼레이터는 사실, 조금 헷갈리기는 하지만 비슷한 패턴이 여러번 나올경우에 상당한 편의를 제공합니다.

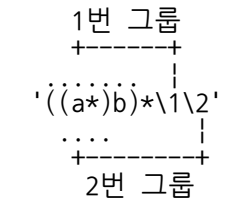
RE_NO_BK_REF 문법 비트가 세팅되어 있지 않다면, 이 오퍼레이터를 인식합니다.
거꾸로참조 오퍼레이터는 이미 기술한 앞의 그룹을 매칭합니다.
정규표현식 중 '숫자' 그룹을 나타내기 위해서는 '\숫자'형태로 사용합니다.
숫자는 '1'에서 '9'까지 가능하며, 이것은 처음의 1에서 9까지의 그룹과 매칭됩니다.

조금더 세부적인 이야기를 해보겠습니다.

* '(a)\1' 은 'aa'와 매칭합니다. '\1'은 첫번째 그룹을 나타내며, '(a)'로 괄호

로 둘러쌈으로써 그룹을 표시하는 것입니다. 마찬가지로, '(bana)na\1bo\1'은 'bananabanabobana'와 매칭됩니다.

* 조금 복잡한 이야기를 해보겠습니다. 반복 오퍼레이터 등의 작동으로 그룹이 한번 이상 매칭이 될 경우 거꾸로참조 오퍼레이터는 마지막으로 매칭된 보조 문자열을 매칭합니다. 말로만 하면 이해가 안되므로, '((a*)b)*\1\2' 와 'aabababa'와의 매칭여부를 따져볼까요? :) 이게 산술연산식이면 얼마나 좋겠습니까마는 안타깝게도 정규표현식이니만큼 조금 햇갈리더라도 잘 살펴보면 그리 어렵지만은 않습니다. 괄호의 순서에 따라 그룹은 다음과 같이 대응합니다.



매칭되는 순서를 살펴봅시다. '--' 는 매단계에서 서로 매칭되는 부분입니다.

```
1) ((a*)b)*\1\2     aabababa
   -------          ---
2) ((a*)b)*\1\2     aabababa
         -             --
3) ((a*)b)*\1\2     aabababa
          --            --
4) ((a*)b)*\1\2     aabababa
            --            -
```

여기서 생각해야 할점은 3)단계의 '\1'은 1단계의 '((a*)b)'와 매칭되나, 이것은 또한 2)단계의 '*' 반복 오퍼레이터에 의해 '*'(ab)와 매칭됩니다. 따라서, 최종적으로 '\1'은 'ab'와 매칭됩니다.
물론 위의 표현식은 'aababa'와도 매칭이 됩니다.

```
.....................................................................
queen:~/regex$ echo "aabababa" | egrep "((a*)b)*\1\2"
aabababa
queen:~/regex$ echo "aababa" | egrep "((a*)b)*\1\2"
aababa
.....................................................................
```

* '(one()|two())-and-(three\2|four\3)' 은 'one-and-three' 와 'two-and-four' 와 매칭이 되지, 'one-and-four'와 'two-and-three'와는 매칭이 되지 않습니다. 여기에서, 먼저 'one-and-' 부분까지 매칭이 되었다고 하면, 두번째 그룹(one 옆의 괄호)은 빈문자열과 매칭이 되었고, 세번째 그룹(two옆의 괄호)는 매칭에 관여하지 않게 됩니다. 그런상황에서 'four'가 매칭이 될 경우, Regex 는 그룹3을 참조하기 위해 거꾸로 돌아갑니다. 그러나 이미 그룹3은 매칭에 관여하지 않기 때문에 전체 매칭은 실패로 돌아갑니다.

거꾸로참조 오퍼레이터를 반복 오퍼레이터의 인자로 쓸수도 있습니다. 예를 들면, '(a(b))\2*'는 'a'다음에 'b'가 하나이상 오는 것과 매칭이 됩니다. 아울러, '(a(b))\2{3}' 은 'abbbb'와 매칭이 됩니다.

당연히, n번째의 보조표현이 없다면 매칭은 실패하게 됩니다.
재미있지 않습니까? ^^

3.9 닻 오퍼레이터 (^, $) (anchoring operators)

닻 오퍼레이터는 전체 문자열이나 하나의 라인에서 시작과 끝을 나타내는 것들입니다.

3.9.1 라인의 시작 오퍼레이터 (^)

이 오퍼레이터는 문자열의 시작이나 뉴라인 문자 다음의 빈문자열와 매칭할 수 있

습니다.

다음의 경우에 '^'는 이 오퍼레이터의 역할을 하고, 다른 경우에는 그냥문자가 됩니다.

* '^' 이 패턴에서 처음에 위치한다. 가령, '^foo' 같은 경우
* 문법비트가 RE_CONTEXT_INDEP_ANCHORS 가 세트되었고, 괄호나 그룹..등의 밖에 있을 경우
* 오픈그룹이나 대체 오퍼레이터 다음에 따라올 경우, 예를 들면, 'a\(^b\)', 'a\|^b'

이러한 규칙은 '^' 를 포함하는 유효한 패턴이라고 하더라도 매칭될 수 없다는 것을 암시합니다. 만약, 패턴 버퍼에서 newline_anchor 필드가 세트되었다면, '^' 는 뉴라인 다음과의 매칭에 실패합니다. 이것은 가끔 전체 문자열을 라인으로 나누어서 처리하지 않을 때에 유용하다고 하는군요.

3.9.2 라인의 끝 오퍼레이터 ($)

이 오퍼레이터는 문자열의 끝이나 뉴라인 문자의 이전의 빈 문자열과 매칭됩니다. 이것은 항상 '$'로 나타납니다. 예를 들면, 'foo$'는 'foo\nbar'의 처음 세글자와 매칭이 됩니다.

(다음 시간에는 GNU 오퍼레이터와 GNU emacs 오퍼레이터를 잠깐 살펴보고 재미있는 Regex 프로그래밍에 들어가겠습니다.)

『 리눅스 학당-리눅스 강좌 / 연재 (go LINUX) 』 463번
제 목:정규표현식 프로그래밍 강좌 [03]
올린이:엠브리오(유형목) 97/05/26 23:04 읽음:1803 관련자료 없음

한동훈님의 정규표현식 강좌입니다. 역시 하이텔 리눅스동에서 퍼왔습니다.

#617 한동훈 (ddoch)
[강좌] Regex (정규표현식) 라이브러리 (3) 05/26 19:42 288 line

GNU REGEX (정규표현식) 라이브러리 강좌 (3)

4. GNU 오퍼레이터

이 장에서 설명하는 것은 POSIX에는 정의되지 않았으나 GNU 에 의해 정의된 오퍼레이터입니다.

4.1 워드 오퍼레이터 (word operators)

여기에 나오는 오퍼레이터는 Regex 가 단어들의 일부분을 인식해야 가능합니다. Regex 는 어느 문자가 단어의 일부분인지 아닌지를 결정하기 위해 문법 테이블을 사용합니다.

사실, 텍스트를 처리하거나 관련작업을 하다보면 단어단위로 하여야 할 작업이 많이 있습니다. 하지만 표준 POSIX에서는 단어(워드)단위의 작업에 대해 특별히 지원가능하게 규정된 것이 없습니다. 하지만 GNU 에서는 쓸만한 워드 단위의 작업을 유용하게 처리할 수 있는 다양한 오퍼레이터를 지원함으로써 정규표현식을 좀더 강력하게 제어할 수 있게 되었습니다. 이런 워드 오퍼레이터는 많이 사용되고 있지 않지만 활용을 잘 하면 아주 똑똑한 일을 많이 처리할 수 있습니다.

4.1.1 이맥스가 아닌 문법 테이블 (non-emacs syntax tables)

문법 테이블은 일반적인 문자세트의 문자들에 의해 인덱스화된 하나의 배열입니다. Regex 는 항상 이 인덱스 테이블을 사용하기 위해 항상 char * 변수값을 사용합니다. 몇몇 경우에는 이 변수값을 초기화하고 순서대로 여러분들이 초기화시킬수도 있습니다.

* Regex 가 전처리 심볼 emacs 로 컴파일되었고, SYNTAX_TABLE 이 둘다 정의되지 않았다면, Regex 는 re_syntax_table 을 할당하고 i가 글자이거나 숫자, ' ' 이라면, 원소 i나 SWord를 초기화한다. i가 그렇지 않다면 그 값은 0으로 초기화됩니다.
* Regex 가 정의되지 않은emacs로 컴파일되었으나 SYNTAX_TABLE 이 정의되었다면 여러분들은 char * 변수 re_syntax_table 을 유효한 문법 테이블(syntax table)로 정의하여야 합니다.
* Regex가 전처리 심볼 emacs가 정의된 상태에서 컴파일되었다면 어떤 일이 일어나는 지는 뒤에서 설명합니다.

4.1.2 Match-word-boundary Operator (\b)

'\b' 는 단어를 구분짓습니다. 즉, 이것은 단어의 시작과 끝의 빈 문자열과 매칭이 됩니다. 예를 들면, '\brat\b'는 분리된 낱말, 'rat'을 매칭시킵니다.
그러나,단어의 범위를 어떻게 규정하는가 하는 것은 몇가지 예제로 충분히 유추할 수 있을 것입니다.

이 강좌의 처음에 든 예를, 이 오퍼레이터를 사용하면 더 간단합니다.

```
.................................................................
grep "\bint\b" regex.c

        mcnt = (int) Sword;
int mcnt;
.........
queen:~/regex$
.................................................................
```

위의 예를 살펴볼 때, 단어는 "공백문자(화이트문자)나 부호문자('(', ']', '-', ..) 가 끼어들지 않는 문자의 연속된 집합" 정도로 생각할 수 있습니다.

4.1.3 Match-within-word Operator (\B)

'B' 는 낱말안에서의 빈문자열과 매칭합니다. 예를 들면, 'c\Brat\Be' 는 'create' 와 매칭하고, 'dirty \Brat'은 'dirty rat'과 매칭하지 않습니다.

4.1.4 Match-beginning-of-word Operator (\<)

'\<' 는 단어의 시작에서 빈문자열을 매칭합니다.

4.1.5 Match-end-of-word Operator (\>)

'\>' 는 단어의 끝에서 빈문자열과 매칭합니다.

```
.................................................................
queen:~/regex$ grep "\<char\>" regex.c
return (char *) re_error_msg[(int) ret];
   const char *s;
....
queen:~/regex$
.................................................................
```

4.1.6 Match-word-constituent Operator (\w)

'\w' 는 낱말을 이루는 어떤 문자와 매칭합니다.

4.1.7 Match-non-word-constituent Operator(\W)

'\W' 는 낱말의 성분요소가 아닌 어떤 문자와 매칭합니다.

```
.................................................................
```

```
queen:~/regex$ echo " int " ¦ grep "\Wi\wt"
 int
queen:~/regex$
```

'\w' 과 '.'의 차이점은 전자는 낱말속의 어느 한문자(그러므로 낱말의 구성요소)와 매칭이 되나, '.'는 이것저것 따지지 않고 어느 한문자와 매칭이 되므로 조금 의미적으로 틀립니다. 아울러, '\W'도 낱말속의 어떤 문자 (예를 들면, 'int'속의 'n')과는 매칭이 되지 않으며 낱말에 포함되지 않는 어떤 한문자 (예를 들면, ' ')와 매칭이 됩니다.

4.2 버퍼 오퍼레이터

이제 설명할 것은 버퍼에서 작동하는 오퍼레이터입니다. 이맥스에서의 buffer는 "이맥스 buffer" 입니다. 다른 프로그램에서는 전체 문자열을 버퍼로 여깁니다.

4.2.1 Match-beginning-of-buffer Operator (\`)

'\`'는 버퍼의 시작되는 부분의 빈문자열과 매칭됩니다.

4.2.2 Match-end-of-buffer Operator (\')

'\''는 버퍼의 끝 부분의 빈문자열과 매칭됩니다.

5. GNU 이맥스 오퍼레이터

이제 설명할 것은 POSIX에서는 정의되지 않았고, GNU에서 정의되었으며, 이것을 사용할 때는 Regex 가 컴파일 될 때 전처리 심볼을 정의된 emacs로 하여야 합니다.

5.1 문법 클래스 오퍼레이터 (syntactic class operators)

이 오퍼레이터들은 Regex 가 이 문법 문자들의 클래스를 인식하여야 합니다.
Regex 는 이것을 검사하기 위해 문법 테이블을 사용합니다.

5.1.1 이맥스 문법 테이블

하나의 문법 테이블은 여러분들의 문자셋(아스키문자셋 같은 것들..)에 의해 인덱스화된 하나의 배열입니다. 아스키 하에서는 따라서 문법 테이블은 256개의 원소를 가집니다.

Regex 가 전처리 심볼, 정의된 emacs 로 컴파일되었다면, 여러분들은 re_syntax_table 을 정의하고 그 값을 이맥스 문법 테이블로 초기화하여야 합니다. 이맥스 문법 테이블은 Regex 의 문법 테이블보다는 좀 더 복잡합니다.

5.1.2 Match-syntactic-class Operator (\sclass)

이 오퍼레이터는 문법 클래스가, 서술된 문자가 명시하는, 어떤 문자를 매칭합니다. '\sclass'가 이 오퍼레이터를 나타내며, class는 여러분들이 원하는 문법 클래스를 나타내는 문자입니다. 예를 들여, 'w' 는 단어를 구성하는 문자의 문법 글래스를 나타내므로, '\sw'은 단어를 구성하는 아무 문자와 매칭합니다.

5.1.3 Match-not-syntactic-class Operator (\Sclass)

위의 오퍼레이터와는 반대되는 뜻입니다. 예를 들어, 'w' 는 단어를 구성하는 문자의 문법 클래스를 나타내므로, '\Sw' 은 단어의 구성성분이 아닌 아무 문자와 매칭됩니다.

지겹게 지금까지 많을 것을 설명드렸지만, 사실 이 모든 것을 다 한꺼번에 기억하실 필요성은 없습니다. 자주 사용하시면서 그때그때 마다 조금씩 익숙하게 익히시는 것이 좋으리라 봅니다.

이제, 조금 더 재미있는 Regex 프로그래밍에 들어가겠습니다.

6. Regex 프로그래밍

Regex 는 세가지 다른 인터페이스가 있습니다. 하나는 GNU를 위해 디자인 된 것과, 하나는 POSIX 에 호환되는 것, 나머지 하나는 Berkeley UNIX 에 호환되는 것입니다.
다른 유닉스 버젼에도 충분히 호환되는 것으로 프로그래밍을 하시려면, POSIX Regex 함수로 프로그래밍하시는 것이 좋을 겁니다. 그렇지 않고 일반적으로, GNU 의 강력한 기능을 사용하시려면 GNU Regex 함수를 사용하시는 것이 좋을 것 입니다.

그럼, 먼저 비교적 간단한 BSD Regex 함수부터 살펴보겠습니다.

6.1 BSD Regex 함수

Berkeley UNIX 에 호환되는 코드를 작성하려면, 이 함수를 사용하십시요.
그러나, 그다지 많은 기능은 지원되지 않고, 간단한 두개의 함수만이 지원됩니다. 따라서, BSD Regex 함수로는 간단한 검색은 할 수 있으나, 매칭작업은 할 수 없습니다.

BSD Regex 함수로 검색을 하기위해서는 다음의 순서를 따라야 합니다.

1) re_syntax_options 의 값을 원하는 정규표현식 문법비트의 값으로 설정합니다.
   앞에서 설명이 된, 각종의 문법 비트를 조합하여 설정할 수 있습니다.

   예) re_syntax_options = RE_SYNTAX_POSIX_BASIC;

2) 정규표현식을 컴파일 합니다.

   char *re_comp (char *regex)

   regex 는 널로 끝나는 정규표현식의 주소입니다. re_comp 는 내부적으로 패턴버퍼를 사용하기 때문에 사용자에게는 노출이 되지 않기 때문에, 새로운 정규표현식으로 검색하려면, 해당 정규표현식을 재 컴파일하여야 합니다. 즉, 내부의 패턴버퍼를 현재의 정규표현식과 맞추어 주어야 한다는 것입니다. 만일 regex 를 NULL스트링 으로 컴파일 할경우에는 내부의 패턴버퍼가 변하지 않으니 주의를 하여야 합니다.

   re_comp 는 성공적으로 컴파일되었다면, NULL을 돌려주며, 정규표현식이 잘못되거나 문제가 생겨서 컴파일 할 수 없다면 에러 문자열을 돌려줍니다.
   이 에러 문자열은 뒤에 나올 re_compile_pattern 의 그것과 같습니다.

3) 검색작업을 합니다.

   int re_exec (char *string)

   한번 re_comp 로 정규표현식을 컴파일 하였다면, 이제 re_exec 를 사용하여 string 문자열내에서 해당 표현이 나오는 지를 검색할 수 있습니다.

   re_exec 는 검색에 성공했을 경우에 1을 리턴하고, 실패했을 경우에는 0을 리턴합니다. 이 함수는 내부적으로 빠른 검색을 위해 GNU fastmap 을 사용합니다.

자, 그럼 이제 간단한 예제를 하나 만들어 보도록 합시다. 위의 함수를 사용하여 간단한 패턴 검색을 테스트 하는 것입니다.

```
/* BSD Regex functions example
```

```
   Usage : bsd search_string pattern
*/

#include <stdio.h>
#include <stdlib.h>
#include "regex.h"

void main(int argc, char *argv[]) {
        char *error;
        re_syntax_options = RE_SYNTAX_POSIX_BASIC;

        if (argc != 3) exit(1);
        if ((error = re_comp(argv[2])) != NULL) {
                fprintf(stderr, "re_comp: %s: %s\n", argv[2], error);
                exit(1);
        }
        switch(re_exec(argv[1])) {
                case 0 :
                        fprintf(stderr, "re_exec: \"%s\" failure..\n", argv[1]);
                        break;
                case 1 :
                        fprintf(stderr, "re_exec: \"%s\" success..\n", argv[1]);
                        break;
        }
}
```

먼저, 현재 여러분들이 테스트 하시는 디렉토리에 "regex.c" 와 "regex.h" 를 한부 복사해 두시고, regex.c 를 컴파일만 하여 오브젝트 파일을 만들거나 이미 컴파일된 regex.o 를 한부 가지고 옵니다. 컴파일 할 경우,

```
queen:~/regex$ gcc -c regex.c -g
```

위의 소스를 bsd.c 로 저장을 한다면, 이제 다음과 같이 컴파일 하면 됩니다.

```
queen:~/regex$ gcc -o bsd bsd.c regex.o
```

다음은 테스트 결과입니다.

```
.....................................................................
queen:~/regex$ bsd "lnx5, 2445 #linux" "[[:digit:]]\{4\}\W.li\w\wx"
re_exec: "lnx5, 2445 #linux" success..
queen:~/regex$ bsd "printf (\"int i = 10\")" "\<int\b"
re_exec: "printf ("int i = 10")" success..
queen:~/regex$  bsd "regex is powerful" "\b\w*\W[is"
re_comp: \b\w*\W[is: Unmatched[ or [^
queen:~/regex$
.....................................................................
```

다음 시간에는 POSIX Regex 함수를 살펴보겠습니다.

현재 할일이 밀려서 이번 시간은 조금 줄이도록 하겠습니다.

또치 한동훈 드림

『 리눅스 학당-리눅스 강좌 / 연재 (go LINUX) 』 466번
 제 목:정규표현식 프로그래밍 강좌 [04]
 올린이:엠브리오(유형목)   97/06/15 13:46   읽음:1654 관련자료 없음

한동훈님의 정규표현식 라이브러리 강좌입니다.

#619 한동훈 (ddoch)
[강좌] Regex (정규표현식) 라이브러리 (4) 06/02 20:14 365 line

GNU REGEX (정규표현식) 프로그래밍 강좌 (4)

6.2 POSIX Regex 함수

POSIX 와 호환되는 코드를 작성하려면 여기에 나오는 함수들을 사용할 수 있습니다.

6.2.1 POSIX 패턴 버퍼

POSIX 에서 정규표현식을 컴파일하거나 매칭작업을 하려면, BSD 와는 다르게 패턴 버퍼를 제공하여야 합니다. regex_t 타입인 POSIX 패턴 버퍼는, re_pattern_buffer 타입인 GNU 패턴버퍼와 구성이 동일 합니다.

"regex.h" 에 보면 다음과 같이 형정의되어 있습니다.

```
typedef struct re_pattern_buffer regex_t;
```

패턴 버퍼란 이전에도 말씀드렸지만, 해당 정규표현식에서 패턴을 매칭시키기 위한 다양한 정보를 가지고 있는 버퍼입니다. 이것은 물론, 컴파일을 함으로써 사용가능하게 됩니다.

그럼, 먼저 GNU 패턴 버퍼를 살펴볼까요?

여러분들은 서로 다른 여러종류의 패턴 버퍼를 동시에 보유할 수 있습니다.
"regex.h" 는 아래와 같은 패턴 버퍼를 정의하고 있습니다.

```
  /* 컴파일된 패턴을 가르키는 포인터. 이것의 원소는 배열의 인덱스로
     사용될 때가 있기 때문에 'unsigned char *'로 정의되었습니다. */
  unsigned char *buffer;

  /* 'buffer' 가 포인트하는 바이트수 */
  unsigned long allocated;

  /* 'buffer' 안에 사용되고 있는 바이트수 */
  unsigned long used;

  /* 패턴이 컴파일될 때 세팅되는 문법 */
  reg_syntax_t syntax;

  /* 어떤 fastmap 을 가르키는 포인터. NULL 이라면 포인팅 하지 않는 경우입니
     다. re_search 는, 만일 fastmap 이 존재할 경우, 빠른 매칭을 위해서, 매칭
     이 불가능한 출발 포인트는 건너 뛰게 됩니다. */
  char *fastmap;

  /* NULL 이 아니라면, 어떤 문자들을 비교하기 전에, 모든 문자들에 적용되는
     변환테이블입니다. NULL 일 경우에는 변환이 없습니다. */
  char *translate;

  /* (정규표현식) 컴파일러에 의해 발견된 보조표현식의 수 */
  size_t re_nsub;

  /* 현재의 패턴이 빈문자열과 매칭할 수 없다면 0이 되고, 그외는 1이 됩니다.
     이것은 're_search_2' 에서만 사용됩니다. */
  unsigned can_be_null : 1;

  /* REGS_UNALLOCATED : 'regs' 구조체에 RE_NREGS 나  re_nsub + 1 중 큰수
                        수 만큼 그룹을 할당합니다.
     REGS_REALLOCATE  : 필요하다면 공간을 재 할당합니다.
     REGS_FIXED       : 그냥 있는 것을 사용합니다.  */
  #define REGS_UNALLOCATED 0
```

```
#define REGS_REALLOCATE 1
#define REGS_FIXED 2
unsigned regs_allocated : 2;

/* 패턴을 'regex_compile' 로 컴파일 할 때 0으로 세팅됩니다.
   're_compile_fastmap'이 fastmap 을 업데이트 할 경우에는 1로 세팅됩니다. */
unsigned fastmap_accurate : 1;

/* 이것이 세트되어 있다면, 're_match_2' 는 보조표현식에 관한 정보를 리턴하
   지 않습니다. */
unsigned no_sub : 1;

/* 이것이 세트되어 있다면, 라인의 시작을 나타내는 표시기(일반적으로는 '^')
   는 문자열의 시작을 매칭하지 못합니다. */
unsigned not_bol : 1;

/* 이것은 라인의 끝을 나타내는 표시기(일반적으로는 '$')와 유사합니다. */
unsigned not_eol : 1;

/* 이것이 세트되면, 뉴라인에서 표시기가 매칭됩니다. */
unsigned newline_anchor : 1;
```

사실, 이 가운데에서 자주 사용하는 것은 몇개 정도에 지나지 않을 것입니다.

6.2.2 POSIX 정규표현식 컴파일

패턴 버퍼를 컴파일하려면 'regcomp' 를 사용합니다.

```
int regcomp (regex_t *preg, const char *regex, int cflags)
```

'preg' 는 초기화할 패턴 버퍼의 주소입니다. 'regex' 는 정규표현식의 주소입니다. 그리고 cflags 는 조합가능한 컴파일 플래그입니다. 유효한 비트는 다음과 같습니다.

REG_EXTENDED
     POSIX 확장 정규표현식을 사용하겠다는 것을 의미합니다. 이것이 세트되어 있지 않다면 POSIX 기본 정규표현식을 사용하겠다는 것을 의미합니다. regcomp 는 'preg'의 syntax 필드를 그에 알맞게 설정합니다.

REG_ICASE
    대소문자를 무시한다는 것을 의미합니다. regcomp 는 'preg' 의 'translate' 필드를 대소문자를 무시하는 변환데이블로 설정합니다.

REG_NOSUB
    'preg' 의 'no_sub' 필드를 세트하라는 의미입니다.

REG_NEWLINE
    * match-any-character operator ('.')는 newline 을 매칭하지 못합니다.
    * nonmatching list ('[^...]')는 newline 을 포함하지 못합니다.
    * match-beginning-of-line ('^') 는 REG_NOTBOL 이 어떻게 설정되어 있는가에 개의치 않고 newline 바로 뒤의 빈문쟈열을 매칭합니다.
    * match-end-of-line operator ('$') 는 REG_NOTEOL 이 어떻게 설정되어 있는가에 개의치 않고 newline 바로 이전에 오는 빈문자열을 매칭합니다.

regcomp 가 성공적으로 정규표현식을 컴파일하게 되면, 0을 리턴하고, '*pattern_buffer' 를 컴파일된 패턴으로 설정합니다. syntax 를 제외하고는, 이후에 살펴볼 GNU 컴파일 함수와 같은 방법으로 같은 필드를 설정합니다.

regcomp 가 컴파일에 실패하게 되면, 아래의 에러코드 중 하나를 반환합니다.

REG_BADRPT
    예를 들면, 'a**' 안의 연속적인 반복 연산자 '**' 의 경우

REG_BADBR
    예를 들면, 'a\{-1' 에서의 count '-1' 같은 경우

REG_EBRACE
    예를 들면, 'a\{1' 과 같이 '}' 가 빠진 경우

REG_EBRACK
    예를 들면, '[a' 와 같이 ']' 가 빠진 경우

REG_ERANGE
    예를 들면, '[z-a]' 나 '[[:alpha:]-¦]' 과 같이 잘못된 경우

REG_ECTYPE
    예를 들면, '[[:foo:]' 와 같이 잘못된 클래스 명칭인 경우

REG_EPAREN
   예를 들면, 'a\)' 와 같이 '(' 를 빠뜨렸을 경우

REG_ESUBREG
   예를 들면, '\(a\)\2' 와 같이 존재하지 않는 그룹을 참조하는 경우

REG_EEND
   예를 들면, 정규표현식이 더 이상의 명백한 에러를 야기하지 않을 경우

REG_EESCAPE
   예를 들면, 'a\' 에서와 같이 '\' 가 잘못 사용되었을 경우

REG_BADPAT
    예를 들면, 확장 정규표현식 문법에서 'a()b' 에서의 빈그룹 '()' 이 나올 경우

REG_ESIZE
   정규표현식이 패턴 버퍼의 크기로 65536 보다 큰 바이트를 필요로 할 경우

REG_ESPACE
    정규표현식이 Regex 가 실행하는 데에 필요한 메모리를 모자라게 할 경우

6.2.3 POSIX 매칭

한번, 패턴을 패턴버퍼로 컴파일을 했다면, 이제 매칭작업을 할 수 있습니다.
이 매칭작업을 'regexec' 가 수행을 합니다.

```
  int regexec (const regex_t *preg, const char *string,
               size_t nmatch, regmatch_t pmatch[], int eflags)
```

'preg' 는 패턴을 컴파일한 패턴 버퍼의 주소이고, 'string' 은 매칭을 하기를
원하는 문자열입니다. 'pmatch' 에 대해서는 뒤에서 자세하게 설명이 됩니다.
'nmatch' 를 0으로 설정하거나, 'preg' 를 컴파일 옵션 REG_NOSUB 로 세팅하였다
면 'regexec' 는 'pmatch' 를 무시할 것입니다. 그렇지 않으면, 여러분들은 적
어도 'nmatch' 원소들 만큼 할당해야 합니다. regexec 는 'nmatch' 바이트 옵셋을
'pmatch' 에 기록을 할것이며, 사용되지 않는 원소를 -1부터 'pmatch[nmatch]-1'
까지 설정할 것입니다.

'eflags' 는 실행 플래그를 설정하며, REG_NOTBOL 과 REG_NOTEOL 이 될 수 있습
니다. REG_NOTBOL 을 설정한다면, match-beginning-of-line operator ('^') 는
항상 매칭에 실패를 합니다. REG_NOTEOL 은 match-end-of-line operator 에 있어
서 위와 유사하게 작동합니다.

regexec 는 컴파일된 패턴이 'string' 과 매칭이 되었다면 0을, 그렇지 않다면,
REG_NOMATCH 를 리턴합니다.

6.2.4 에러 메시지 출력하기

regcomp 나 regexec 가 실패하게 되면, 0이 아닌 에러코드를 반환합니다. 이러한
에러코드들은 위의 6.2.2 와 6.2.3 에서 설명한 것들입니다. 에러코드에 해당하는
에러 문자열을 얻으려면 'regerror'를 사용할 수 있습니다.

```
  size_t regerror (int errcode,
                   const regex_t *preg,
                   char *errbuf,
                   size_t errbuf_size)
```

'errcode' 는 에러코드이고, 'preg' 는 에러가 발생한 패턴버퍼이며, 'errbuf' 는 에러 버퍼이며, 'errbuf_size' 는 'errbuf' 의 크기입니다.

regerror 는 'errcode' 에 대응하는 에러 문자열의 바이트 크기(널문자까지 포함)를 반환합니다. 'errbuf' 와 'errbuf_size' 가 0이 아니라면, 'errbuf' 에 처음 errbuf_size-1 문자의 에러 문자열을 널문자를 추가해서 돌려줍니다.
'errbuf_size' 는 'errbuf' 의 바이트 크기보다 작거나 같은 양수이어야 합니다.
여러분들은, 'regerror' 의 에러 문자열을 담아내는 데 얼마만큼 크기의'errbuf' 가 필요한지 알아보기 위해서 'errbuf' 를 NULL로, 'errbuf_size' 를 0으로 해서 호출할 수 있습니다.

6.2.5 바이트 옵셋 사용하기

POSIX 에서, regmatch_t 형 변수는 GNU 의 레지스터와 비슷하지만, 똑같지는 않습니다. POSIX 에서 레지스터의 정보를 얻으려면 regexec 에, regmatch 형 변수인, 0이 아닌 'pmatch'를 넘겨줄 수 있습니다. regmatch_t 형 구조체는 다음과 같습니다.

```
  typedef struct {
    regoff_t rm_so;
    regoff_t rm_eo;
  } regmatch_t;
```

매칭 함수가 정보를어떻게 레지스터에 저장하는 지는 뒷부분에서 설명하겠습니다.

GNU Regex 의 'regs' 와 POSIX 의 'regs' 는 유사하게 대응합니다.

'reg' 의 'pmatch', pmatch[i]->rm_so 는 regs->start[i] 와 대응하고
pmatch[i]->rm_eo 는 regs->end[i] 와 대응합니다.

6.2.6 POSIX 패턴 버퍼를 Free 하기

패턴 버퍼에 할당된 것을 free 하는 함수는 'regfree' 입니다.

```
void regfree (regex_t *preg)
```

'preg' 는 free 할, 할당된 패턴버퍼입니다. regfree 는 또한 'preg'의 'allocated' 와 'used' 필드를 0으로 설정합니다. 패턴 버퍼를 free 한 이후에는, 매칭 작업을 수행하기 전에 정규표현식을 해당 패턴 버퍼에 다시 컴파일해야 합니다.

6.2.7 POSIX Regex 로 egrep 만들기

grep 은 기본 정규표현식을 사용하고, egrep 은 확장 정규표현식을 사용하는데, 여기서는 egrep 의 기능을 간단하게 구현해 보도록 하겠습니다.

지금까지 설명한 기능만으로도 egrep 의 기본적인 기능은 쉽게 만들 수 있습니다. grep 류의 기본적인 기능은 '매칭' 이 아니라 '검색'이기 때문입니다.

우리가 만들'egrep' 을 'my_egrep' 이라고 부른다면, 'my_egrep' 의 기본적인 작동은 다음과 같이 하도록 합시다.

1) 특별한 옵션은 지원하지 않고, 인자는 모두 패턴이나 파일명으로 처리한다.
2) 입력파일명이 명시되지 않았을 경우에는 표준입력에서 받는다.
3) 컴파일 플래그는 'REG_EXTENDED' 를 사용하여 확장정규표현식을 지원한다.

추가적인 옵션을 지원하는 것은 소스를 조금씩 고치면서 시도해 보시기 바랍니다.

```
/* POSIX Regex 테스트 프로그램 : egrep 의 기본 기능 구현
 *
 * Designed by Han-donghun, 1997.5.31
 *
 * name    : my_egrep.c
 *
 * compile : First, you must have "regex.c" and "regex.h",
 *              in the current directory.
 *
 *           To get "regex.o " , type "gcc -c regex.c"
 *           Finally, to compile my_egrep.c, type follow.
 *
 *           "gcc -o my_egrep my_egrep.c regex.o"
 *
 * usage   : my_egrep pattern [files...]
 *
 * This is simple "pattern search" program
 *        using POSIX regex, like egrep.
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <errno.h>
#include "regex.h"    /* regex main header file */

void main(int argc, char *argv[]) {
  int ret = 0, error, i;
  char *msg;
  char buf[2048];
  FILE *fp;
  regex_t preg;

  if (argc <= 1) {
    fprintf(stderr, "usage: %s pattern [files..]\n", argv[0]);
    exit(1);
  }

  /* regex compile */
  if ((error = regcomp(&preg, argv[1],
        REG_EXTENDED | REG_NOSUB)) != 0) {
    ret = regerror(error, &preg, NULL, 0);
    msg = (char *)malloc(sizeof(char)*ret);
    regerror(error, &preg, msg, ret);
    fprintf(stderr, "%s: %s\n", argv[0], msg);
    free(msg);
    exit (1);
  }

  if (argc == 2) {
    while (fgets(buf, 2048, stdin) != NULL) {
      /* regex matching */
      if ((regexec(&preg, buf, 0, NULL, 0)) == 0) {
        printf("%s", buf);
      }
    }
  } else if (argc > 2) {
    for (i = 2; i < argc ; i++) {
      if ((fp = fopen(argv[i], "r")) == NULL) {
        fprintf(stderr, "%s: %s: %s\n", argv[0], argv[i], strerror(errno));
        continue;
      }
      while (fgets(buf, 2048, fp) != NULL) {
        /* regex matching */
        if ((regexec(&preg, buf, 0, NULL, 0)) == 0) {
          printf("%s", buf);
        }
```

```
      }
    }
  }
  regfree(&preg);
}
```

대소문자를 무시하게 만들려면, 정규표현식의 컴파일시에, regcomp 의
REG_EXTENDED 에 REG_ICASE 를 추가하시면 됩니다 (grep 류의 '-i' 옵션).
grep 류의 '-v' 나 -n' 옵션을 지원하는 것은 이제 간단하게 해결될 것입니다.

다음은 테스트 한 결과입니다.

```
$ gcc -o my_egrep my_egrep.c regex.o
$ my_egrep regcomp 정규표현식강좌.네번째
 패턴 버퍼를 컴파일하려면 'regcomp' 를 사용합니다.
  int regcomp (regex_t *preg, const char *regex, int cflags)
  ................
$ my_egrep "^[0-9]+\.[0-9]+\b" 정규표현식강좌.네번째
6.2 POSIX Regex 함수
6.2.1 POSIX 패턴 버퍼
6.2.2 POSIX 정규표현식 컴파일
6.2.3 POSIX 매칭
6.2.4 에러 메시지 출력하기
6.2.5 바이트 옵셋 사용하기
6.2.6 POSIX 패턴 버퍼를 Free 하기
6.2.7 POSIX Regex 로 egrep 만들기
$
```

(다음 시간에 이어집니다..)

『 리눅스 학당-리눅스 강좌 / 연재 (go LINUX) 』 467번
 제 목:정규표현식 프로그래밍 강좌 [마지막]
 올린이:엠브리오(유형목)   97/06/15 13:46   읽음:1809 관련자료 없음

한동훈님의 정규식 라이브러리 마지막 강좌입니다.

#622   한동훈   (ddoch   )
[강좌] Regex (정규표현식) 라이브러리 (끝   06/09 17:01   459 line

GNU REGEX (정규표현식) 프로그래밍 강좌 (5)

6.3 GNU Regex 함수

특별히 POSIX 나 버클리 UNIX 에 호환성을 생각하지 않아도 된다면, GNU regex
함수를 사용하는 것이 여러모로 좋을 지 모르겠습니다.
GNU regex 함수도 이전에 설명드린 POSIX 나 BSD regex 함수의 기능을 포함하
고 나머지 여러개의 복잡한 기능을 추가한 것입니다.

그럼, 하나씩 알아보도록 하겠습니다.

6.3.1 GNU 패턴 버퍼

GNU regex 는 GNU 패턴 버퍼를 이용하여 컴파일된 정규표현식을 활용합니다.
이 패턴 버퍼는 POSIX regex 에서 설명하였으므로 건너뛰겠습니다.

6.3.2 GNU 정규표현식 컴파일

GNU regex 에서는 정규표현식을 검색하고 매칭하는 것을 둘다 할 수 있습니다. GNU regex 에서도 POSIX 나 BSD regex 처럼, 먼저 정규표현식을 컴파일하여, 패턴버퍼에 마련해 두어야 합니다. 이전과 마찬가지로 패턴버퍼는 어떤 문법으로 컴파일되느냐에 따라 매칭이나 검색의 결과가 달라지게 마련입니다. 이러한 문법을 지정하는 변수는 re_syntax_options 입니다. 따라서 컴파일을 하기전에 정확한 문법을 세팅해 두는 것이 중요합니다.

GNU regex 에서 패턴을 컴파일하는 것은, re_compile_pattern 입니다. re_compile_pattern 은 패턴버퍼를 인자로 취하는 데, 패턴 버퍼의 다음의 필드는 초기화를 시켜주어야 합니다.

translate initialization
translate
: 매칭이나 검색이전에 적용되는 변환테이블을 사용한다면 그 변환테이블에 대한 포인터로 초기화 시키줍니다. 변환테이블이 없다면 NULL로 초기화 시켜주면 됩니다. translate 는 GNU 패턴버퍼에서 char * 형 필드임을 상기하세요.
변환 테이블에 대한 이야기는 뒷쪽에서 설명하겠습니다.

fastmap
: fastmap (re_search 로 빠른 검색에 사용됨) 을 사용하려면 그 포인터를 지정하면 되며, 필요없다면 NULL로 지정하면 됩니다. 이 또한 char * 필드 입니다.

buffer
allocated
: re_compile_pattern 으로 컴파일된 패턴에 필요한 메모리를 할당하고자 할 경우에는 둘다 0이나 NULL로 초기화 하면 됩니다. (buffer 는 unsigned char *, allocated 는 unsignedlong 형입니다. 0이나 NULL이나 결국에는 0입니다.) 여러분들이 이미 할당한 메모리 블록을 Regex 에 사용하려면, buffer 는 그것의 주소로, allocated 는 블록의 바이트 크기로 설정하면 됩니다.

: re_compile_pattern 은 컴파일된 패턴에 필요하다면 메모리를 확장하기 위해서 realloc 를 사용합니다.

패턴을 컴파일 하려면 다음과 같이 사용하면 됩니다.

```
char *re_compile_pattern (const char *regex, const int regex_size,
                          struct re_pattern_buffer *pattern_buffer)
```

'regex' 는 정규표현식 문자열의 주소이고, 'regex_size' 는 그것의 길이입니다. pattern_buffer 는 패턴버퍼의 주소입니다.

re_compile_pattern 이 성공적으로 해당 정규표현식을 컴파일하였다면 0(NULL)을 리턴하고, *pattern_buffer 를 컴파일된 패턴으로 설정을 합니다. 아울러 아래의 패턴버퍼 내의 필드를 세팅합니다.

```
buffer          컴파일된 패턴
used            buffer가 가르키는 곳에서 사용중인 바이트
syntax          re_syntax_options 의 현재값
re_nsub         'regex' 에서 보조표현식의 갯수
fastmap_accurate
```

re_compile_pattern 이 'regex' 를 컴파일 할 수 없다면, '6.2.2 POSIX 정규표현식' 에서 설명한 에러 문자열을 돌려줍니다.

6.3.3 GNU 매칭

GNU 매칭은 문자열속에서 가능한한 시작위치에서 명시된데로 매칭을 시킵니다. 한번 패턴을 패턴버퍼로 컴파일을 했다면, 문자열에서 패턴을 매칭 시킬수 있습니다.

```
int re_match (struct re_pattern_buffer *pattern_buffer,
              const char *string, const int size,
              const int start, struct re_registers *regs)
```

pattern_buffer 은 컴파일된 패턴버퍼의 주소이고, string 은 매칭을 하고자 하는 문자열입니다. 이 문자열에는 NULL 이나 newline 을 포함할 수 있습니다. size 는 그 문자열의 길이이며, start 는 매칭하기를 원하는 문자열속의 인덱스(문자열 첫 문자의 인덱스는 0)입니다.

re_match 는 pattern_buffer 의 syntax 필드의 문법에 따라, 문자열 string 을, pattern_buffer의 정규 표현식과 매칭을 시키는 역할을 합니다. 문자열과 매칭할 수 없다면 -1을 리턴하고, 내부적인 에러일 경우에는 -2를, 성공적일 경우에는 문자열과 매칭된 횟수를 돌려줍니다.

예를 들면, pattern_buffer 이 'a*'를 컴파일한 패턴버퍼라고 하고, string 이 'aaaaab'이며, 따라서 size는 6이 되고, start 는 2라고 가정한다면, re_match 는 3을 리턴합니다. 'a*' 는 문자열에서 마지막 세개의 'a'를 매칭시킬 것입니다. start 가 0이라고 한다면, re_match 는 5를 리턴합니다. start 가 5나 6일 경우에는 0을 반환합니다.
start 가 0에서 size 사이가 아니라면, re_match 는 -1을 반환합니다.

6.3.4 GNU 검색

검색하는 데 사용되는 함수는 re_search 입니다.

re_search 를 사용하기 전에 정규표현식을 컴파일 하셔야 겠죠?
re_search 의 정의는 다음과 같습니다.

```
int re_search (struct re_pattern_buffer *pattern_buffer,
               const char *string, const int size,
               const int start, const int range,
               struct re_registers *regs)
```

이 인자들은 re_match 와 유사합니다. 여기서 start 와 range 는 re_match 의 start 를 대응합니다.

range 가 양수이면, re_search 는 인덱스 start 에서 최초의 매칭을 시작하며 실패할 경우 start+1 에서 검색을 하며 계속 하나씩 나아가서 start+range 까지 수행합니다. range 가 음수라면, 인덱스 start 에서 첫 매칭을 수행하며, 이후에 -1씩 위치를 반대로 옮겨서 수행합니다.

start 가 0에서 size 사이가 아니라면, re_search 는 -1을 돌려줍니다. range 가 양수일 경우에는 re_search 는, 필요하다면 range 를 조절해서 start+range-1 이 0에서 size 사이가 되도록 하여 검색이 문자열 바깥으로 나가지 못하도록 합니다. 유사하게, range 가 음수라면, re_search 는 범위를 start+range+1 이 0에서 size 사이가 되도록 필요할 경우 조절하게 됩니다.

패턴버퍼의 fastmap 필드가 NULL 이라면, re_search 는 연속적인 위치로 매칭을 시작하며, NULL 이 아니라면, fastmap 을 사용하여 좀 더 효율적으로 검색을 수행합니다.

매칭이 한번도 되지 않는다면, re_search 는 -1을 반환하고, 매칭이 된다면 매칭이 시작된 위치의 인덱스를 돌려주며, 내부에러일 경우에는 -2를 돌려줍니다.

6.3.5 분리된 데이터로 매칭과 검색하기

re_match_2 와 re_search_2 를 사용하면, 두개의 문자열로 나누어진 데이터를 매칭하거나 검색할 수 있습니다.

```
int re_match_2 (struct re_pattern_buffer *buffer,
                const char *string1, const int size1,
                const char *string2, const int size2,
                const int start,
                struct re_registers *regs,
```

```
                const int stop)
```

이 함수는 re_match 와, 두개 데이터의 문자열과 크기를 넘겨주고, 이후의 매칭을 원하지 않을 경우의 인덱스 stop 을 제외하면 유사합니다.
re_match 처럼, re_match_2 가 성공적으로 수행되었다면, 문자열 string 에서 매칭된 횟수를 돌려줍니다. re_match 는 string1 과 string2 를, start 와 stop 인자를 설정하여 regs 를 사용할 때 에는 연속된 것으로 취급합니다.

```
int re_search_2 (struct re_pattern_buffer *buffer,
                 const char *string1, const int size1,
                 const char *string2, const int size2,
                 const int start,
                 struct re_registers *regs,
                 const int stop)
```

이것은 re_search 함수와 유사합니다.

6.3.6 fastmap 으로 검색하기

몇 십만바이트 이상 되는 문자열에서 검색을 하려면 fastmap 을 사용해야 합니다. 순차적으로 연속적인 위치에서 검색을 한다면 아마도 상당한 시간이 걸릴 것입니다. fastmap 은 내부적인 알고리즘을 유지하면서 최적의 검색을 수행합니다.

문자열 검색 시 효율을 높이기 위한 알고리즘은 많이 우리들에게 알려져 있습니다. 그들의 많은 부분들은 strstr과 같이 순차적으로 검색하는 것이 아니라 검색의 효율을 높이기 위해서 내부의 테이블을 갖추고 현재 위치의 문자가 검색의 시작점이 될 수 있는지를 검사하며 최대한의 포인터를 건너뛰도록 설계된 경우가 있습니다.

fastmap 을 이러한 역할을 하는 테이블에 대한 포인터입니다.
즉, 여러분들의 문자셋(아스키문자 등)으로 인덱스된 하나의 배열입니다. 아스키 encoding 하에서는, 따라서, fastmap 은 256 개의 원소를 가집니다. 주어진 패턴 버퍼에 있어서 검색시 fastmap 을 사용하려고 할 때에는, 먼저 배열을 할당하고 배열의 주소를 패턴버퍼의 fastmap 에 지정해야 합니다. fastmap 은 일반적으로 사용자가 직접 컴파일하거나 또는 re_search가 대신 할 수도 있습니다.fastmap 이 어떤 테이블을 가르키고 있다면, re_search 는, 컴파일된 패턴버퍼를 사용한 검색을 하기 이전에, 먼저 fastmap 을 자동적으로 컴파일합니다.

직접 수동으로 하려면 다음과 같이 사용하면 됩니다.

```
int re_compile_fastmap (struct re_pattern_buffer *pattern_buffer)
```

pattern_buffer 은 패턴버퍼의 주소입니다. 어떠한 문자 c 가 매칭에 있어서 시작점이 될 수 있다면, re_compile_fastmap 은 'pattern_buffer->fastmap[c]'를 0이 아닌 수로 지정을 합니다.
이 함수가 fastmap 을 컴파일 할 수 있다면 0을 리턴하고,내부에러일 경우에는 -2를 리턴합니다. 예를 든다면, 패턴버퍼 pattern_buffer 가 'a¦b' 를 컴파일한 패턴을 보유하고 있다면, re_compile_fastmap 은 fastmap['a'] 와 fastmap['b'] 를 세트한다는 것입니다. 'a' 와 'b' 는 매칭의 시작점이 될 수 있으니까요..

re_search 는 문자열의 각 원소들 중에서 fastmap 에 있는 것 중의 하나가 나올때까지 차례로 비교합니다. 그리고 나서 그 문자에서부터 매칭을 시도합니다. 매칭이 실패할 경우에는 이러한 처리를 반복합니다. 따라서 이렇게 fastmap 을 사용할 경우, re_search 는 매칭의 시작점이 될 수 없는 문자열의 위치에서 쓸데 없이 매칭하려고 하는 시도를 줄임으로써 시간을 절약할 수 있는 것입니다.

fastmap 을 re_search 에서 사용하기를 원치 않을 경우에는 fastmap 필드에 NULL (0)을 저장하면 됩니다. 물론 re_search 사용이전에 말이죠...

패턴버퍼의 fastmap 필드를 한번 초기화 했다면 다시 fastmap 을 컴파일 할 필요는 없습니다. re_search 는 fastmap 이 NULL 이면 컴파일을 하지 않으며, NULL 이 아니라면 새로운 패턴에 새로운 fastmap 을 컴파일합니다.

6.3.7 GNU 변환 테이블

패턴버퍼의 translate 필드를 변환 테이블로 설정하였다면, Regex 는 찾는 모든 문자열과 정규표현식에서 간단한 변환을 하기 위해 translate 로 지정된 변환 테이블을 사용합니다.

"변환테이블" 은 아스키와 같은 문자세트의 원소들로 인덱스된 배열입니다. 따라서 아스키코드에서는 변환테이블은 256개의 원소를 가집니다. 이 배열의 원소들도 마찬가지로 여러분의 문자세트에 포함이 됩니다. Regex 함수가 문자 c를 만났다면, 문자 c 대신 translate[c] 를 사용합니다.

가령 대소문자를 무시한 Regex 일 경우,

```
translate['A'] = 'A';
translate['a'] = 'A';
translate['B'] = 'B';
translate['b'] = 'B';
............
```

이렇게 값이 초기화 되어 있다면, 이후의 Regex 의 검색시 'a' 문자를 만난다면, 'a' 를 변환테이블의 인덱스로 하여 해당값으로 대신한다는 이야기입니다.
(translate['a'] 의 값은 'A' 이므로 'a' 대신 'A' 의 값을 적용함.)

그러나, 단한가지의 예외가 있다면, '\' 문자 뒤에 따라오는 문자는 변환하지 않는다는 것입니다. '\' 문자가 이스케이프의 역할을 한다면, '\B' 와 '\b' 는 항상 구별되는 것입니다.

이제 위와 같이, 소문자를 대문자로 변환하는, 대소문자를 무시하는 변환테이블을 초기화 하는 예를 보이겠습니다. (메뉴얼에 나와 있는 내용입니다. ^^)

```
    struct re_pattern_buffer pb;  /* 패턴 버퍼 */
    char case_fold[256];          /* 변환 테이블 */

    for (i = 0; i < 256; i++)
      case_fold[i] = i;
    for (i = 'a'; i <= 'z'; i++)  /* 소문자를 대문자로 변환 */
      case_fold[i] = i - ('a' -'A');

    pb.translate = case_fold;
```

이렇게 translate 에 변환 테이블의 주소를 지정하면 이후에변환테이블을 사용합니다. 변환테이블을 사용하고 싶지 않다면 translate 에 NULL 을 넣어주시면 됩니다. 만일, 패턴버퍼를 컴파일할 때나, fastmap 을 컴파일할 때, 패턴버퍼로 매칭이나 검색을 수행할 때 이러한 테이블의 내용을 바꾼다면 이상한 결과를 얻을 것입니다.

6.3.8 레지스터 사용하기

사실 이 부분이 regex 에서 중요한 부분입니다.

지금까지는 regex 를 사용하여 어떤 문자열내에 해당 패턴(정규표현식)이 있느냐 없느냐만 따졌으나 이 레지스터를 사용하면 세부 매칭의 결과를 저장하게 됩니다. 즉, 문자열 인덱스 어디에서 어디까지 패턴과 매칭이 되었는지에 대한 정보를 확보함으로써 나아가서는 문자열 치환 작업까지도 생각할 수 있습니다.

정규표현식에서 하나의 그룹은 전체적으로 정규표현식과 매칭되는 문자열의 하나의 부분문자열과 매칭할 수 있습니다. 매칭작업을 수행할 때 각각의 그룹과 매칭된 보조문자열의 시작과 끝이 기억됩니다.

이러한 검색이나 매칭시에는 GNU 매칭 및 검색 함수에 0이 아닌 'regs' 인자를 넘겨줘야 합니다.

```
struct re_registers {
  unsigned num_regs;
  regoff_t *start;
  regoff_t *end;
};
```

레지스터 옵셋 타입(regoff_t) 는 'int'를 형정의 한 것 입니다.

start 와 end 의 i번째 원소는 패턴에서의 i번째 그룹에 대한 정보를 기록합니다. 이 start 와 end 는 다양한 방법으로 할당되는 데, 이것은 패턴버퍼의 regs_allocated 필드에 의존합니다.

제일 간편하고 유용한 방법은 regex 의 매칭함수로 하여금 각각의 그룹에 대한 정보를 기록할 공간을 충분히 할당하게 하는 것입니다. regs_allocated 가 REGS_UNALLOCATED 라면, 매칭함수는 1+re_nsub(패턴버퍼의 다른 멤버) 만큼을 할당합니다. 여분의 원소는 -1 로 설정하고, regs_allocated 를 REGS_REALLOCATE 로 설정합니다. 이후에 다시 호출할 경우에, 필요하다면 매칭함수는 공간을 더 할당할 수 있습니다.

re_compile_pattern 은 regs_allocated 를 REGS_UNALLOCATED 로 설정하기 때문에, GNU 정규표현식 함수에서는 위와 같은 행동이 기본으로 되어 있습니다.

POSIX 에서는 조금 다릅니다. 호출자에 매칭함수가 채울, 고정길이의 배열을 넘겨줘야 합니다. 따라서 regs_allocated 가 REGS_FIXED 라면, 매칭함수는 그 고정배열을 간단하게 채웁니다.

아래의 예제는 re_registers 구조체에 기록되는 정보를 보여줍니다. ('(' 와 ')' 이 그룹오퍼레이터라고 하고, 문자열 string 에서 첫번째 문자의 인덱스를 0이라 하겠습니다.)

ㄱ. 정규표현식이 또다른 그룹을 포함하지 않는, i번째 그룹을 가지고 있다면, 함수는 'regs->start[i]' 에 그룹과 매칭하는 보조문자열의 시작 인덱스를 저장하고, 'regs->end[i]' 에는 보조문자열의 끝 인덱스를 저장합니다.
'regs->start[0]'과 'regs->end[0]' 에는 전체 패턴에 대한 정보가 들어갑니다.

예를 들면, 'ab' 에 대해 '((a)(b))' 를 매칭시킨다면, 다음의 결과를 얻을 것입니다.

* 0 in `regs->start[0]' and 2 in `regs->end[0]'
* 0 in `regs->start[1]' and 2 in `regs->end[1]'
* 0 in `regs->start[2]' and 1 in `regs->end[2]'
* 1 in `regs->start[3]' and 2 in `regs->end[3]'

ㄴ. 그룹이 반복오퍼레이터 등을 사용하여 한번보다 더 많이 매칭된다면, 함수는 마지막으로 매칭된 그룹에 대한 정보를 저장합니다.

예를 들면, 'aa' 에 대해 '(a)*' 를 매칭시킨다면, 다음의 결과를 얻을 것입니다.

* 0 in `regs->start[0]' and 2 in `regs->end[0]'
* 1 in `regs->start[1]' and 2 in `regs->end[1]'

여기에서 그룹 1 은 '(a)' 이지만, 뒤의 '*' 오퍼레이터로 인해 'aa' 와는 1번 초과 매칭되므로, 마지막에 매칭되는 'a'에 대한 인덱스를 기록합니다.

ㄷ. i번째 그룹이, 어떤 성공적인 매칭에 관여하지 않는다면, 반복오퍼레이터는 0번 반복을 허용하고, 함수는 'regs->start[i]' 와 'regs->end[i]' 를 -1로 채웁니다.

예를 든다면, 'b' 에 대해 '(a)*b' 를 매칭하는 경우는 다음의 결과를 얻을 것입니다.

* 0 in `regs->start[0]' and 1 in `regs->end[0]'
* -1 in `regs->start[1]' and -1 in `regs->end[1]'

여기에서 1번째 그룹인 '(a)' 는 매칭에 관여하지 않기 때문에 'regs->start[1]' 과 'regs->end[1]' 은 -1로 됩니다.

ㄹ. i번째 그룹이 길이가 0인 문자열을 매칭한다면, 함수는 regs->start[i] 와 regs->end[i] 를 "길이가 0인 문자열"의 인덱스로 설정합니다.

예를 든다면, 'b' 에 대해 '(a*)b' 를 매칭하는 경우는 다음의 결과를 얻을 것입니다.

```
* 0 in `regs->start[0]' and 1 in `regs->end[0]'
* 0 in `regs->start[1]' and 0in `regs->end[1]'
```

여기에서 '(a*)b' 는 위의 '(a)*b' 와는 다릅니다. 1번째 그룹인 '(a*)' 는 'b' 의 앞부분의 빈 문자열과 매칭이 되므로 regs->start[1]과 regs->end[1] 은 둘다 '0' 이 됩니다.

ㅁ. i번째 그룹이 j번째 그룹을 포함하고, j번째 그룹은 i번째 그룹에만 포함되는 경우, 함수는 i번째 그룹의 매칭을 기록하고, regs->start[j] 와 regs->end[j] 에는 j번째 그룹과 마지막으로 매칭된 것에 대한 정보를 기록합니다.

조금 햇갈리기 쉬운 경우인데, 예를 들어보지요..
'abb' 에 대해 '((a*)b)*'를 매칭시키는 경우를 봅시다.

regs->start[0] 과 regs->end[0] 은 당연히 전체 문자열의 정보를 가지므로 0(첫번째 문자 인덱스), 3(시작인덱스 + 길이로 보는 것이 좋을 듯..) 이 됩니다.

```
1:     ((a*)b)*       abb       : 1번째 그룹 첫 매칭 (0, 2)
       ^^^^^^^        ^^
       ((a*)b)*       abb       : 2번째 그룹 첫 매칭 (0, 1)
        ^^^           ^
```

처음의 매칭에서, 1번째 그룹은 'ab' 와 매칭되고, 2번째 그룹은 'a'와 매칭됩니다.

```
       ^^^^^^^
2:     ((a*)b)*       abb       : 1번째 그룹 둘째 매칭 (2, 3)
             ^          ^

        ^^^
       ((a*)b)*       a b b     : 2번째 그룹 둘째 매칭 (2, 2)
             ^          ^ (빈문자열)
```

반복 오퍼레이터의 영향으로, 두번째 매칭에서, 1번째 그룹은 'b'와 매칭되고, 2번째 그룹은 마지막 'b' 의 바로 앞의 빈문자열과 매칭됩니다.

따라서, 위의 'ㄴ' 규칙 (그룹이 반복오퍼레이터 등을 사용하여 한번보다 더 많이 매칭된다면, 함수는 마지막으로 매칭된 그룹에 대한 정보를 저장한다.) 에 따라, regs->start[1], regs->end[1] 에는 2, 3 이 각각 기록되며, 그룹1은 그룹2를 포함하고 그룹2는 그룹1에만 포함되기 때문에, 마지막으로 매칭된 2번째 그룹의 기록값인 2, 2가 각각 regs->start[2] 와 regs->end[2] 에 각각 기록됩니다.

결과를 정리하면,

```
* 0 in `regs->start[0]' and 3 in `regs->end[0]'
* 2 in `regs->start[1]' and 3 in `regs->end[1]'
* 2 in `regs->start[2]' and 2 in `regs->end[2]'
```

'abb' 에 대해 '((a)*b)*' 를 매칭한다면, 그룹2(괄호안쪽의 '(a)') 는 마지막 매칭에 관여하지 않으므로 다음과 같은 결과를 얻습니다.

```
* 0 in `regs->start[0]' and 3 in `regs->end[0]'
* 2 in `regs->start[1]' and 3 in `regs->end[1]'
* 0 in `regs->start[2]' and 1 in `regs->end[2]'
```

조금 햇갈리는 분도 계실 것이고, 재미를 느끼는 분도 계실 것입니다. :)

ㅂ. 위와 같을 경우에, 매칭이 되지 않을 경우, 함수는 regs->start[i], regs->end[i]와 regs->start[j], regs->end[j] 를 모두 -1 로 설정합니다.

예를 들면, 'c' 에 대해 '((a)*b)*c' 를 매칭한다면, 다음의 결과를 얻을 것입니다.

```
* 0 in `regs->start[0]' and 1 in `regs->end[0]'
* -1 in `regs->start[1]' and -1 in `regs->end[1]'
```

    * -1 in `regs->start[2]' and -1 in `regs->end[2]'

 레지스터에 대한 이야기는 이걸로 마치겠습니다.

6.3.9 GNU 패턴버퍼를 free 하기

패턴버퍼에서 할당된 필드를 free 하기 위해서는 이전에 설명드린 POSIX 함수 (6.2.6 의 regfree) 를 사용할 수 있습니다. POSIX 에서 사용하는 regex_t 는 GNU 의 re_pattern_buffer 와 동일합니다. 패턴버퍼를 free 한 이후에는, 어떤 검색과 매칭작업을 다시 수행하려면, 정규표현식을 패턴버퍼로 다시 컴파일하여야 합니다.

7. 나오는 말

이것으로 regex 프로그래밍 강좌는 마치겠습니다.

장시간 지루하셨을 텐데, 사실 정규표현식만 알려면 regex 프로그래밍 부분은 다 잊어버리셔도 상관없습니다. 다만 sed 나 awk 같은 정규표현식을 인식하는 응용 프로그램을 만드시려면 잘 익혀두시는 것이 좋습니다.

지금까지 Regex 에 대한 이야기를 쓰면서, Regex 메뉴얼의 번역도 하면서 (기본 자료가 없으므로..), 예제도 만들어가면서 진행해 왔는 데, 번역의 딱딱함을 많은 곳에서 느낄 수 밖에 없는 것 같아서 조금 안타깝지만, 그래도 Regex 의 단순 번역본보다는 조금이라도 알기쉽게 설명드리지 않았나 하는 것으로 위안을 삼아야겠군요. :) 마지막으로 regs 레지스터를 사용한 응용 소스를 시간상의 문제로 (사실 강좌를 너무 오래 끌었죠..^^) 추가하지 못한 점 아쉽지만 여러분들의 숙제로 남겨 두겠습니다.

그럼...

written by Han dong-hun
Mon Jun 9 17:01:42 KST 1997
ddoch@hitel.kol.co.kr

Good bye!!