Learning Regular Expressions

For precision of text manipulation and description, it's hard to beat the power of regexps.

by Giovanni Organtini

Imagine you are looking for a name in a telephone directory, but you can't remember its exact spelling. You can spend ages searching through all the possible combinations, unless you have a tool that extracts the relatively small number of options that matches your search, however incomplete it may be. Regular expressions are such a tool.

Roughly speaking, a regular expression (or regexp) is a string that describes another string or a group of strings. Several applications can profit from this ability: Perl, sed, awk, egrep and even Emacs (try Ctrl-Alt-% after reading this article) to name a few.

In fact, many of you already have used some sort of regular expression. In the shell command:

```
ls *.pl
```

the characters *.pl act as a regular expression. That is, it is a string that describes all the strings composed by any number of characters of any kind (*), followed by a period (.), followed by two given characters (pl).

The standard set of rules used for composing regular expressions is able to describe all strings, no matter how complicated they are. Unfortunately, life is always more complicated. It turns out that at least two different versions of regular expressions exist: extended and basic. Moreover, not all applications support all the possible rules.

Basics of Regular Expressions

A regular expression is said to match a given string if it correctly describes it. A given regular expression can match with zero to many strings. By convention, regular expressions are written between slashes (/.../). In what follows, I use extended regular expressions.

The simplest regular expression is a plain alphanumeric string. Such a regexp matches with all strings containing its content as a substring. As an example, consider the following verse from Cenerentola, my favorite opera by G. Rossini: "Zitto, zitto, piano, piano, senza strepito e rumore." The regexp /piano/ is said to match with the verse, because the latter contains the same characters, with the same sequence, of the regexp.

In order to better understand the examples I discuss, you can play with the following Perl script, trying variations of the regexp it contains:

```
#!/usr/bin/perl
$verse = "Zitto, zitto, piano, piano, senza " .
    "strepito e rumore";
if ($verse =~ /piano/) {
    print "Match!\n";
} else {
    print "Do not match!\n";
}
```

In Perl, the operator =~ compares two regular expressions and returns "true" if they match.
A few characters (called metacharacters) are not recognized as ordinary characters and are used for special purposes. The *, for example, is used to match zero or more times a group of characters that, in turn, is identified by a couple of parentheses defining an atom, or a group of characters that must be considered as a single entity. The regexp /\( piano,\)/ matches with the sample verse because the characters `piano,"`, forming an atom, are repeated twice. If the atom is composed of a single character, parentheses may be omitted.

The meaning of the * within a regular expression is different from the one it has in the shell. In regular expressions, the * is a modifier; it describes the multiplicity of the atom on its left. So, the string "piano" is matched by p* in a shell, but not within a regular expression: /p*/ matches with p, pp, ppp and so on, and even with a null string.

To specify that an atom's multiplicity ranges between N and M, the symbol \{N,M\} is used. \{N\} matches strings with exactly N repetitions of the preceding atom; \{N,\} will match at least N of them. So, the following regular expressions will match:

///\( piano,\)\{0,10\}/
///\( piano,\)\{1,2\}/
///\( piano,\)\{2\}/

Of course, the first regexp will match with "piano, piano, piano," too.

The metacharacters + and ? are shorthands, respectively, for \{1,\} and \{0,1\}.

Matched parenthesized atoms are automatically stored into special variables (called back references) identified by the symbol \ followed by a number. The first parenthesized atom occurrence in a regular expression will be stored in \1, the second in \2 and so on. For example:

///\Z\1\1\, ( piano,)\2///

will match the above-mentioned verse (imagine that \1 = "itto" and \2 = "piano,").

The . metacharacter can describe any character, so the regular expression /.\1to\, ./\1/ matches both "Zitto, zitto," and "zitto, zitto,". However, it even matches with "Ritto, ritto," which does not have the same meaning. To avoid being so generic, you can specify a set of possible alternatives, listing the possible characters in brackets:

///\[Zz\]\1\1\, [Zz]\2///

A dash in brackets is used to specify a range of characters. For example, /\[a-z\]/ matches all lowercase characters, and /\[A-Z\]/ matches all uppercase characters. /\[a-zA-Z0-9_\]/ matches any alphanumeric character or an underscore.

The metacharacter | can be used to express different alternatives. It works like a logical OR statement. Therefore:

///\Z\1\1\, ( piano,)\2///

will match with both "Zitto" and "zitto".

The metacharacters ^ and $ match, respectively, the beginning and the end of a string. If used inside brackets, the caret is interpreted as the negation operator. So:

///\[^a-z\]\1\1\, ( piano,)\2///

will match Zitto, but not zitto ([^a-z] can be read as "any letter that is not a lowercase letter").
To match a metacharacter it's enough to put a backslash (\) in front of it to tell the regexp to interpret it as an ordinary character. The \ character is often called an escape character.

**Using Regular Expressions**

To appreciate the power of regular expressions, let's look at a simple Perl script that helps system administrators look for authentication failures. For the following examples I used rather expressive regular expressions to show different features. You may write simpler ones to describe the same strings.

Each time someone fails to log in, syslogd writes messages to /var/log/messages that read like this:

```
Jul 26 16:35:25 myhost su(pam_unix)[2549]:
authentication failure; logname=verdi uid=500
euid=0
tty= ruser=organtin rhost= user=root
```

Jul 27 14:54:36 myhost login(pam_unix)[688]:
authentication failure; logname=LOGIN uid=0
euid=0 tty=tty1 ruser= rhost= user=mozart

These lines list the time at which the login attempt was made, the user who tried to log in as another user, if available, and the target user. For example, the user verdi tried to log in as root two times, while someone failed to log in as mozart from the console.

Consider the Perl script shown in Listing 1. It reads the /var/log/messages file, then identifies the lines that look interesting and extracts only the relevant information.

**Listing 1. Sample Perl Script for Finding Authentication Errors**

First of all, we select only relevant lines and match them with the regular expression /authentication failure/ shown on line 7. Everything else is discarded. Then each line is matched with a regular expression (line 8) that should be read as follows: take all the strings starting (^) with exactly three (\{3\}) alphabetic ([a-zA-Z]) characters, followed by a space, followed by at most two (\+) characters that could be either numeric (0-9, equivalent in Perl to the metacharacter \d) or a space. After a space, an arbitrary number (*) of digits or semicolons must follow. The portion of the string described so far is enclosed in parentheses, so it is stored in a back reference called \1 (it is the first one). After that, any number of characters (\.*) can be found before the string ``logname=''. That string must be followed by any number of alphanumeric characters. Again, because there are a couple of parentheses, we will store them in \2. Any number of characters, finally, can be present before the string ``user='', followed by any number of alphanumeric characters. This all gets stored into \3.

From this example, you can see how it is possible to extract substrings from strings. You do not need to know their relative positions, as long as you can describe their appearance.

Perl provides a helpful feature for working with regexps. The automagic definition of Perl variables named after the back references as S1, S2 and so on, can be used after a regular expression has been matched. Perl also lets users define useful symbols, such as \d or \w (equivalent to [A-Za-z0-9_]), as well as POSIX-compliant symbols representing the same things (see man perlre for more information).

**Basic Regular Expressions**

Basic regular expressions are used by several other programs, like sed or egrep.

In basic regular expressions, the metacharacters |, + and ? do not exist, and parentheses and braces need to be
escaped to be interpreted as metacharacters. The ^, $ and * metacharacters follow more complicated rules (see man 7 regex for more details). In most cases, however, they behave like their extended counterparts. It is often convenient to express the regular expression in the extended format, then add the escape characters when needed.

As an example, the script shown in Listing 2 generates an HTML-formatted page to read the content of system log files using an internet browser. Besides echoing HTML tags for the headers of the page and for a table, it simply lists files in a given directory and pipes the result to sed, which transforms it using a regexp. The syntax used by sed for text substitution is rather common and is something like:

```
s/regexp/replacement/
```

where regexp is a regular expression that must be replaced.

**Listing 2. Example Script for Generating and HTML-Formatted Page for Reading Log Files**

Essentially, the syntax represents a string composed of nine elements properly described by the appropriate regular expressions. For example `[rwxds-]` asks for the possible characters that can be found within the first element.

The latter part of the string consists of alphanumeric characters, with slashes interspersed. You may notice that the regular expression used in this case is `(.*/)(.*)`. The first group matches all characters preceding a (escaped) slash, i.e., the path name. The second group lists all the following characters (the filename). The number of slashes in the path doesn't matter. Regular expressions (both basic and extended), in fact, are said to be greedy--they try to match as many characters as possible.

The result of the script is written to standard output and can be redirected to a given file (by cron at fixed intervals, for example) to be shown on the Web.

**Conclusion**

Regular expressions are by far the most powerful tool for text manipulation and description, and they are well supported under Linux on many applications. Unfortunately, they are not supported at all (to my knowledge) by the most popular search engines because of their complexities. But, can you imagine how precise your search would be if you had the ability to describe the page you are looking for with a regular expression?

**Giovanni Organtini** ([g.organtini@roma1.infn.it](mailto:g.organtini@roma1.infn.it)) is a professor of Introduction to Computing and Programming for Physicists at the University of Rome. He has used Linux for years, both for fun and at work, where it is used for the simulation of the CMS experiment ([cmsdoc.cern.ch](http://cmsdoc.cern.ch)) on large farms and as part of a complex data-acquisition system and machine control. Before the birth of his son, Lorenzo, he used to travel, seeking good restaurants and attending concerts and operas.