Mastering Regular Expressions

Finding information in documents has always been an intimidating task; just finding documents themselves is tough at times. Operating systems have offered little help. In DOS, you could use the DIR command, such as DIR *.doc. This improved marginally in Windows with the Search feature. Microsoft Word offers the ubiquitous Find and Replace dialog, a convenient tool, if you know exactly what you're looking for. The problem scope creeps larger with increasingly huge disks, file proliferation, multiple users, and even multiple locations for a single user. For all this, how do you find specify information across many files locations, formats, and often when you don't know the specific text? That's were regular expressions come in. A regular expression (also called regex, and commonly but mistakenly as grep or wildcards) is a flexible method capable of finding exact text patterns in a file or files.

For instance, you'd like to make sure that a team member's name, Jeffrey, is spelled correctly each time. Forms you might want to look for would include *Jeff*, *Jeffrey*, *Jeffery*, *Geoffery*, and *Geoffrey*, assuming there are no other variations. In Word, you could do a conventional find, or likely four finds, for a single file but still end up with mismatches such as *jeffreyi*, *Mendelejeff*, or *Jeffers*. Now you want to search all the files on the Web server. Expanding the scope again, you want to change all the other occurrences to "Jeffery.", Finally, you want to make sure that *Jeffery* always appears as his full name *Jeffery Smith*. Clearly, the last three requests are not possible with conventional search tools and even the simplest request is impractical with 32,000 files. A regex could do this, and often in less than 15 seconds. In another example, you need to check all your files for matching <p> and <\p> statements and get a list of files that do not conform. These are both real world example of problems that need to be solved. The last example can be solved with a regex such as

```
% perl -0ne 'print $ARGV\n if s/\<p\>//ig != s/\<\\p\>//ig *
```

This statement likely doesn't make sense right now, it doesn't have to, but it does show the power of regex.

Mastering Regular Expressions (third edition) by Jeffrey E. F. Friedl attempts to make sense of these. Although it is part tutorial and part cookbook, it's more a story to introduce a regex mind set. By achieving full understanding, you can get new perspectives for creations solutions. This is consistent with the author's "teach a man to fish" parable. There are not a copious number of examples but the examples are complete and detailed. Studying them in detail. Once you learn regexes, you gain an invaluable skill and wonder how you ever got along without it. TiVo users already understand this feeling. The edition includes the latest versions of Perl (5.8.8), Java 1.6, .NET Framework 2.0. It is slightly dated, listing the US population as 298, 444,215.

There are several things to understand first.

- Regex are not a singular entity. It is generalized concept with individual implementations. Unix, POSIX, .NET, Java, Perl, Visual Basic, and VBScript, along others, each has its own version, and differ slightly.

- You will need a tool to use regex. As each implementation is different, you will need an explicit tool on your computer. Unix and POSIX have built in versions, .NET has libraries during compiling, Windows has many downloadable versions (many of them free). eGrep is a common application available at http://www.gnu.org/directory/grep.html. Windows has two older command line version still available (DOS isn't completely dead) called FIND and FINDSTR.

- Regexes are a programming language ranging from simple to extremely terse. Programmer-writers and programmers can make the most of these tools. However, this is not exclusive and all users can write their own expressions. A common case, Microsoft Word's Find and Replace dialog uses Wildcards (a form of regex). If nothing else, using eGrep is a better replacement for the Windows Search feature.

Expressions

Starting with simple forms of finding exact text such as "cat" finds all words with the letters c-a-t in that order. This includes Cat, catalogue and vacation. This means that regex are not word based. It's better to think of these searches patterns than as words.

Matching exact text is only the starting point. To match one of several characters using brackets ([]). To find *gray* or *grey*, search with *gr[ae]y*. *<H[123]>* finds HTML headings 1 (<H1>), 2 (<H2>), or 3 (<H3>), but no others. A dash indicates a range. *<H[1-3]>* has the same results. These can even be mixed. *[abcdA-Z04-8]* matches a,b, c,d, capital letters A through Z, the character zero, and the characters 4 through 8. The caret (called a negated class) excludes characters. *<H[^123]>* finds all headings except 1 through 3.

Taking pattern matching one step further, the dot character (.) finds any sequence of characters (this is similar to DOS' * such as DIR *.txt). Sloshing through Word-generated HTML becomes easier with statements such as *<span.>* which matches anything within a span statement.

To end the examples of basic pattern searches, there are repetition quantifiers. The plus sign (+) matches one or more of the immediately proceeding characters, and the asterisk (*) matches zero or more of the of the immediately proceeding characters (in other words, it would be alright not to match that character). While the difference sounds minor, the implications are profound. For instance, the <HR> tag can take several forms, including having superfluous spaces. such as <HR●> (● here represents a space character). A basic search of *<HR>* would not find those instances. To make the expression more flexible, use <HR●*> to match any number of trailing spaces. However, the <HR> statement can also take the form <HR SIZE=14> (along with any number of extra spaces). The search *<HR(●+SIZE●*=●*[0-9]+)?*●*> catches any HR statement regard of the SIZE value or spacing inside the tag.

The more you know about the target pattern, the better searches you can construct. Of course, there is a book's worth of additional options. Constructing the statements themselves is only the start. There are plenty of nuisances that can affect the search quality.

Programming

If single statements by themselves are powerful, then adding programmatic logic makes them more powerful and more versatile. For example, you can search for and extract all the HTML links such as

```
<a href="http://www.oreilly.com">O'Reilly Media</a>
```

Due to the possible complexities of an <a> tag, it's better to write code, analyzing the separate parts, each with its own regex. The simplified Perl code might look like this:

```
# Note: the regex in the while(...) is overly simplistic - see text for discussion
while ($Html =~ m{<a\b([^>]+)>(.*?)</a>}ig)
{
  my $Guts = $1; # Save results from the match above, to their own . . .
  my $Link = $2; #  . . . named variables, for clarity below.

  if ($Guts =~ m{
                  \b HREF        # "href" attribute
                  \s* = \s*      # "=" may have whitespace on either side
                  (?:            # Value is . . .
                    "([^"]*)"    #   double-quoted string,
                    |            #   or . . .
                    '([^']*)'    #   single-quoted string,
                    |            #   or . . .
                    ([^'">\s]+) #   "other stuff"
                  )              #
                 }xi)
  {
    my $Url = $+; # Gives the highest-numbered actually-filled $1, $2, etc.
    print "$Url with link text: $Link\n";
  }
}
```

To produce output such as

```
O'Reilly Media with link text: http://www.oreilly.com
```

Again, it’s not important to understand this example other than to show that coding extends the power of regex. You can see that each statement contains its own regex statement, making for flexible search and print statements.