electronic text center

Introduction to TEI

Using Regular Expressions

Stephen Ramsay, Assistant Director
Electronic Text Center, University of Virginia

What are regular expressions?

If you've ever typed "`cp *.html ../`" at the UNIX command prompt, or entered "`garden?`" into a web-based search engine, you've already used a simple regular expression. Regular expressions ("regex's" for short) are sets of symbols and syntactic elements used to match patterns of text.

Even these simple examples testify to the power of regular expressions. In the first instance, you've copied all the files which end in ".html" (as opposed to copying them one by one); in the second, you've conducted a search not only for "`garden`," but for "`garden`, `gardening`, `gardens`, and `gardeners`" all at once.

For a tool with full regex support, metacharacters like "`*`" and "`?`" (or "wildcard operators," as they are sometimes called) are only the tip of the iceberg. Using a good regex engine and a well-crafted regular expression, one can easily search through a text file (or a hundred text files) searching for words that have the suffix ".html" (but only if the word begins with a capital letter and occurs at the beginning of the line), replace the .html suffix with a .sgml suffix, and then change all the lower case characters to upper case. With the right tools, this series of regular expressions would do just that:

```
    s/(^[A_Z]{1})([a-z]+)\.sgml/\1\2\.html/g
    tr/a-z/A-Z/
```

As you might guess from this example, concision is everything when it comes to crafting regular expressions, and while this syntax won't win any beauty prizes, it follows a logical and fairly standardized format which you can learn to read and write easily with just a little bit of practice.

What sort of things can I do with regular expressions?

Regular expressions figure into all kinds of text-manipulation tasks. Searching and search-and-replace are among the more common uses, but regular expressions can also be used to test for certain conditions in a text file or data stream. You might use regular expressions, for example, as the basis for a short program that separates incoming mail from incoming spam. In this case, the program might use a regular expression to determine whether the name of a known spammer appeared in the "From:" line of the email. Email filtering programs, in fact, very often use regular expressions for exactly this type of operation.

And the drawbacks?

Regular expressions tend to be easier to write than they are to read. This is less of a problem if you are the only one who ever needs to maintain the program (or sed routine, or shell script, or what have you), but if several people need to watch over it, the syntax can turn into more of a hindrance than an aid.

Ordinary macros (in particular, editable macros such as those generated by the major word processors and editors) tend not to be as fast, as flexible, as portable, as concise, or as fault-tolerant as regular expressions, but they have the advantage of being much more readable; even people with no programming background whatsoever can usually make enough sense of a macro script to change it if the need arises. For some jobs, such readablitity will outweigh all other concerns. As with all things in computing, it's largely a question of fitting the tool to the job.

What do I need in order to use regular expressions?

Actually, you probably already have everything you need to start using regular expressions to get your work done. Regular expressions don't constitute a "language" in the way that C or Perl are languages or a tool in the way that *sed* or *grep* are tools; instead, regular expressions constitute a syntax which many languages and tools (including these) support.

Several languages, in fact, support regular expressions--Perl, Tcl, Python, *awk*, and the various shells naturally, but also many other popular languages (including C/C++, Java, and Visual Basic) with a little coaxing from libraries and whatnot. You don't need to be a programmer, however, to use regular expressions to the fullest. Several editors (including Nisus Writer, BBEdit, and every flavor of Emacs and *vi* you care to mention) and a great many text-manipulation tools used in UNIX (including *sed* and every flavor of *grep*) support regular expressions. *grep*, in fact, stands for **g**lobal **r**egular **e**xpression **p**rint.

Why are they called "regular expressions?"

Regular expressions trace back to the work of an American mathematician by the name of Stephen Kleene (one of the most influential figures in the development of theoretical computer science) who developed regular expressions as a notation for describing what he called "the algebra of regular sets." His work eventually found its way into some early efforts with computational search algorithms, and from there to some of the earliest text-manipulation tools on the Unix platform (including **ed** and **grep**). In the context of computer searches, the "*" is formally known as a "Kleene star."

How do I write a simple search pattern using a regular expression?

In a regular expression, everything is a generalized pattern. If I type the word "serendipitous" into my editor, I've created one instance of the word "serendipitous." If, however, I indicate to my tool (or compiler, or editor, or what have you) that I'm now typing a regular expression, I am in effect creating a template that matches all instances of the characters "s," "e," "r," "e," "n," "d," "i," "p," "i," "t," "o," "u," and "s" all in a row. The standard way to find "serendipitous" (the word) in a file is to use "`serendipitous`" (the regular expression) with a tool like *egrep* (or **e**xtended **grep**):

```
    $ egrep "serendipitous" foobar >hits
```

This line, as you might guess, asks *egrep* to find instances of the pattern "`serendipitous`" in the file "`foobar`" and write the results to a file called "`hits`".

How do I write a simple search-and-replace using regular expressions?

The process here is quite similar, and the general pattern tends to be the same from tool to tool. Suppose we wanted to find all instances of "`serendipitous`" in the file "`foobar`" and replace them with the word "fortuitous." You might use *sed* (which stands for **s**tream **ed**itor) like so:

```
$ sed 's/serendipity/fortuitous/g' foobar >hits.
```

In most regular expression "environments," the "`s`" operator (for "substitute") at the beginning tells the interpreter to substitute one pattern for another; "`g`" (for global) tells it to do so as many times as possible on a line.

How do I construct complex patterns?

In the preceding examples, we have been using regular expressions that adhere to the first rule of regular expressions: namely, that all alphanumeric characters match themselves. There are other characters, however, that match in a more generalized fashion. These are usually referred to as the *metacharacters*.

Single-Character Metacharacters

Some metacharacters match single characters. This includes the following symbols:

.	Matches any one character
[...]	Matches any character listed between the brackets
[^...]	Matches any character *except* those listed between the brackets

Suppose we have a number of filenames listed out in a file called "Important.files." We want to "grep out" those filenames which follow the pattern "`blurfle1`", "`blurfle2`", "`blurfle3`," and so on, but *exclude* files of the form "`1blurfle`", "`2blurfle`", "`3blurfle`" The following regex would do the trick:

```
$ egrep "blurfle." Important.files >blurfles
```

The important thing to realize here is that this line will *not* match merely the string "`blurfle.`" (that is, "blurfle" followed by a period). In a regular expression, the dot is a reserved symbol (we'll get to matching periods a little further on).

This is fine if we aren't particular about the character we match (whether it's a "1," a "2," or even a letter, a space, or an underscore). Narrowing the field of choices for a single character match, however, requires that we use a *character class*.

Character classes match any character listed within that class and are separated off using square brackets. So, for example, if we wanted to match on "`blurfle`" but only when it is followed immediately by a number (including "`blurfle1`" but not "`blurflez`") we would use something like this:

```
$ egrep "blurfle[0123456789]" Important.files >blurfles
```

The syntax here is exactly as it seems: "Find 'blurfle' followed by a zero, a one, a two, a three, a four, a five, a six, a seven, an eight, *or* a nine." Such classes are usually abbreviated using the range operator ("-"):

```
$ egrep "blurfle[0-9]" Important.files >blurfles
```

The following regex would find "`blurfle`" followed by any alphanumeric character (upper or lower case).

```
    $ egrep "blurfle[0-9A-Za-z]" Important.files >blurfles
```

(Notice that we didn't write `blurfle[0-9 A-Z a-z]` for that last one. The spaces might make it easier to read, but we'd be matching on anything between zero and nine, anything between a and z, anything between A and Z, *or a space*.)

A carat at the beginning of the character class negates that class. In other words, if you wanted to find all instances of blurfle *except* those which end in a number, you'd use the following:

```
    $ egrep "blurfle[^0-9]" Important.files >blurfles
```

Many regex implementations have "macros" for various character classes. In Perl, for example, `\d` matches any digit ([0-9]) and `\w` matches any "word character" ([a-zA-Z0-9_]). *Grep* uses a slightly different notation for the same thing: `[:digit:]` for digits and `[:alnum:]` for alphanumeric characters. The man page (or other documentation) for the particular tool should list all the regex macros available for that tool.

Quantifiers

The regular expression syntax also provides metacharacters which specify the number of times a particular character should match.

?	Matches any character zero or one times
*	Matches the preceding element zero or more times
+	Matches the preceding element one or more times
{*num*}	Matches the preceding element *num* times
{*min*, *max*}	Matches the preceding element at least *min* times, but not more than *max* times

These metacharacters allow you to match on a single-character pattern, but then continue to match on it until the pattern changes. In the last example, we were trying to search for patterns that contain "blurfle" followed by a number between zero and nine. The regex we came up with would match on `blurfle1`, `blurfle2`, `blurfle3`, etc. If, however, you had a programmer who mistakenly thought that "blurfle" was supposed to be spelled "blurffle," our regex wouldn't be able to catch it. We could fix it, though, with a quantifier.

```
    $ egrep "blur[f]+le[0-9]" Important.files >blurfles
```

Here we have "Find 'b', 'l', 'u,' 'r' (in a row) followed by one or more instances of an 'f' followed by 'l' and 'e' and then any single digit character between zero and nine."

There's always more than one way to do it with regular expressions, and in fact, if we use single-character metacharacters and quantifiers in conjunction with one another, we can search for almost all the variant spellings of "blurfle" ("bllurfle," "bllurrfle", bbluuuuurrrfffllle", and so on). One way, for example, might employ the ubiquitous (and exceedingly powerful) `.*` combination:

```
    $ egrep "b.*e" Important.files >blurfles
```

If we work this out, we come out with something like: "find a 'b' followed by any character any number of times (including zero times) followed by an 'e'."

It's tempting to use ".*" with abandon. However, bear in mind that the preceding example would match on words like "blue" and "baritone" as well as "blurfle."

Suppose the filenames in blurfle are numbered up to 12324, but we only care about the first 999:

```
$ egrep "blurfle[0-9]{3}" Important.files >blufles
```

This regex tells *egrep* to match any number between zero and nine exactly three times in a row. Similarly, "blurfle[0-9]{3,5}" matches any number between zero and nine at lest three times but not more than five times in a row.

Anchors

Often, you need to specify the position at which a particular pattern occurs. This is often referred to as "anchoring" the pattern:

^	Matches at the start of the line
$	Matches at the end of the line
\<	Matches at the beginning of a word
\>	Matches at the end of a word
\b	Matches at the beginning or the end of a word
\B	Matches any charater *not* at the beginning or end of a word

"^" and "$" are some of the most useful metacharacters in the regex arsenal--particularly when you need to run a search-and-replace on a list of strings. Suppose, for example, that we want to take the "blurfle" files listed in Important.files, list them out separately, run a program called "fragellate" on each one, and then append each successive output to a file called "fraggled_files." We could write a full-blown shell script (or Perl script) that would do this, but often, the job is faster and easier if we build a very simple shell script with a series of regular expressions. We'd begin by greping the files we want to operate on and writing the output to a file.

```
$ egrep "blurfle[0-9]" Important.file >script.sh
```

This would give us a list of files in script.sh that looked something like this:

```
blurfle1
blurfle2
blurfle3
blurfle4
.
.
.
```

Now we use *sed* (or the "/%s" operator in *vi*, or the "query-replace-regexp" command in *emacs*) to put "fragellate" in front of each filename and ">>fraggled_files" after each filename. This requires two separate search-and-replace operations (though not necessarily, as I'll explain when we get to backreferences). With *sed*, you have the ability to put both substitution lines into a file, and then use that file to iterate through another making each substitution in turn. In other words, we create a file called "fraggle.sed" which contains the following lines:

```
s/^/fraggelate /
s/$/ >>fraggled_files/
```

Then run the following "sed routine" on script.sh like so:

```
$ sed -f fraggle.sed script.sh >script2.sh
```

Our script would then look like this:

```
fraggelate blurfle1 >>fraggled_files
fraggelate blurfle2 >>fraggled_files
fraggelate blurfle3 >>fraggled_files
fraggelate blurfle4 >>fraggled_files
.
.
.
```

Chmod it, run it, and your done.

Of course, this is a somewhat trivial example ("Why wouldn't you just run "fragglate blurfle* >>fraggled_files" from the command line?"). Still, one can easily imagine instances where the criteria for the file name list is too complicated to express using [filename]* on the command line. In fact, you can probably see from this sed-routine example that we have the makings of an automatic shell-script generator or file filter.

You may also have noticed something odd about that caret in our sed routine. Why doesn't it mean "except" as in our previous example? The answer has to do with the sometimes radical difference between what an operator means inside the range operator and what it means outside it. The rules change from tool to tool, but generally speaking, you should use metacharacters inside range operators with caution. Some tools don't allow them at all, and others change the meaning. To pick but one example, most tools would interpret [A-Za-z.] as "Any character between A and Z, a and z *or a period*.

Most tools provide some way to anchor a match on a word boundary. In some versions of *grep*, for example, you are allowed to write:

```
$ grep "fle\>" Important.files >blurfles
```

This says: "Find the characters "f", "l", "e", but only when they come at the end of a word." `\b` tells the regex engine to match any word boundary (whether it's at the beginning or the end) and `\B` tells it to match any position that *isn't* a word boundary. This again can vary considerably from tool to tool. Some tools don't support word boundaries at all, and others support them using a slightly different syntax. The tools that do support word boundaries generally consider words to be bounded by spaces or punctuation, and consider numerals to be legitimate parts of words, but there are some variations on these rules that can effect the accuracy of your matches. The man page or other documentation should resolve the matter.

Escape Characters

By now, you're probably wondering how you go about searching for one of the special characters (asterisks, periods, slashes, and so on). The answer lies in the use of the escape character--for most tools, the backslash ("\"). To reverse the meaning of a special character (in other words, to treat it as a normal character instead of as a metacharacter), we simply put a backslash before that character. So, we know that a regex like "`.*`" finds any character any number of times. But suppose we're searching for ellipses of various lengths and we just want to find periods any number of times. Because the period is normally a special character, we'd need to escape it with a backslash:

```
$ grep "\.*" Important.Files >ellipses.files
```

Unfortunately, this contribute to the legendary ugliness of regular expressions more than any other element of the syntax. Add a few escape characters, and a simple sed routine designed to replace a couple of URL's quickly degenerates into confusion:

```
sed
```

```
    's/http:\/\/etext\.lib\.virginia\.edu\//http:\/\/www\.etext\.virginia\.edu/g
```

To make matters worse, the list of what needs to be escaped differs from tool to tool. Some tools, for example, consider the "+" quantifier to have its normal meaning (as a ordinary plus sign) until it is escaped. If you're having trouble with a regex (a sed routine that won't parse or a grep pattern that won't match even though you're certain the pattern exists), try playing around with the escapes. Or better yet, read the man page.

Alternation

Alternation refers to the use of the "|" symbol to indicate logical OR. In a previous example, we used "blur[f]+le" to catch those instances of "blurfle" that were misspelled with two "f's". Using alternation, we could have written:

```
    $ egrep "blurfle|blurffle" Important.files >blurfles
```

This means simply "Find either blurfle OR blurffle."

The power of this becomes more evident when we use parentheses to limit the scope of the alternative matches. Consider the following regex, which accounts for both the American and British spellings of the word "gray":

```
    $ egrep "gr(a|e)y" Important.files >hazy.shades
```

Or perhaps a mail-filtering program that uses the following regex to single out past correspondence between you and the boss:

```
    /(^To:|^From:) (Seaman|Ramsay)/
```

This says, "Find a 'To:' or a 'From:' line followed by a space and then either the word 'Seaman' or the word 'Ramsay'

This can make your regex's extremely flexible, but be careful! Parentheses are also metacharacters which figure prominently in the use of . . .

Backreferences

Perhaps the most powerful element of the regular expression syntax, backreferences allow you to load the results of a matched pattern into a buffer and then reuse it later in the expression.

In a previous example, we used two separate regular expressions to put something before and after a filename in a list of files. I mentioned at that point that it wasn't entirely necessary that we use two lines. This is because backreferences allow us to get it down to one line. Here's how:

```
    s/\(blurfle[0-9]+\)/fraggelate \1 >>fraggled_files/
```

The key elements in this example are the parentheses and the "\1". Earlier we noted that parentheses can be used to limit the scope of a match. They can also be used to save a particular pattern into a temporary buffer. In this example, everything in the "search" half of the sed routine (the "blurfle" part) is saved into a buffer. In the "replace" half we recall the contents of that buffer back into the string by referring to its buffer number. In this case, buffer "\1". So, this sed routine will do precisely what the earlier one did: find all the instances of blurfle followed by a number between zero and nine and replace it with "fragellate blurfle[some number] >>fraggled files".

Backreferences allow for something that very few ordinary search engines can manage; namely, strings of data that change slightly from instance to instance. Page numbering schemes provide a perfect example of this. Suppose we had a document that numbered each page with the notation <page n="[some number]" id n="[some chapter name]">. The number and the chapter name change from page to page, but the rest of the string stays the same. We can easily write a regular expression that matches on this string, but what if we wanted to match on it and then replace everything *but* the number and the chapter name?

```
    s/<page n="\([0-9]+\)" id="\([A-Za-z]+\)">/Page \1, Chapter \2/
```

Buffer number one ("\1") holds the first matched sequence, ([0-9]+); buffer number two ("\2") holds the second, ([A-Za-z]+).

Tools vary in the number of backreference they can hold. The more common tools (like *sed* and *grep*) hold nine, but Python can hold up to ninety-nine. Perl is limited only by the amount of physical memory (which, for all practical purposes, means you can have as many as you want). Perl also lets you assign the buffer number to an ordinary scalar variable ($1, $2, etc.) so you can use it later on in the code block.

Perl and Regular Expressions

Perl has evolved over the years into a flexible and sophisticated language capable of just about any programming task; including such "low-level language jobs" as large-scale application development and graphical user interface design. Still, there's no denying that it continues to dominate the field in the task for which it was originally designed: text manipulation. (Perl, as you may know, stands for "**P**ractical **E**xtraction and **R**eport **L**anguage"). Part of the reason it's so good at text manipulation comes from the fact that it has the most extensive support for regular expressions of any tool out there.

If you're a programmer who's new to regular expressions, you can probably imagine the advantage of using Perl as a regex "wrapper." As a full-blown programming language, Perl allows you to embed regular expressions in file tests, control loops, output formats, and everything else. Even if you're not a programmer, you can still use Perl and to enhance the capability of your regular expressions considerably.

Let me end with a brief code fragment which illustrates how one might use Perl to automate a text-manipulation task. The first uncompresses a file specified on the command line, runs a search-and-replace on the file, and then re-compresses it.

```
#!/usr/bin/perl -w

$file = $ARGV[0];

system( "uncompress $file" );

open( CURRENTFILE, "$file");
open( OUTFILE, ">outfile" );

while ( <CURRENTFILE> ) {


    $_ =~ s/ he / she /g;

    print OUTFILE $_;


}

close( CURRENTFILE );
close( OUTFILE );
```

This program, like Perl itself, combines the strengths of the shell with the power of regular expressions. The heart of the program is the `while ( <CURRENTFILE> )` loop, which tells the Perl interpreter to iterate through the file represented by the `CURRENTFILE` filehandle, making the specified substitution of "she" for "he" on each line. Outside the loop, we use the `system()` function to pass a command string to the shell.

A simple example, but one which gains significant utility when we expand the number of shell commands and the number of potential files. We might, for example, read an entire directory using `readdir()`, test for the presence of the ".Z" suffix (using a regex, of course), load those files into an array, and then iterate through each file in the array.

Perl also allows you to match on a string, save it into a buffer, evaluate the contents of that buffer, and perform a computation upon it. So for example, you might match on "page *n*" save the contents of *n* into a buffer as $1, and then use an expression like "`$newnumber += $1`" to increment the value of the page number by one.

Where can I get more information on regular expressions?

If you're looking for a book to read, you want **Mastering Regular Expressions** by Jeffrey E. F. Freidl (published by O'Reilly & Associates, Inc.). Friedl's book serves both as an extremely detailed tutorial and as an extremely detailed reference work on regular expression syntax. Get through this book, and you can consider yourself a serious expert on text manipulation in Unix.

Man pages and other forms of documentation abound for the tools which support regular expressions. The regex documentation for Perl is included with the distribution and can be found in "perlre.pod," but there are also versions of the documentation in Tex, html, pdf, and ascii format (visit CPAN, the Comprehensive Perl Archive Network for details).

If you're interested in regex libraries, you may want to check out GNU's **regex** package, available via ftp at ftp.gnu.org.

There are also a number of introductions to and summaries of regular expression syntax on the web. A search for "regular expressions" through any of the major web-based search engines should turn up dozens of them.

Maintained by the Electronic Text Center, University of Virginia
etext@virginia.edu **(804) 924-3230**