Advanced Unix Tools	**Lecture 11**
CS214 Spring 2002	**Wednesday, March 13, 2002**

We continue with our description of Perl. Today, we introduce more string operations, focusing on the all important *regular expression matching* capabilities. We then discuss how to interact with files.

1 Regular expressions

(I'll assume you have all seen basic regular expressions before.) The first way of using regular expressions in Perl that I'll discuss is as a test operator. The expression:

```
string =˜ /regexp/
```

where *string* is an arbitrary string (or a string variable, of course), and *regexp* is a regular expression. The above expression is true if any part of the given string matches the given regular expression. The following characters appearing in regular expression constrain the matching:

c	matches character *c*
ˆ	matches beginning of string
$	matches end of string
.	matches any one character, except newline
?	matches zero or one occurence of the previous character
*	matches zero or more occurences of the previous character
+	matches one or more occurences of the previous character
\d	matches a digit
\w	matches a letter, a digit, or undescore

There are more special characters, which you can lookup in the documentation. Any special character must be escaped with a \ if it is to be matched as that character, instead of interpreted as a regular expression metacharacter.

By default, matching is case-sensitive. Hence, *”foo bar” = /bar/* is true, while *”foo BAR” = /bar/* is false. To force matching to be case-insensitive, you can use the *i* flag, as follows: *”foo BAR” = /bar/i*, which is true.

As an generic example, consider the following regular expressions, which matches URLs to HTML files:

```
/ˆhttp:\/\/.+html$/
```

Notice the use of ˆ and *$* to signify that the URL should occur at the beginning of the string, and should be the whole string. Also, notice that the */* are escaped, as they otherwise signify the end of the regular expression.

One aspect of Perl regular expressions is they allow you, as a side-effect of matching ,to extract pieces of the string that matched parts of the regular expression. If you use grouping in your regular expression, that is, a subexpression of the form *(regexp)*, then whenever that part of the regular expression is matched, the substring that matched that subexpression will be remembered by Perl. Consider the URL example above, but this time, as a side-effect of matching, we would like to get our hands on the actual path to the HTML file (along with the host name). Add the appropriate grouping in the regular expression:

```
/ˆhttp:\/\/(.+html)$/
```

If you successfully match this expression with a string, then the part of the string that matches .+*html* will be saved in the special variable *$1*, as the following shows:

```
$test = "http://www.cs.cornell.edu/riccardo/index.html";
if ($test = /ˆhttp:\/\/(.+html)$/) {
  print "The path is $1\n";
}
```

What happens if there are more than one grouping in the regular expression? The substring matching each grouping is put in variables *$1*, *$2*, ... in left-to-right order of left-parentheses. Hence, *”abcdefghi” = /(..(..).).(..)/* will put *abcde* in variable *$1*, *cd* in variable *$2*, and *gh* in variable *$3*.

A second way of using regular expressions in Perl is as a replacement command. The command:

```
string-variable =˜ s/regexp/string/;
```

takes the content of the string variable, finds the first substring in that string that matches the regular expression, and replaces it with the given *string*, updating the value of the variable with the resulting string. Hence,

```
$test = "some foo and another foo";
$test = s/foo/bar;
print "$test\n";
```

The variable *$test* is updated with the result of replacing the first occurence of *foo* by *bar*. To replace all occurences of *foo*, you can use the flag *g*, as in *$test = s/foo/bar/g*. Notice that in the replacement string *string*, you can use the variable *$1*, *$2*, ... that have been set by the matching of the regular expression. As an example, consider the following example, which takes a string of the form *Last name, First name*, and switches the first and last name:

```
$test = "Doe, John";
$test = s/(\+), (\+)/$2 $1/;
print "$test\n";
```

File handles

Until now, the only way our Perl scripts can communicate with the outside world is by writing to standard output. Let's remedy that. All input and ouput in Perl is done via *handles*. There are two predefined handles, corresponding to standard input and standard output, appropriately named *STDIN* and *STDOUT*. We have already been using *STDOUT*, as it turns out that *print expr,expr,...* is just an abbreviation for *print STDOUT expr,expr,...*. This last form makes explicit what handle to send the output to.

How do you use *STDIN*? The basic way to perform input is to use the expression <*handle*>, which reads one line from the specified handle. Hence, <*STDIN*> will read one line from standard input. The following excerpt will read two lines from standard input and print the first:

```
$first = <STDIN>;
$second = <STDIN>;
print "$first\";
```

To read or write to files, you need to create an appropriate handle to the file. Let's consider input first. To create a handle to read from a file *foo.txt*, use the command:

```
open (NEWIN, "foo.txt");
```

This command creates a new handle *NEWIN* (you can choose whichever name you want) to read from file *foo.txt*. To read from the handle, simply use <*NEWIN*>. It is a good idea to close a handle when you're done using it, as follows:

```
close (NEWIN);
```

One can also read not from a file, but from the output of executing a shell command. For instance, assume that instead of wanting to read from *foo.txt*, you wanted to read from a sorted *foo.txt*. You can use:

```
open (ANOTHERIN, "cat foo.txt | sort |");
```

The expression is meant to remind you of a pipe, which in a sense it is.

To write to a file, you need to create an output handle. To create a handle to write to file *foo.txt*, you first need to decide whether you want to overwrite an existing *foo.txt*, or whether you want to append to an existing *foo.txt* (if it doesn't exist, it will be created). To overwrite, you use:

```
open (NEWOUT, ">foo.txt");
```

and to append, you use:

```
open (ANOTHEROUT, ">>foo.txt");
```

To write to a given handle, you use the full form of the *print* command, which takes an output handle as an argument:

```
print NEWOUT "this should be sent to foo.txt ", "and this too."
```

Note that *open* actually returns a result; *open* is true if the open operation was successful, and falso otherwise. Hence, you can catch errors, via, say:

```
if (! open (NEWIN,"foo.txt")) {
  print "ERROR!\";
  exit (1);
}
```