Advanced Unix Tools	**Lecture 8**
CS214 Spring 2004	**Wednesday March 10, 2004**

Conditionals and Loops

Some obvious operations are used to write comparison expressions. For example, the integer comparions $<$, $>$, $<=$, $>=$; the comparison operation == checks that two integer values are equal, `!=` checks that they are not. The string comparison operation `eq` checks that two strings are equal. Note that integer equality and string equality behave quite differently. For example, `"15" == 15` is true, since the string `"15"` is interpreted as the integer `15`. Similarly, `" 15 " == "15"` is also true, since both `" 15 "` and `"15"` are interpreted as the integer `15`. However, `" 15 " eq "15"` is false, since `" 15 "` and `"15"` are clearly not the same string (one has six characters, the other has only two).

Comparison expressions are useful in conditional and loop statements. A condition statement has the standard form:

```
if (comparison expression) {
  statement;
  ...
  statement;
} else {
  statement;
  ...
  statement;
}
```

The interpretation is as usual. Note that you can drop the `else` branch. The `while` loop is also pretty standard:

```
while (comparison expression) {
  statement;
  ...
  statement;
}
```

There are two flavors of `for` loops. The first is reminiscent of the `for` loop in `bash`. It allows you to iterate over a sequence of values:

```
for variable (value,...,value) {
  statement;
  ...
  statement;
}
```

The variable *variable* takes on the different values specified successively, and the statements in the body of the `for` are executed for each such value. For example, the following loop:

```
for $i (2,3,5,7,11,13,17,19) {
  print "Here's a prime: $i\n";
}
```

outputs the predictable

```
Here's a prime: 2
Here's a prime: 3
Here's a prime: 5
Here's a prime: 7
Here's a prime: 11
Here's a prime: 13
Here's a prime: 17
Here's a prime: 19
```

The other variant of `for` loop is reminiscent of the `for` loop found in the programming language C. It's syntax is slightly more complex:

```
for (setup; condition; increment) {
  statement;
  ...
  statement;
}
```

The idea is simple. First, the *setup* expression is executed, which typically sets up the index variable to some start value. Then the body of the `for` loop is executed, for as long as the *condition* is true. After every iteration, the *increment* expression is executed, which typically increments or decrement the index variable. For example, the following loop prints all even numbers from `0` to `30`:

```
for ($i = 0; $i<=30 ; $i=$i+1) {
  print "$i\n";
}
```

Regular Expression Matching

Let's now introduce more string operations, focusing on the all important *regular expression matching* capabilities. I'll assume you have all seen basic regular expressions before.

One way of using regular expressions in Perl that I'll discuss is as a comparison operation. 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`
`^`	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:

```
/^http:\/\/.+html$/
```

Notice the use of `^` 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:

```
/^http:\/\/(.+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 = /^http:\/\/(.+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`.