Regular Expression Substitution

Last time, we saw how regular expressions could be used as a matching operation.

Another 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);
}
```