1 Why Perl?

In an ideal world, statisticians could focus on statistical problems, without having to worry about getting programs to communicate with each other, extracting information from overly verbose output or rearranging data files to correspond to the needs of a particular program. Unfortunately, these kinds of problems do occur, and they are often crucial to the success of a project. Furthermore, with the increasing spread of the World Wide Web, many people are finding a need to write programs processing interactive web pages or using interprocess communication, as well as providing easier-to-use front ends for other programs. The perl programming language, developed by Larry Wall in the late 1980s, is emerging as the tool of choice for these tasks and many others. In this article I'm going to try to explain why perl is so popular – and so daunting to learn. I'll also provide some online resources which you may find helpful in your study of perl. In the next article in the series, I'll focus on some more advanced applications of perl.

2 Why People Love/Hate perl

A quick glance at the perl manual page or at some perl programs makes it clear that perl has borrowed quite a bit from many other languages. Its overall command structure is strongly reminiscent of the C programming language, while at the same time it contains many features generally only found in shell programming languages. But perl is neither an imitation of some other language or a slave to the latest programming trends.

Accordingly, it has some features which may not be familiar to even an experienced programmer. For example, perl functions can be called without arguments and parentheses are optional. When a function is called without arguments, the default argument is a variable known as "`$_`", which is generally the most recently read line from an input source. Although it takes some time to get used to these sorts of rules, it makes it very simple to perform many tasks. For example, to print each line of a file which is longer than 80 characters, the following one line perl program could be used:

```
perl -ne 'print if length > 80' filename
```

where `filename` is the name of the file in question. The `-n` flag tells perl to read each line of the file in turn, putting the current line's contents in `$_`, and the -e flag tells perl to read its program from the command line, not from a named file. Finally, the snippet of code tells perl to print those lines from file filename whose length (actually length($_) is greater than 80 .

Another unusual operator is the input operator, which, in its simplest form is a pair of angle brackets (<>). When this operator is "called" it reads a line of input into $_. Thus, a perl program like the following can be quite unnerving to an unsuspecting programmer:

```
while(<>){
     chop;
     @T = split;
     print if $T[2] eq "one";
     }
```

The program keeps reading until it reaches an end-of-file, removes the newline from each line (chop), breaks the line up into words (stored in the array @T), and prints the line if the third word was "one". (Perl arrays start at index 0, not 1). An additional feature of the input operator is that it is very smart about where it gets its input. If you invoke the program with no command line arguments, the operator reads from standard input, but if you list filenames on the command line, it will open each of the files in turn. Getting familiar with these features is well worth the effort, because you'll then have a single extensible programming tool which can meet all your scripting, text processing and network communication needs.

Every identifier (except for built in functions and commands) must be preceded by a symbol to identify its type to the perl interpreter. These different prefixes are displayed in Table 1. Each type of object has its own namespace. That means that there can can be an array, a subroutine, a scalar and an associative array all named x; perl can tell them apart by the way that they are used. One subtlety illustrated by the simple example program above is that when you refer to whole arrays (like @T in the example above), you precede them with an @ sign, while the individual elements of the array (like $T[2] in the example) are preceded by a $ sign. Note that the scalar object $T is a totally different thing, and has not been created or accessed by the above program.

Table 1: Prefixes of perl identifiers

Prefix	Meaning	Example
none	built-in function	`@T = split;`
&	subroutine	`$result = &mysub($x);`
$	scalar	`$color = "red";`
@	array	`join(" ",@values);`
%	associative array	`@names = keys(%info);`

3 Using perl under UNIX

There are a variety of ways to invoke perl in the UNIX environment. You can invoke perl directly from the command line, followed by the name of a perl program, but by far the most common way of running a perl program under UNIX is to put a line like the following as the first line of your perl program:

```
#!/usr/local/bin/perl
```

This line, which must be the first line of your file and which must begin in the first column, signals to the shell that the commands which follow should be interpreted by a program other than the shell. (The technique works for any language where, like perl, the pound sign (`#`) serves as a comment delimiter.) Then, by making the perl script executable by using the UNIX `chmod` command, you can execute your perl script by simply typing its name. In addition, as shown in previous examples, you can execute a short perl script through the command line using the `-e` flag.

Perl contains an excellent debugger, which is invoked by the `-d` flag, making it easy to pinpoint problems in your perl programs. One additional feature of the debugger should not be overlooked. If you invoke the debugger with an script consisting of just a 0, the debugger acts as a interactive perl "shell", allowing you to try different perl commands and see what the results are. Simply type

```
perl -de 0
```

at the UNIX prompt to get started.

4 A Quick Example

To give you a feel for why perl is so useful, I'll relate a problem which I saw a few days ago. A data tape in an unusual format had been downloaded to our system, and, although it was claimed to have a record length of 233, a quick check showed that somewhere along the line, the file had been broken up into lines of length 80, along with their associated newlines. Thus some fields were arbitrarily broken up between lines, making it very difficult to read the file correctly with even the most flexible program. But by using perl, it was easy to restore the file to its proper format. Here's the program I used:

```
#!/usr/local/bin/perl

while(<>){
        chop;
        $line .= $_;
        $n += length;
        if($n > 233){
                $out = substr($line,0,233);
                $n = length($line) - 233;
                $line =  substr($line,233,$n);
                print "$out\n";
                }
        }
        print $line if length($line) > 0;
```

I simply concatenated the input lines (after removing the trailing newlines with `chop`), and then wrote the file out in chunks of 233 characters. A more general solution would accept the new record length as a command line argument, as explained in a previous article in this series on Unix commands.

5 Regular Expressions

Much of perl's text processing power comes from its novel use of regular expressions. Central to the use of regular expressions is the special operator `= `. This operator takes its right hand argument and applies it to its left hand argument. In addition, statements using this operator return a 1 if a match takes place and a 0 otherwise. The right hand argument can either be a regular expression enclosed in slashes (`//`), in which case the left hand argument is unchanged, or it can be of the form `s/old/new/`, in which case the first occurence of a string matched by `old` is changed to `new`. (A `g` after the final slash changes all occurences, not just the first; similarly, an `i` performs a case insensitive search and can also be used with the `//` operator. As might be expected, if either of these arguments are presented without the `=` the searching or changing takes place with respect to `$_`, the usual default input. Once again, this can lead to some deceptively powerful one line programs. To change each occurrence of `one` to `two` in a file we could write:

```
perl -pe 's/old/new' filename
```

The `-p` flag is like the `-n` flag but it prints each line of the input file before reading the next.

To count the occurrences of the string "perl" in a file you could write

```
perl -ne '$i++ while s/perl//; print "$i\n" if eof' filename
```

Since the `s///` operator returns a value of one when it makes a change, `$i` gets incremented each time the string is found.

6 Online Resources

The start for most perl explorations should be the CPAN (Comprehensive Perl Archive Network), modeled after the invaluable CTAN archive for the TeX programming language. To find the location of the site nearest you, check `http://www.perl.com/perl/CPAN/CPAN.html`. Three tutorial introductions which are worth looking at are:

1. Patrick Ryan's brief and excellent "Introduction to Perl" which can be found at `http://www.eecs.nwu.edu/perl/SAG-perl.ps`.
2. Doug Bates' paper from the 1992 Interface meetings: `ftp://wingra.stat.wisc.edu/pub/doc/Data_in_perl.ps.gz`.
3. The slides from my 1994 Interface tutorial on perl: `ftp://www.stat.berkeley/pub/users/spector/perl_intro.ps.Z`