Advanced Unix Tools	**Lecture 10**
CS214 Spring 2003	**Monday, March 10, 2003**

Perl, the "Practical Extraction and Report Language", is a scripting language with powerful text-manipulation capabilities. Because of these, Perl has become a very popular language in the context of web servers. As you have seen doing homeworks 1 and 3, creating web pages involves hacking HTML, which is just a textual representation.

This week, we will cover the basics of Perl programming. We will be using Perl 5.6, /usr/local/bin/perl on babbage, although I doubt that we will ever talk about features specific to 5.6.

We will use Perl to write scripts, in a way similar to the way we have been writing shell scripts. We will store Perl scripts in text files with extension *.pl*, which is the conventional extension. There are two ways of executing a Perl script. The first is simply to invoke the Perl interpreter *perl* with the script to execute as an argument, e.g., *perl foo.pl*. The second way is to use the "magic number" approach. Recall that when Unix is instructed to execute a file and that file starts with the two special characters *#!*, it uses the remainder of the first line as an indication of what program to use to execute the file. In the case of shell scripts, we had scripts begin with *#! /bin/bash* to indicate that *bash* was to be invoked to execute the script. In our case, we can put *#! /usr/local/bin/perl* as the first line of our scripts, and set execute permissions on the script, and Unix will automatically invoke *perl* to execute the script.

Basics of Perl

A Perl script is a sequence of statements. There are a few forms of statements, most of which we will cover in the remainder of this lecture. Let me first note that Perl programming is not shell programming. Whereas shell programming involves putting together shell commands, Perl is a different language. Although they are syntactical similarities, you cannot directly use shell commands in Perl scripts.

Comments can be put anywhere in a Perl script. A comment starts with the character #. Anything on a line after a # is ignored by the Perl interpreter.

Perl recognizes two different types of values: integers and strings.[1] Integer literals are written as one expects, *1*, *2*, *-1*, ... A string is a sequence of characters. String literals can be written in two distinct ways. A string *'this is a string'* written using single-quotes is taken to mean the sequence of characters appearing as is between the single-quotes. A string *"this is also a string"* written using double-quotes is taken to mean the sequence of characters obtained after interpreting what's

[1] In fact, Perl 5.0 introduced a new type, objects, but we will not cover those in this course.

between the double-quotes. For most characters, interpretation doesn't affect anything. Hence, *"this is also a string"* represents the same string as *'this is also a string'*. We will see in a little while some examples on which single-quotes and double-quotes differ.

Perl allows the use of variables. A (scalar) variable can contains strings or integers. All scalar variables in Perl are required to start with the *$* characters. An assignment statement is used to assign value to variables. For example,

```
$i = 10;
$word = 'hello';
```

Notice that statements are always terminated with a semicolon *;*. Also, notice that *$* is part of the variable name, and hence must be used even when assigning to a variable. (This is different than the use of variables in the shell, a frequent beginner's problem.) One of the difference between single-quotes and double-quotes for defining strings is how to handle variables appearing in the string being defined. Compare the following definitions:

```
$test1 = 'this is the value of variable i: $i';
$test2 = "this is the value of variable i: $i";
```

Since *$test1* is defined as a literal string, all the characters between the single-quotes are part of the string. However, *$test2* is interpreted, which among other things means that variables appearing inside the string are replaced by their value. Hence, the *$i* in *$test2* is replaced by *10*, so that the string stored in *$test2* is in fact the string *this is the value of variable i: 10*.

Variable assignment is our first example of statement. A statement that comes in handy is *print* that simply prints values to standard output. In fact, *print* is an example of a built-in function. Perl has many built-in functions, and we will describe some of them as we encounter them in examples. The basic form of *print* is:

```
print expr,expr,...,expr;
```

where *expr*s can be integers or strings, or in general any expression yielding such. (We will define useful operations to build expressions shortly.) The values printed are printed without spaces in between them. Also, *print* does not put newlines after printing. If you want a newline, you need to explicitely use *"\n"*. (Notice the double-quotes; if you use single-quotes *'\n'*, then you simply output the characters \ followed by *n*; you want to interpret it as a newline, requiring you to use double-quotes.) Here are some examples:

```
$i = 42;
print "Hello,\n";
```

```
print "This is the value ";
print "of integer variable i:", $i, "\n";
```

Executing the above script outputs the following to stdout:

```
Hello,
This is the value of integer variable i: 42
```

Operations are useful for constructing expressions. Some operations assume that their arguments are integers, such as +, -, *, /, and others. If these operations are given arguments that are strings, Perl will automatically convert those values to integers before applying the operation. If the string cannot be converted to a meaningful integer, then *0* is used. Consider the following example:

```
$a = "10"; $b = $a + 2; $c = $b + " 34 ";
```

The variable *$c* gets final value *46*. Other operations assume that their arguments are strings. The basic example of such an operation is . representing string concatenation. Hence, *"foo" . "bar"* yields the string *"foobar"*. As before, if an argument is not a string, it is converted to a string prior to application of the operation. This leads to some cute behavior:

```
$a = 5;
$b = $a + 10;
$c = $b . "200";
$d = $c + 10;
```

You can check that the variables are assigned the following values: *$a* gets *5*, *$b* gets *15*, *$c* gets *15200* (!), *$d* gets *15210*.

Other 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";
}
```