B211 Internet Computing

Perl Language

Learning Objective

- To know enough about the syntax of the Perl programming language to write Perl/CGI scripts.

Lecture Outline

- Why Perl?
- Basic Perl programs
- Variables
- Control Structures
- Subroutines
- File Handling

CGI Programming

- In the last lecture, we went through the relationship between a web browser, a web server, CGI scripts and HTML documents.
- But to get a CGI script to do something interesting, you will need to know enough to write the script.
- CGI applications can be written in most programming languages.
  - They have facilities (in the form of libraries, modules, packages, etc) similar to the CGI module in Perl, and
  - They can do raw environment processing such as on page 8 in the last lecture.

Purpose of this lecture

- The purpose of this lecture is to introduce enough Perl syntax for you to understand and code basic CGI scripts.
- I will show how to use Perl for certain task, and for the sake of avoiding confusion among those of you who are not programming-oriented, I will only present one way.
  - Just keep in mind that this is Perl, there are ALWAYS MANY other ways of doing the same task.
  - You will see many different approaches as you read more and more example code from different sources.
  - Those who are doing their Computer Science major should explore these alternate ways to increase your understanding of programming.

Advantages of Perl for CGI

- Comparisons of the different alternatives to server-side scripting can get very contentious
  - we will look at some of those comparisons in Topic 4.
  - You can also read pages 188-189 in the textbook.
- The main advantages for using Perl (for any purpose) are:
  - Rapid development - can have very terse code.
  - There are literally hundreds of ways to do any one thing - programmers develop their own unique (most efficient?) way of doing things.
  - Powerful text handling facilities through reg exp.
  - Hash as a fundamental data type.

Disadvantages of Perl

- Unfortunately, most of its biggest strengths are also its weaknesses
  - Because it is terse
    - Code can be completely undecipherable by non-Perl programmers
    - Less experienced programmers tend to write really bad code.
  - Because there are so many ways to do the same thing, no one can agree on the standard way
    - Hard in team work situations.
  - Because Perl programmers have concentrated on text-handliing, graphical development have not been as intense as it may otherwise be.

A Basic Perl Script

```
#!/usr/bin/perl

print "Hello!" ;
```

Indicates where the perl interpreter can be found. This “shebang” statement must be at the beginning of every script.

A statement to print something to the the standard output.

Comments and Statements

- When a # symbol is encountered, anything from the # to the end of the line is ignored (with the exception of the first line).
  - To comment multiple lines, use a # on each line.
- All statements in Perl end with a semicolon.
- There is no concept of a separate “main” program in Perl.
  - The statements starting from the second line gets executed from top to bottom, unless they are functions/subroutines.

Variables

- The 3 main forms of variables in Perl are:
  - Scalars - single values
  - Arrays - a list of values
  - Hashes - collection of values indexed using “keys”
- The “my” keyword indicates that a variable is a local variable.

Scalar Variables

- A scalar variable is represented by a name with the $ symbol in front of it.
- A scalar variable can hold a single string and or a number.
- As in JavaScript, you do not have to declare whether a scalar is a string or a number.
- Eg.

```
$num = 21 ;
$str = ‘a string’ ;
my $var = 21 + 48 ;
my $var = ‘a’ . ‘string’ ;
```


Some Common Operations

Arithmetic operators:

```
$a = 1 + 2;      # Add 1 and 2 and store in $a
$a = 3 - 4;      # Subtract 4 from 3 and store in $a
$a = 5 * 6;      # Multiply 5 and 6
$a = 7 / 8;      # Divide 7 by 8 to give 0.875
$a = 9 ** 10;    # Nine to the power of 10
$a = 5 % 2;      # Remainder of 5 divided by 2
$a++;            # Return $a and then increment it
$a--;            # Return $a and then decrement it
```

String Operations:

```
$a = $b . $c;    # Concatenate $b and $c
$a = $b x $c;    # $b repeated $c times
```

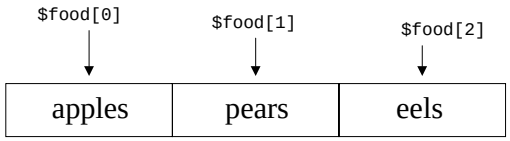
Assignments:

```
$a = $b;                # Assign $b to $a
$a += $b;        # Add $b to $a
$a -= $b;        # Subtract $b from $a
$a .= $b;        # Append $b onto $a
```

Array Variables

- Arrays are lists of scalars (ie numbers and strings).
- An array name have an @ symbol in front of it.
- Eg.

```
@food = ("apples", "pears", "eels");
my @numbers = (1,2,3,4) ;
```



* Note that you use $food[2] instead of @food[2]. This is because the element is a scalar.

Some Common Array Operations

```
push(@food, "eggs");              # add "eggs" to the end of the array @food.
push(@food, "eggs", "lard");      # add 2 more to the end of the array @food.
$grub = pop(@food);               # take the last element out of array, and
                                  # assign to $grub

$f = @food;                       # assigns the length of @food
$f = "@food";                     # turns the list into a string with a space
                                  # between each element.

($a, $b) = ($c, $d);              # Same as $a=$c; $b=$d;
($a, $b) = @food;                 # $a and $b are the first two items of @food.
($a, @somefood) = @food;          # $a is the first item of @food
                                  # @somefood is a list of the others.
(@somefood, $a) = @food;          # @somefood is @food and
                                  # $a is undefined.

print @food;                      # display the list.
```


Hashes

- Perl also allows us to create arrays which are accessed by a string rather than a number. These are called *hashes* (or *associative arrays*).
- A hash is prefixed by a % symbol.
- Eg.

```
%ages = ("Michael Caine" => 39,
         "Dirty Den" => 34,
         "Angie" => 27,
         "Willy" => "21 in dog years",
         "The Queen Mother" => 108);
```


Hashes

- From the previous example:

  `$ages{"Michael Caine"}` returns 39

  `$ages{"Dirty Den"}` returns 34

  `$ages{"Angie"}` returns 27

  `$ages{"Willy"}` returns "21 in dog years"

  `$ages{"The Queen Mother"}` returns 108

- Note again the use of the symbol $ instead of % to access the elements.

Hashes

- All the strings used to do the indexing are called *keys*.
- The function `keys` can be used to get all the keys in a hash. Eg.

```
keys(%ages)
```

gives

```
("Michael Caine", "Dirty Den", "Angie",
     "Willy", "The Queen Mother")
```


Control Structures

- Like most other languages, Perl has control structures for conditionals and looping
- We will look at examples of common ones:
  - if…elsif…else
  - foreach
  - for
  - while
  - do…while

if statements

- For executing statements based on a condition
- Eg.

```
if ($a == $b) {
        print "The numbers are equal\n";
} else {
        print " The numbers are not equal\n";
}
```


Test Operators

```
$a == $b     # Is $a numerically equal to $b?
             # Beware: Don't use the = operator.
$a != $b     # Is $a numerically unequal to $b?
$a eq $b     # Is $a string-equal to $b?
$a ne $b     # Is $a string-unequal to $b?

($a && $b)   # Is $a and $b true?
($a || $b)   # Is either $a or $b true?
!($a)        # is $a false?
```

elsif

- If more than one condition:

```
if (!$a) {
    print "The string is empty\n";
} elsif (length($a) == 1) {               # If above fails, try this
    print "The string has one character\n";
} elsif (length($a) == 2) {               # If that fails, try this
    print "The string has two characters\n";
} else {                                  # Now, everything has failed
    print "The string has lots of characters\n";
}
```

- Note that “elsif” is NOT spelled “elseif”!

foreach loop

- To go through each line of an array or other list-like structure (such as lines in a file)
- Eg.

```
foreach $yum (@food) {    # Visit each item in turn and call it $yum
        print "$yum\n";
}
```

```
foreach $name (keys %ages) {
        print "$name $ages{$name}\n";
}
```


foreach loop

- Unlike most programming languages, even when your control block has one statement (examples above), you MUST still put the curly brackets.

for loops

- General form:

```
for (initialise; test; inc) {
        first_action;
        second_action;
        etc
}
```

- Eg.

```
for ($i = 0; $i < 10; ++$i)       # Start with $i = 1
                                  # Do it while $i
{                                 # Increment $i before repeating
        print "$i\n";
}
```

Subroutines

- In Perl, we define encapsulated blocks of code by using subroutines
  - We call them functions in Perl as well.
- You may place subroutines anywhere in the code.
  - But we usually have the “main” statements at the beginning, and place all subroutines at the end.

Example Subroutines

```
#!/usr/bin/perl

&mysubroutine;   # Call the subroutine

sub mysubroutine {
        print "Not a very interesting routine\n";
        print "This does the same thing every time\n";
}
```

```
#!/usr/bin/perl

print &add(1,2); # Call the subroutine with parameters and
                 # print the result


sub add {

        my ($a, $b) = @_ ; # read in the parameters into $a and $b
        return $a + $b ;
}
```


File Handling

- The simplest way to do file handling is to open a file and read the whole content into an array as strings. We can then process that array.
- Eg. Reading from a file

```
open(INFO, ’passwd.txt’);         # Open the file
@lines = <INFO>;                  # Read it into an array
close(INFO);                      # Close the file
print @lines;                     # Print the array
```


Different ways of opening files

```
open(INFO, $file);        # Open for input
open(INFO, ”>$file");     # Open for output
open(INFO, ”>>$file");    # Open for appending
open(INFO, "<$file");     # Also open for input

print INFO "This line goes to the file.\n"; # print into a file opened
                                            # for output or appending
```

Different ways of passing parameters

- There are two basic ways passing parameters to standard functions like `print` and `keys`.
- The following are equivalent:

```
print INFO “This is a line” ;
print(INFO,“This is a line”) ;
```

- This applies to all other standard functions as well.